Exploring strategies to improve FPGA design with higher levels of abstraction

By

TOBÍAS ALONSO PUGLIESE

PH.D. THESIS

UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Departamento de Tecnología Electrónica y de las Comunicaciones

Ph.D. Advisors:
Dr. Gustavo Sutter Capristo
Dr. Jorge E. López de Vergara Méndez

MADRID, MAY 2022

A mi Luky, quien todos estos años ha estado siempre a mi lado.

ABSTRACT

While our data processing needs continue to grow at a fast pace, the capacity of general purpose hardware to increase their compute power is decreasing with each successive technology generation. Then, it becomes more important to resort to other means to improve performance. Custom designed hardware, in particular, implemented in FPGAs, has been a vital tool in many application areas and, under the current circumstances, it might be essential to push performance forward. In this context, this thesis explores strategies that can be used in the different hardware design stages to enhance performance of custom-designed solutions using FPGAs and the ability of hardware designers to tackle larger hardware design problems. The work is focused on three application areas, image processing, network packet processing and artificial intelligence, where system requirements and design challenges are analyzed, and later addressed, proposing solutions.

In the area of image processing, we tackle compression under stringent constraints, like those faced in satellite and drone sensing. Given that available image compression algorithms do not properly address the problems in these scenarios, an algorithm-hardware co-design approach was followed. We proposed a series of enhancements to the JPEG-LS standard, aimed at improving its coding efficiency at a low computational overhead, and later, developed an FPGA implementation of the encoder. As a result, a low latency near-lossless compressor with the highest pixel rate and highest compression ratio was achieved, outperforming existing implementations.

In the area of network packet processing, we focus on the problem of offloading the flow metering process for 100 Gbit Ethernet. Although for slower link rates a high-level synthesis implementation can provide a solution, for the target link and relevant applications that require managing large memories, this route was insufficient. For this reason, an alternative architecture was designed to address the problem of implementing high-throughput and complex read-update processes dealing with significant propagation delays associated with the memory system. This allowed to implement in an FPGA a TCP flow metering system supporting 100GbE packet rate and achieving over a 50%

offload of the computational load. We also demonstrate that using arrays of the developed cores enable the use of larger flow tables, increasing the offloading capabilities, while still supporting the maximum packet rate.

In addition, we studied a complementary throughput optimization for read-update processes, the conditional stalling technique. We concluded that, with efficient implementations, its utilization will not have frequency penalties for most designs and that few extra resources are required. Also, we examined the performance of the technique as a function of input data and architecture characteristics, showing that, even in adverse cases, it can significantly enhance performance. However, we demonstrated that to optimize throughput, we must take into account both the memory address statistics and the evolution of frequency as the pipeline is deepened.

In the area of artificial intelligence, we deal with a set of related implementation problems that arise when scaling up convolutional neural network dataflow accelerators, in particular in non-monolithic (multi-SLR) FPGAs. To tackle them, we developed a partitioning and resource balancing optimization tool. This tool addresses the control signals connection of large designs in multi-SLR FPGAs, and balances multiple resources across FPGA regions and/or chips, while it minimizes the communication cost among them. The tool natively maps systems to a multi-node implementation if it does not fit in a single FPGA, and supports different multi-node paradigms. The application of this optimization significantly enhanced performance of the accelerators. Targeting multi-node platforms led to further compute density, latency and power improvements.

Finally, from the experience obtained in the development of these applications, we identified key methodological aspects that led us to a successful hardware implementation using high-level synthesis (HLS). In particular, we observed that HLS can achieve great quality of results, leveraging wider algorithmic exploration and function specialization. Also, we identify the benefits of modular partitioning and refinement, and using a hardware oriented development mentality.

Resumen

Mientras nuestras necesidades de procesamiento de datos siguen creciendo a gran velocidad, la capacidad del hardware de propósito general para aumentar su potencia de cálculo va disminuyendo con cada generación tecnológica sucesiva. Entonces, se hace más importante recurrir a otros medios para mejorar el rendimiento. El hardware diseñado a medida, en particular, implementado en FPGAs, ha sido una herramienta vital en muchas áreas de aplicación y, en las circunstancias actuales, puede ser esencial para impulsar el rendimiento. En este contexto, esta tesis explora las estrategias que se pueden utilizar en las diferentes etapas de diseño de hardware para mejorar el rendimiento de las soluciones diseñadas a medida utilizando FPGAs y la capacidad de los diseñadores de hardware para abordar problemas de diseño de hardware más grandes. El trabajo se centra en tres áreas de aplicación, el procesamiento de imágenes, el procesamiento de paquetes de red y la inteligencia artificial, donde se analizan los requisitos del sistema y los retos de diseño, para posteriormente abordarlos, proponiendo soluciones.

En el área de procesamiento de imágenes, abordamos la compresión bajo estrictas restricciones, como las que se plantean en la exploración de satélites y drones. Dado que los algoritmos de compresión de imágenes disponibles no abordan adecuadamente los problemas en estos escenarios, se siguió un enfoque de codiseño algoritmo-hardware. Propusimos una serie de mejoras en el estándar JPEG-LS, destinadas a mejorar su eficiencia de codificación con una baja sobrecarga computacional, y luego, desarrollamos una implementación FPGA del codificador. Como resultado, se consiguió un compresor de baja latencia y con pérdidas matemáticamente acotas (near-lossless) con la mayor tasa de píxeles y la mayor relación de compresión, superando las implementaciones existentes.

En el área del procesamiento de paquetes de red, nos centramos en el problema de descargar el proceso de medición de flujos para 100 Gbit Ethernet. Aunque para velocidades de enlace más lentas una implementación de síntesis de alto nivel es capaz de proporcionar una solución, para el enlace objetivo y aplicaciones relevantes que requieren la gestión de grandes memorias, esta vía fue insuficiente. Por este motivo, se diseñó una arquitectura alternativa para abordar el problema de la implementación de

procesos de lectura-actualización complejos y de alto rendimiento que se enfrentan a importantes retardos de propagación asociados al sistema de memoria. Esto permitió implementar en una FPGA un sistema de medición de flujos TCP capaz de soportar la tasa máxima de paquetes de 100 GbE y logrando una descarga de más del 50% de la carga computacional. También demostramos que el uso de arreglos de los módulos desarrollados permite utilizar tablas de flujos más grandes, lo que aumenta la capacidad de descarga, sin dejar de soportar la tasa máxima de paquetes.

Además, estudiamos una optimización de rendimiento complementaria para los procesos de lectura-actualización, la técnica de bloqueo condicional. Llegamos a la conclusión de que, con implementaciones eficientes, su utilización no tendrá penalizaciones de frecuencia para la mayoría de los diseños y que se requieren pocos recursos adicionales. Además, examinamos el rendimiento de la técnica en función de las características de los datos de entrada y de la arquitectura, mostrando que, incluso en casos adversos, puede mejorar significativamente el rendimiento. Sin embargo, demostramos que, para optimizar el rendimiento, hay que tener en cuenta tanto las estadísticas de direcciones de memoria, como la evolución de la frecuencia a medida que se incrementa la segmentación.

En el área de la inteligencia artificial, abordamos un conjunto de problemas de implementación relacionados que surgen al escalar aceleradores de redes neuronales convoluciones segmentados, en particular, en FPGAs no monolíticas (multi-SLR). Para abordarlos, desarrollamos una herramienta de optimización de particionamiento y balanceo de recursos. Esta herramienta determina la conexión de las señales de control de grandes diseños en FPGAs multi-SLR, y equilibra múltiples recursos a lo largo de las regiones y/o FPGAs, al tiempo que minimiza el coste de comunicación entre ellas. La herramienta mapea de forma nativa a una implementación multi-nodo si el diseño no cabe en una sola FPGA, y soporta diferentes paradigmas multi-nodo. Esta optimización mejoró significativamente el rendimiento de los aceleradores. La utilización de plataformas multi-nodo permitió mejorar la densidad de cálculo, la latencia y el consumo de energía.

Por último, a partir de la experiencia obtenida en el desarrollo de estas aplicaciones, identificamos aspectos metodológicos clave que nos condujeron a una exitosa implementación de hardware utilizando la síntesis de alto nivel (HLS, por sus siglas en inglés). En particular, observamos que HLS puede lograr muy buenos resultados, aprovechando una mayor exploración algorítmica y especialización de funciones. Además, identificamos las ventajas del particionamiento y refinamiento modular, y de la utilización de una mentalidad de desarrollo orientada al hardware.

AGRADECIMIENTOS

Esta tesis refleja el trabajo de poco más de cuatro años de mi vida. Claro es que si tuviese que nombrar a todas aquellas personas que lo influenciaron, el largo de la misma se duplicaría. Cuantas conversaciones afectaron mi forma de pensar, muchas con gente que nunca conocí. Dicho esto, hay personas que no puedo dejar de mencionar.

En primer lugar, quiero que agradecer a mi compañera en la vida, Luky, quien siempre estuvo al lado mío para luchar las más difíciles batallas. Sin ella, no veo como podría hoy estar cantando victoria. En estas batallas, nunca faltó mi familia. Mi mamá, Silvia, mi papá, Julio, y mi hermana, Tami. A quienes se sumaron mi nueva familia Melián. Aún a la distancia, a un centímetro de mí cuando los necesité.

En el campo, en Madrid, Alberto, Silvia, Rafa, Moni y Franco. Gracias por su amistad, sus consejos y los buenos momentos. En especial, menciono a Mario, gran amigo y compañero de laboratorio. Gracias por los consejos, discusiones, charlas y compañía en el laboratorio, en Madrid, en Argentina y en Dublín. Un miembro especial de mi equipo es Noela, a quien agradezco profundamente por prestarme su oído y ayudarme a crecer.

También, doy las gracias a Gustavo y a Jorge, mis directores, por todo lo que me enseñaron, por la oportunidad de trabajar con ellos y por la ayuda en el camino de llegar a ser doctor. Ya sea por charlas espontáneas o por el tiempo que me dedicaron, mis trabajos se beneficiaron de los miembros de HPCN. En particular, agradezco a Sergio, por su crítica constructiva y honesta. Durante estos años en el laboratorio, tuve grandes compañeros con quienes intercambié ideas, frustraciones, problemas, y soluciones. Por esto quiero agradecer a Angel, Dani, David, Edu, Guille, Jesus, Jose Fernando, Rafa y Sergio. Adicionalmente, agradezco a aquellos profesores de la Facultad de Ingeniería de la UNSJ que me dieron herramientas para enfrentar este desafío.

Last, but not least, I want to thank the Xilinx Research Labs team in Ireland for the great experience that I had working with them. In particular, thanks to Jakoba, Yaman, Guilio, Alina, Ela, Nick, and Michaela, for the help and jokes that made the job easy. Specially, I would like to thank Lucian, who guided me in my work. Thanks for all that he taught me, and for both the pragmatic and the idealistic discussions we had.

List of contents

List of Tables

List of Figures

List of Algorithms

Part I

Introduction & Background

CHAPTER 1

INTRODUCTION

In this chapter, we detail the motivations and context of this thesis. This establishes the challenges we face and, therefore, the starting point upon which we set the objectives, and then, determine the methodology to achieve them. Finally, we outline the structure of the rest of the thesis.

1.1 Context & Motivation of the Thesis

1.1.1 Trends of Data Processing Requirements

In many application areas, it is possible to see a continuous push towards higher performance systems in terms of bandwidth, latency, functionality, energy efficiency, among others. In the image and video processing area, the amount of data grows steadily as sensors and solutions target higher resolutions, number of colors, and frame rates. Other applications prioritize the improvement of different metrics, such as in the transmission of first-person view video for drone telecommand, where very low latencies have to be achieved. A generalization of this case are closed-loop control systems with video feedback, where increased latencies in the loop decrease the system's stability.

In computer networks, a similar trend is present. Connection bandwidth offered by Internet service providers has been growing at an exponential rate (approx. 50% per

year) over the last decades, which is known as the Nielsen's law of Internet bandwidth [1]. Greater data rates bring many challenges, not only for the information delivery systems, but also for the mechanisms to analyze in real-time the health of this infrastructure.

Another example is found in the artificial intelligence area, where the great success of their algorithms, and, in particular, of artificial neural networks, in addressing problems that a few years ago were considered to be, in practice, out of reach for computers, impulsed a considerable number of applications that make use of this technology [2]. These range from high-throughput datacenter accelerated inference applications to critical embedded systems like advanced driver assistance systems (ADAS) and autonomous vehicles (AV). These applications, tend not only to consume big amounts of information, but also are very compute intensive.

1.1.2 Trends of General Purpose Hardware Performance

Contrasting with this increase in the data processing needs, the capacity of the leading technology (silicon semiconductors) to keep pushing performance forward is decreasing with each successive generation. Fig. 1.1 shows the performance evolution of general purpose processors over the last decades, measured by the SPECint benchmark and power. Until about 2004, new process nodes not only allowed doubling the transistor

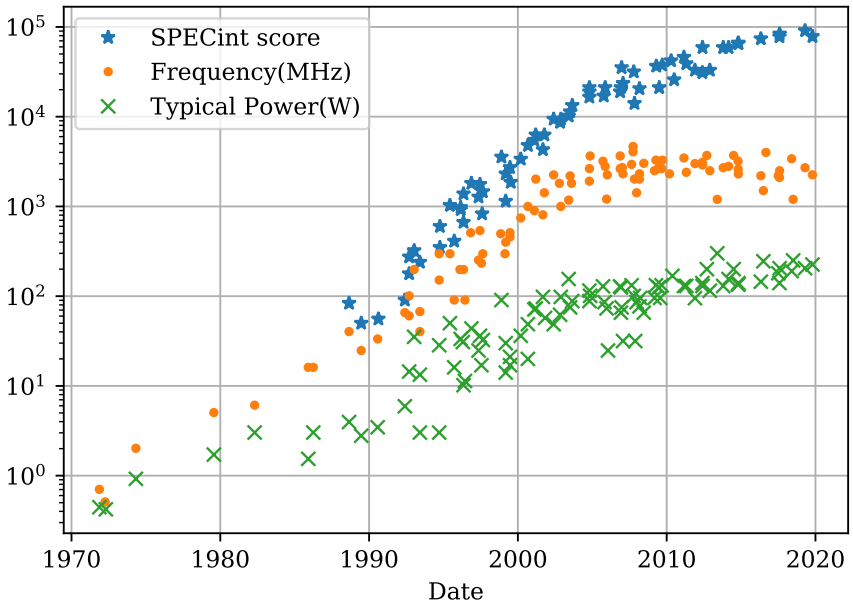


Figure 1.1: Processor performance evolution. Data source: [3]

count per chip approx. every two years (Moore's Law [4]), but also increasing operation frequency while keeping power per unit area constant, which was accomplished using what is known as Dennard scaling [5]. This explains most of the performance increase shown in the figure until that year. After this, although higher transistor densities could be achieved, frequencies plateau and power started limiting performance.

For this reason, chip companies moved towards multicore architectures to keep increasing the work performed per unit of time. However, for a fix size problem, the speed-up experienced using parallelization faces diminishing returns, which is a function of the sequential portion of the procedures, as dictated by Amdahl's law [6]. Gustafson argued that as more compute power was available, the sizes of the problems we would try to tackle would increase, and with that, their parallel portion, resulting in an alternative formula for the speed-up accomplished by multicore systems, Gustafson's law [7]. Although this latter law is applicable to many problems, latency improvements are still governed by Amdahl's law, and power, chip area, among other constraints often limit the number of processing units that can be added. What is more, the increasing percentage of dark silicon (underutilized transistors) each new process node brings (caused by power dissipation issues derived from the end of Dennard scaling) does not allow multiplying the number of running cores (at peak performance) by the same factor the number of transistors grows [8]. In addition, other factors, like the memory system structure, also limits performance of general purpose hardware.

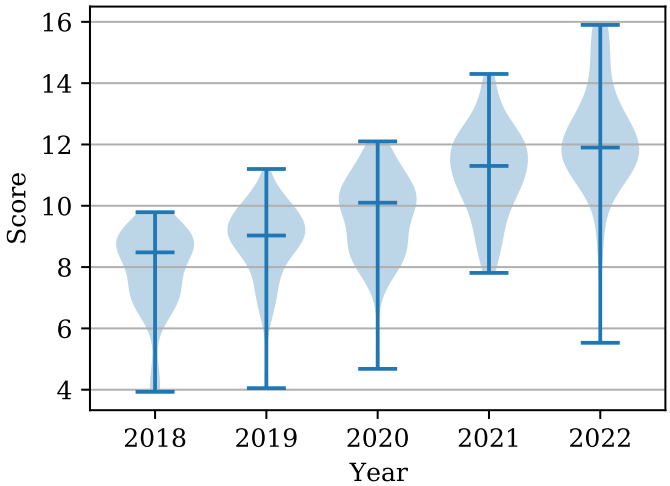


Figure 1.2: Violin plots showing SPEC2017int speed scores for the first quartile of each year. Minimum, median, and maximum values are marked. Data source: [9]

To observe in more detail what the trend has been over the last few years, we can analyze the published results of the SPEC2017int speed benchmark [9]. Fig. 1.2 shows the distribution of these results for the first quarter of each year. Although they correspond to high-end datacenter servers, the inter-annual performance rates, taking the maximum score as a reference, were 14.4%, 8.0%, 18.2%, and 11.2%, resulting in an average 12.9% increase per year (doubling performance every 5.7 years), which is a modest growth compared to what we have enjoyed in the previous decades. The median results have been growing slower at an average 8.2% inter-annual rate.

1.1.3 Trends of Custom Designed Hardware and FPGAs

Considering these trends, at the present, it is increasingly necessary to resort to other means to improve performance. Leiserson *et al.* [10], foreseeing a potential stall in the miniaturization process for both physical and economic reasons, identified three areas that can push computer capacities forward: software performance engineering, algorithms, and hardware architecture. In particular, for the last two, they stress the importance of algorithm-hardware co-design and hardware specialization, which is inline with other works [11, 12].

Custom hardware has been a vital tool in many application areas, given that it can exceed the performance of other technologies, leveraging custom memory hierarchies, avoiding instruction fetch and decode logic, using custom operators, improving data reuse, among others. Also, nowadays, this alternative has become more profitable, given that general purpose hardware would take longer than in the past to reach the same level of performance. Yet, Application-Specific Circuit Technologies (ASICs) have high non-recurring engineering cost (which are increasing), then these are normally only justified for high-volume production. Conversely, these costs are low for field programmable gate arrays (FPGA), thus enabling the use of custom-designed hardware in lower-volume applications. At the same time, FPGAs provide a greater flexibility, given the possibility of updating the implemented hardware, which, in turn, lowers development costs further as errors are easier to fix.

Nevertheless, FPGA have to compete against other specialized technologies like digital signal processors (DSP), graphic processing units (GPU) and application specific standard products (ASSP). Given their ease of use, flexibility and price, CPUs are, normally, the default target device. If system constraints cannot be met with them, for the same reasons, DSPs, GPUs and ASSPs tend to be the next devices to be considered. FPGAs are generally taken into account only if the previous options are not suitable for

the task, as they are harder to program, so they have longer time-to-market and higher development costs. Because of this, the FPGA engineer normally confronts problems with stringent constraints in terms of latency, throughput, or power. Therefore, and considering that FPGAs achieve slower clock frequencies, efficient implementations are required, not only because of the hard nature of the problems they normally face, but also to fit the solution in smaller, cheaper chips to better compete in prices.

As a response to this situation, and to enable designers to develop larger systems, FPGA vendors, electronic design automation (EDA) companies and the academia are pushing towards easier FPGA programmability, and, in general, faster digital hardware design. For this, it is sought to move away from register transfer level (RTL) languages, like Verilog and VHDL, which require a more detailed specification and in-depth understanding of hardware and its design. As an alternative, researchers have worked in increasing the abstraction level, among other means, developing compilers to generate RTL from untimed languages, like C, C++ or Scala [13]. Although many of these tools have matured for more than a decade, their use still tends to imply a significant trade-off between quality of results (QoR) and development time [14], which, as previously indicated, the FPGA designer cannot often afford.

1.2 Objectives & Methodology of the Thesis

1.2.1 Objectives

In this context, we consider fundamental to increase the hardware designer productivity, understood as the achieved performance per unit of developer time. In other words, our main objective is to reduce the time the designer needs to achieve a satisfactory system. To reach this objective, we take a holistic approach, targeting all the hardware design stages. That is, we aim to:

- **Automate design tasks:** Task automation is key to enable greater productivity, not only because the designer is freed from the task, at least partially, but also because computers are generally better at optimizing large high-dimensional spaces, so both development time and QoR can be improved simultaneously. Automating is particularly impactful when it is applied to error-prone processes, and also can enable less experienced designers to obtain expert-level QoR.
- **Obtain better architectures for common algorithm patterns:** For a given problem, architectures constitute different points in the design space, that is,

different trade-offs between latency, throughput, area, power, determinism, among others. For HLS to increase productivity in the sense stated above, it needs to be able to achieve not just a functionally correct system, but one that satisfies the often stringent constraints of FPGA applications. It is therefore important for HLS tools to have at their disposal a set of Pareto optimal architectures to implement different algorithm patterns.

- **Obtain better algorithms for hardware implementation:** Algorithms establish fundamental limitations to what can be achieved, and yet most algorithms were designed targeting the random-access machine [15]. As previously noted, algorithm-hardware co-design can be a key enabler of greater performance. For example, QuickSort [16] is a widely used and efficient sorting algorithm, but Bitonic sorting [17] may be preferred in GPU or FPGA implementations, despite being slower if run sequentially. This is because of its lower latency when run in parallel.
- **Improve the hardware design methodology using high-level synthesis:** misusing a good tool will lead us to poor results. Good methodologies allow us to get better results and in a reduced amount of time. It is therefore important to understand how to better use HLS and how to incorporate it in the hardware design process.

1.2.2 Methodology

To achieve the objectives, we explore difficult problems in relevant areas, studying the systems requirements and identifying the design challenges. Although these may emerge from specific problems, we aim to provide solutions to a wider range of applications.

The selected areas and problems are the following:

- **Network packet processing:** In this area, we address the challenge of network flow metering offloading of 100 Gigabit Ethernet (GbE) links. Extensive work has been done to offload stateless network packet processing like filtering and routing, and these applications have even influenced how FPGA fabric is designed [18]. Yet, when it is necessary to keep a state, dependencies are established, which, added to the required high throughput that 100 GbE demands, constitutes a hard problem to face. Dependencies are common in many application areas, then strategies to ease them in one area, may be applied in others.
- **Image processing:** In this area, we address the problem of image lossless and near-lossless compression in highly constraint scenarios. Given the large amount

of data sensors are capable of producing, compression is often mandatory not only because of throughput limitations but also for power limitations, as data transmission is often the main contributor and harder to optimize process in terms of power [11]. On top of this, many applications also require low latency. Developments for this application may also be applied to other data compression algorithms, easing the power consumption problems faced from the embedded systems to datacenters.

- **Artificial intelligence:** In this area, we aim to implement large Convolutional Neural Networks (CNN) for datacenter application acceleration. This application is compute-intensive and highly parallelizable, thus suitable for FPGA acceleration. Although GPUs are also very suitable for CNN acceleration, CNN implemented as dataflows (all layers running concurrently in a large pipeline) in FPGAs allow larger accelerators to simultaneously increase throughput and decrease latency. Thus, these FPGA scaled up implementations may allow pushing the Pareto Frontier forward. However, the main challenge here is to face the problems that arise when these accelerators require spanning over the whole FPGA (specially for non-monolithic FPGAs) and even several FPGAs. Developments for this application may also be applied to other large-scale systems, which we will almost certainly face as data processing needs and the FPGA designer productivity continue to rise.

Notice that these applications cover a wide range of target devices, from small and generally cost and power optimized ones for image compression, passing through high-end medium/large FPGAs for 100 GbE flow metering, and finally, to large FPGA acceleration cards or clusters of them for CNN acceleration. For each of these problems, we first analyze the suitability of using higher level languages and tools to address them. If this leads to a successful development, we aim to identify those methodological components considered to be key contributors to the design success. When the higher-level approach is deemed insufficient, we aim to understand why the tools do not address the problem efficiently, and to propose improvements to these tools or complementary ones. In some cases, we may observe opportunities to enhance available algorithms for their implementation in hardware, in those cases, we will also aim at improving them. Finally, through this process, solutions to the specific problems were be developed, which are also be considered contributions of this thesis.

Fig. 1.3 serves as a summary, illustrating the relationship among the different components of this thesis.

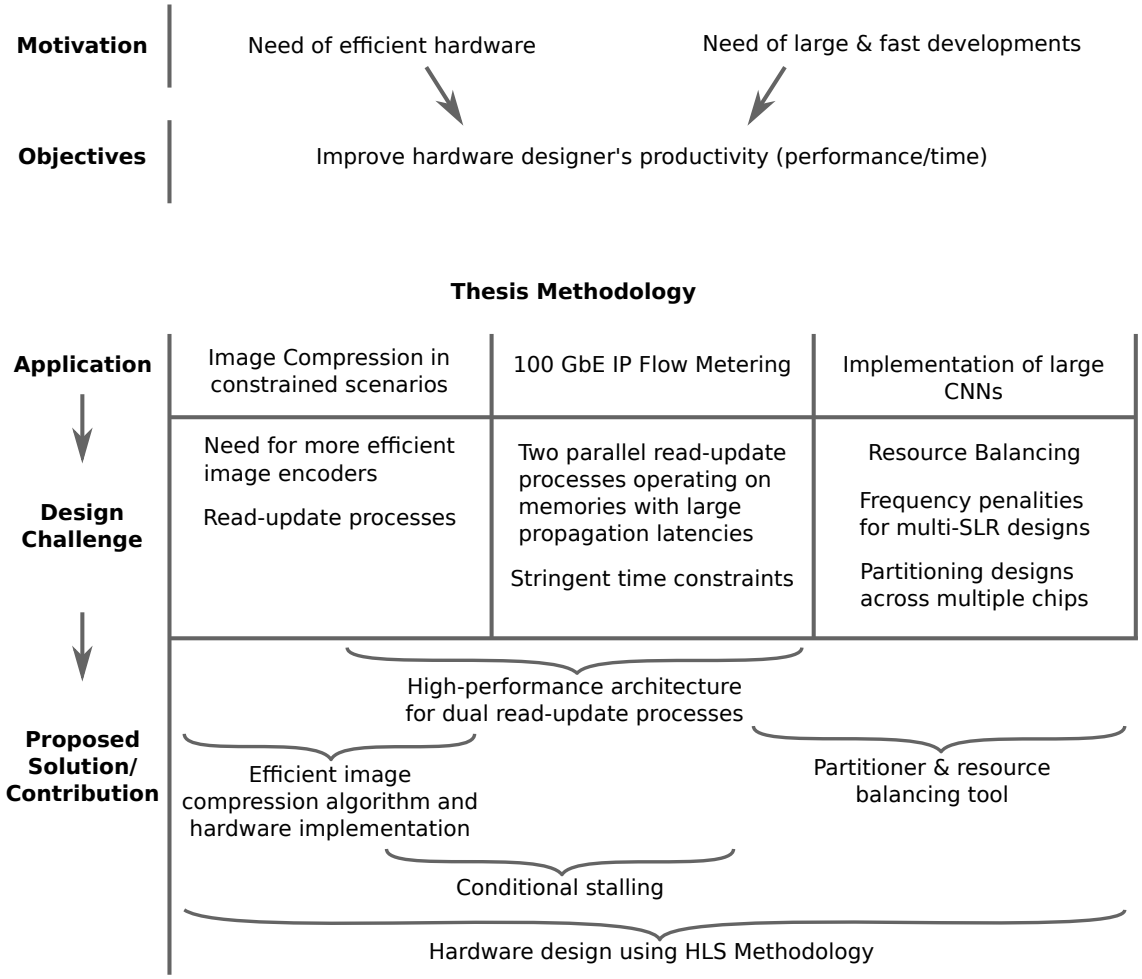


Figure 1.3: Thesis methodology

1.3 Structure of the Thesis

This work is structured in four parts. The first one, *Introduction & Background*, includes this chapter and an additional one providing general background about FPGA design and high-level synthesis. Then, *Case Studies and Challenges Presentation* part is constituted by three chapters (one per application area), each of which introduces the application, providing specific background and analyses of the problems at hand, concluding with the determination of the specific research and development goals. After that, *Addressing the Challenges* part, includes a series of chapters, each of which provides solutions to the challenges identified in the previous part. Finally, the last *Putting All Together* part, presents key methodological aspects that led us to a successful hardware implementation using high-level synthesis and provides a summary of the contributions of this thesis. A brief abstract can be found at the beginning of each chapter.

CHAPTER 2

INTRODUCCIÓN

En este capítulo detallamos las motivaciones y el contexto de esta tesis. En él se establecen los retos a los que nos enfrentamos y, por tanto, el punto de partida sobre el cual fijamos los objetivos y, a continuación, determinamos la metodología para alcanzarlos. Por último, esbozamos la estructura del resto de la tesis.

2.1 Contexto y Motivación de la Tesis

2.1.1 Tendencias de los Requerimientos de Procesamiento

En muchas áreas de aplicación, se observa un avance continuo hacia sistemas de mayor rendimiento en términos de ancho de banda, latencia, funcionalidad y eficiencia energética, entre otros. En el área de procesamiento de imágenes y vídeo, la cantidad de datos crece constantemente a medida que los sensores y las soluciones se orientan hacia mayores resoluciones, número de colores y velocidad de fotogramas. Otras aplicaciones priorizan la mejora de otras métricas, como en la transmisión de vídeo en primera persona para el telecontrol de drones, donde hay que conseguir latencias muy bajas. Una generalización de este caso son los sistemas de control de bucle cerrado con retroalimentación de vídeo, donde el aumento de las latencias en el bucle disminuye la

estabilidad del sistema.

En las redes de ordenadores se da una tendencia similar. El ancho de banda ofrecido por los proveedores de Internet ha crecido a un ritmo exponencial (aprox. 50% al año) durante las últimas décadas, lo que se conoce como la ley de Nielsen del ancho de banda de Internet [1]. Mayores tasas de datos conllevan muchos desafíos, no solo para los sistemas de entrega de información, sino también para los mecanismos de análisis en tiempo real de la salud de esta infraestructura.

Otro ejemplo lo encontramos en el área de la inteligencia artificial, donde el gran éxito de sus algoritmos y, en particular, de las redes neuronales artificiales, para abordar problemas que hace unos años se consideraban, en la práctica, fuera del alcance de los ordenadores, impulsó un número considerable de aplicaciones que hacen uso de esta tecnología [2]. Estas van desde aplicaciones de inferencia acelerada en centros de datos de alto rendimiento hasta sistemas integrados críticos como los sistemas avanzados de asistencia al conductor y los vehículos autónomos. Estas aplicaciones, además de consumir grandes cantidades de información, suelen ser muy intensivas en computación.

2.1.2 Tendencias del Desempeño del Hardware de Propósito General

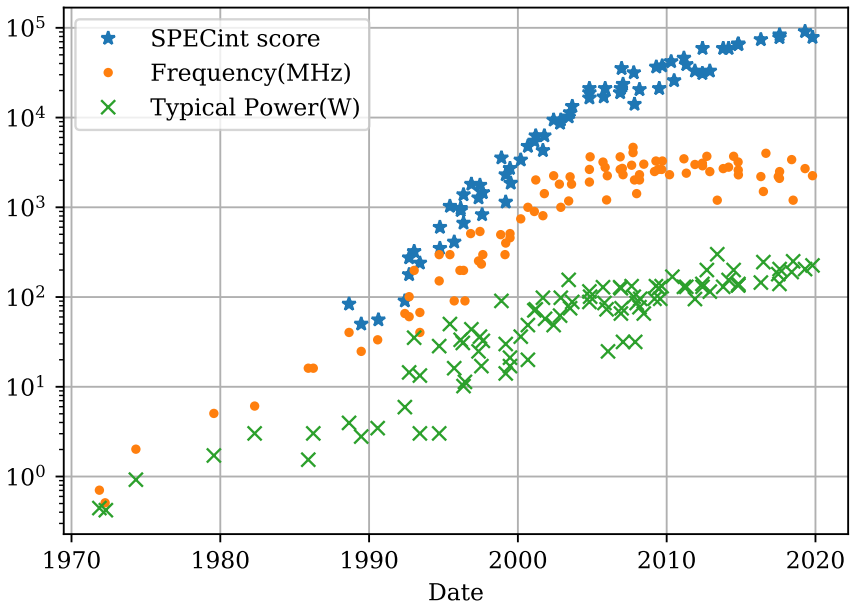


Figure 2.1: Evolución del desempeño de los procesadores. Fuente de los datos: [3]

En contraste con este aumento de las necesidades de procesamiento de datos, la capacidad de la tecnología de punta (semiconductores de silicio) para seguir impulsando el rendimiento disminuye con cada generación sucesiva. La Fig. 2.1 muestra la evolución del rendimiento de los procesadores de propósito general en las últimas décadas, medido por el *benchmark* SPECint y la potencia consumida. Hasta aproximadamente 2004, los nuevos nodos de proceso no solo permitían duplicar el número de transistores por chip cada dos años aproximadamente (Ley de Moore [4]), sino también aumentar la frecuencia de funcionamiento manteniendo constante la disipación de energía por unidad de superficie, lo que se conseguía utilizando lo que se conoce como escalado de Dennard [5]. Esto explica la mayor parte del aumento de rendimiento mostrado en la figura hasta ese año. Después de esto, aunque se pudieron lograr mayores densidades de transistores, las frecuencias se estancaron y la disipación de energía comenzó a limitar el rendimiento.

Por esta razón, las empresas de chips se han movido hacia arquitecturas de múltiples núcleos para seguir aumentando el trabajo realizado por unidad de tiempo. Sin embargo, para un problema de tamaño fijo, el aumento de velocidad experimentado mediante la paralelización se enfrenta a rendimientos decrecientes, lo que es una función de la parte secuencial de los procedimientos, según dicta la ley de Amdahl [6]. Gustafson argumentó que, a medida que se dispusiera de más potencia de cálculo, aumentaría el tamaño de los problemas que intentáramos abordar y, con ello, su parte paralela, lo que dio lugar a una fórmula alternativa para el aumento de velocidad logrado por los sistemas multi-núcleo, la ley de Gustafson [7]. Aunque esta última ley es aplicable a muchos problemas, las mejoras de latencia siguen rigiéndose por la ley de Amdahl, y la potencia, el área del chip, entre otras restricciones, suelen limitar el número de unidades de procesamiento que pueden añadirse. Es más, el creciente porcentaje de silicio oscuro (transistores infrautilizados) que trae consigo cada nuevo nodo de proceso (provocado por los problemas de disipación de energía derivados del fin del escalado de Dennard) no permite multiplicar el número de núcleos en funcionamiento (a máximo rendimiento) por el mismo factor por el que crece el número de transistores [8]. Además, otros factores, como la estructura del sistema de memoria, también limitan el rendimiento del hardware de propósito general.

Para observar con más detalle cuál ha sido la tendencia en los últimos años, podemos analizar los resultados publicados del *benchmark* de velocidad SPEC2017int [9]. La Fig. 2.2 muestra la distribución de estos resultados para el primer trimestre de cada año. Aunque corresponden a servidores de centros de datos de gama alta, las tasas de rendimiento interanuales, tomando como referencia la puntuación máxima, han sido

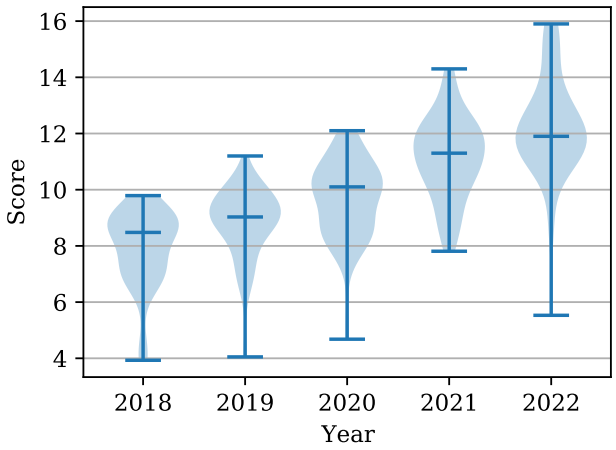


Figure 2.2: Gráficos de violín que muestran las puntuaciones de velocidad de SPEC2017int para el primer cuartil de cada año. Se han marcado los valores mínimo, mediano y máximo. Fuente de los datos: [9]

del 14,4%, 8,0%, 18,2% y 11,2%, lo que supone un incremento medio del 12,9% por año (duplicando el rendimiento cada 5,7 años), lo que supone un crecimiento modesto comparado con el que hemos disfrutado en las décadas anteriores. La mediana de los resultados ha crecido más lentamente, a un ritmo medio del 8,2% interanual.

2.1.3 Tendencias del Hardware Diseñado a Medida y las FPGA

Teniendo en cuenta estas tendencias, en la actualidad es cada vez más necesario recurrir a otros medios para aumentar el rendimiento. Leiserson *et al.* [10], previendo un posible estancamiento en el proceso de miniaturización por razones tanto físicas como económicas, identificaron tres áreas que pueden impulsar la capacidad de los ordenadores: ingeniería de desempeño de software, algoritmos y arquitecturas de hardware. En particular, para las dos últimas, destacan la importancia del co-diseño algoritmo-hardware y la especialización del hardware, lo que coincide con otros trabajos [11, 12].

El hardware a medida ha sido una herramienta vital en muchas áreas de aplicación, dado que puede superar el rendimiento de otras tecnologías, aprovechando jerarquías de memoria adaptadas, evitando la lógica de lectura y decodificación de instrucciones, utilizando operadores a medida, mejorando la reutilización de datos, entre otros. Además, hoy en día, esta alternativa es más rentable, dado que el hardware de propósito general tardaría más que en el pasado en alcanzar el mismo nivel de rendimiento. Sin embargo, las tecnologías de circuitos específicos para aplicaciones (ASIC, por sus siglas en inglés)

tienen un alto coste de ingeniería no recurrente (que va en aumento), por lo que normalmente solo se justifican para la producción de grandes volúmenes. Por el contrario, estos costes son bajos en el caso de las matrices de puertas lógicas programables en campo (FPGA, por sus siglas en inglés), lo que permite el uso de hardware diseñado a medida en aplicaciones de menor volumen. Al mismo tiempo, las FPGA proporcionan una mayor flexibilidad, dada la posibilidad de actualizar el hardware implementado, lo que, a su vez, reduce aún más los costes de desarrollo, ya que los errores son más fáciles de corregir.

No obstante, las FPGA tienen que competir con otras tecnologías especializadas, como los procesadores de señales digitales (DSP, por sus siglas en inglés), las unidades de procesamiento gráfico (GPU, por sus siglas en inglés) y los productos estándar para aplicaciones específicas (ASSP, por sus siglas en inglés). Dada su facilidad de uso, flexibilidad y precio, las CPU son, normalmente, el dispositivo objetivo por defecto. Si las restricciones del sistema no pueden cumplirse con ellas, por las mismas razones, los DSP, las GPU y los ASSP suelen ser los siguientes dispositivos a tener en cuenta. Las FPGA suelen tenerse en cuenta solo si las opciones anteriores no son adecuadas para la tarea, ya que son más difíciles de programar, por lo que tienen un tiempo de comercialización más largo y unos costes de desarrollo más elevados. Por ello, el ingeniero de la FPGA suele enfrentarse a problemas con estrictas restricciones en términos de latencia, rendimiento o potencia. Por lo tanto, y teniendo en cuenta que las FPGAs alcanzan frecuencias de reloj más lentas, se requieren implementaciones eficientes, no solo por la dificultad de los problemas a los que normalmente se enfrentan, sino también para ser capaz de utilizar chips más pequeños y baratos para competir mejor en precios.

Como respuesta a esta situación, y para que los diseñadores puedan desarrollar sistemas más grandes, los proveedores de FPGA, las empresas de automatización del diseño electrónico (EDA, por sus siglas en inglés) y el mundo académico están impulsando una programabilidad más sencilla de las FPGA, y en general, del diseño de hardware digital. Para eso se busca alejarse de los lenguajes de nivel de transferencia de registros (RTL, por sus siglas en inglés), como Verilog y VHDL, que requieren una especificación más detallada y un conocimiento más profundo del hardware y su diseño. Como alternativa, los investigadores han trabajado en aumentar el nivel de abstracción, entre otros medios, desarrollando compiladores para generar RTL a partir de lenguajes no temporizados, como C, C++ o Scala [13]. Aunque muchas de estas herramientas han madurado desde hace más de una década, su uso todavía tiende a implicar un importante intercambio entre calidad de resultados (QoR, por sus siglas en inglés) y tiempo de desarrollo [14], que, como se ha indicado anteriormente, el diseñador de FPGAs a menudo no puede

permitirse.

2.2 Objetivos & Metodología de la Tesis

2.2.1 Objetivos

En este contexto, consideramos fundamental aumentar la productividad del diseñador de hardware, entendida como el rendimiento alcanzado por unidad de tiempo del desarrollador. En otras palabras, nuestro principal objetivo es reducir el tiempo que el diseñador necesita para conseguir un sistema satisfactorio. Para alcanzar este objetivo, adoptamos un enfoque holístico, dirigido a todas las etapas del diseño de hardware. Es decir, buscamos:

- **Automatiza tareas de diseño:** La automatización de tareas es clave para permitir una mayor productividad, no solo porque el diseñador se libera de la tarea, al menos parcialmente, sino también porque los ordenadores suelen ser mejores optimizadores de grandes espacios y de elevadas dimensiones, por lo que tanto el tiempo de desarrollo como la calidad de los resultados pueden mejorarse simultáneamente. La automatización es especialmente útil cuando se aplica a procesos propensos a errores, y también puede permitir a los diseñadores menos experimentados obtener resultados de calidad a nivel de experto.
- **Obtener mejores arquitecturas para patrones de algoritmos comunes:** Para un problema determinado, las arquitecturas constituyen diferentes puntos en el espacio de diseño, es decir, diferentes compromisos entre latencia, rendimiento, área, potencia, determinismo, entre otros. Para que HLS aumente la productividad en el sentido indicado, tiene que ser capaz de conseguir no solo un sistema funcionalmente correcto, sino uno que satisfaga las restricciones, a menudo estrictas, de las aplicaciones de las FPGA. Por lo tanto, es importante que las herramientas HLS tengan a su disposición un conjunto de arquitecturas pareto-óptimas para implementar diferentes patrones de algoritmos.
- **Obtener mejores algoritmos para la implementación en hardware:** Los algoritmos establecen limitaciones fundamentales a lo que se puede conseguir, y, sin embargo, la mayoría de los algoritmos se diseñaron pensando en la máquina de acceso aleatorio [15]. Como se ha señalado anteriormente, el co-diseño de algoritmos y hardware puede ser un factor clave para lograr un mayor rendimiento. Por

ejemplo, QuickSort [16] es un algoritmo de ordenación muy utilizado y eficiente, pero la ordenación bitónica [17] puede ser preferible en implementaciones de GPU o FPGA, a pesar de ser más lenta si se ejecuta secuencialmente. Esto es debido a su menor latencia cuando se ejecuta en paralelo.

- **Mejorar la metodología de diseño de hardware utilizando la síntesis de alto nivel:** un mal uso de una buena herramienta nos llevará a resultados pobres. Buenas metodologías nos permiten obtener mejores resultados y en un tiempo reducido. Por lo tanto, es importante entender cómo utilizar mejor HLS y cómo incorporarlo en el proceso de diseño de hardware.

2.2.2 Metodología

Para lograr los objetivos, exploramos problemas difíciles en áreas relevantes, estudiando los requisitos de los sistemas e identificando los retos de diseño. Aunque estos surjan de problemas específicos, buscamos aportar soluciones a una gama más amplia de aplicaciones.

Las áreas y problemas seleccionados son los siguientes:

- **Procesamiento de paquetes de red:** En esta área, abordamos el reto de la descarga de la medición de flujos de red de los enlaces de 100 Gigabit Ethernet (GbE). Se ha realizado un amplio trabajo para descargar el procesamiento de paquetes de red sin estado, como el filtrado y el enrutamiento, y estas aplicaciones han influido incluso en la forma en que se diseña el tejido FPGA [18]. Sin embargo, cuando es necesario mantener un estado, se establecen dependencias que, sumadas al alto rendimiento requerido que exige 100 GbE, constituyen un problema difícil de afrontar. Las dependencias son comunes en muchas áreas de aplicación, luego las estrategias para aliviarlas en un área, pueden aplicarse en otras.
- **Procesamiento de imágenes:** En esta área, abordamos el problema de la compresión de imágenes sin pérdidas y casi sin pérdidas en escenarios con estrictas restricciones. Dada la gran cantidad de datos que los sensores son capaces de producir, la compresión es a menudo obligatoria no solo por las limitaciones de ancho de banda, sino también por las de potencia, ya que la transmisión de datos es a menudo el principal contribuyente y el proceso más difícil de optimizar en términos de consumo de potencia [11]. Además, muchas aplicaciones también requieren una baja latencia. Los desarrollos para esta aplicación también pueden aplicarse a otros

algoritmos de compresión de datos, aliviando los problemas de consumo de energía a los que se enfrentan desde los sistemas embebidos hasta los centros de datos.

- **Inteligencia artificial:** En esta área, nuestro objetivo es implementar grandes redes neuronales convolucionales (CNN, por sus siglas en inglés). Esta aplicación requiere una gran cantidad de cálculos y es altamente paralelizable, por lo que es adecuada para la aceleración mediante FPGA. Aunque las GPU también son muy adecuadas para la aceleración de CNN, las CNN implementadas como flujos de datos (todas las capas se ejecutan simultáneamente usando circuitos segmentados con una gran cantidad de etapas) en FPGA permiten que los aceleradores más grandes aumenten el rendimiento y reduzcan la latencia simultáneamente. Por lo tanto, estas implementaciones escaladas en FPGAs pueden permitir que se empuje la frontera de Pareto. Sin embargo, el principal reto aquí es hacer frente a los problemas que surgen cuando estos aceleradores requieren abarcar toda la FPGA (especialmente en el caso de las FPGAs no monolíticas) e incluso utilizar varias FPGAs. Los desarrollos para esta aplicación también pueden aplicarse a otros sistemas a gran escala, a los que nos enfrentaremos casi con toda seguridad a medida que las necesidades de procesamiento de datos y la productividad de los diseñadores de FPGA sigan aumentando.

Nótese que estas aplicaciones cubren una amplia gama de dispositivos objetivo, desde los pequeños y generalmente optimizados en coste y potencia para la compresión de imágenes, pasando por FPGAs de gama media/grande para la medición de flujo de 100 GbE, y finalmente, hasta grandes tarjetas de aceleración FPGA o conjunto de ellas para la aceleración de CNN. Para cada uno de estos problemas, analizamos primero la idoneidad de utilizar lenguajes y herramientas de alto nivel para abordarlos. Si esto conduce a un desarrollo exitoso, pretendemos identificar aquellos componentes metodológicos que se consideran clave para el éxito del diseño. Cuando el enfoque de mayor nivel de abstracción se considera insuficiente, pretendemos entender por qué las herramientas no abordan el problema de forma eficiente, y proponer mejoras a las mismas u otras complementarias. En algunos casos, podemos observar oportunidades para mejorar los algoritmos disponibles para su implementación en hardware, en esos casos, también nos propondremos mejorarlos. Finalmente, a través de este proceso, se desarrollarán soluciones a los problemas concretos, que también se considerarán contribuciones de esta tesis.

La Fig. 2.3 sirve de resumen, ilustrando la relación entre los diferentes componentes de esta tesis.

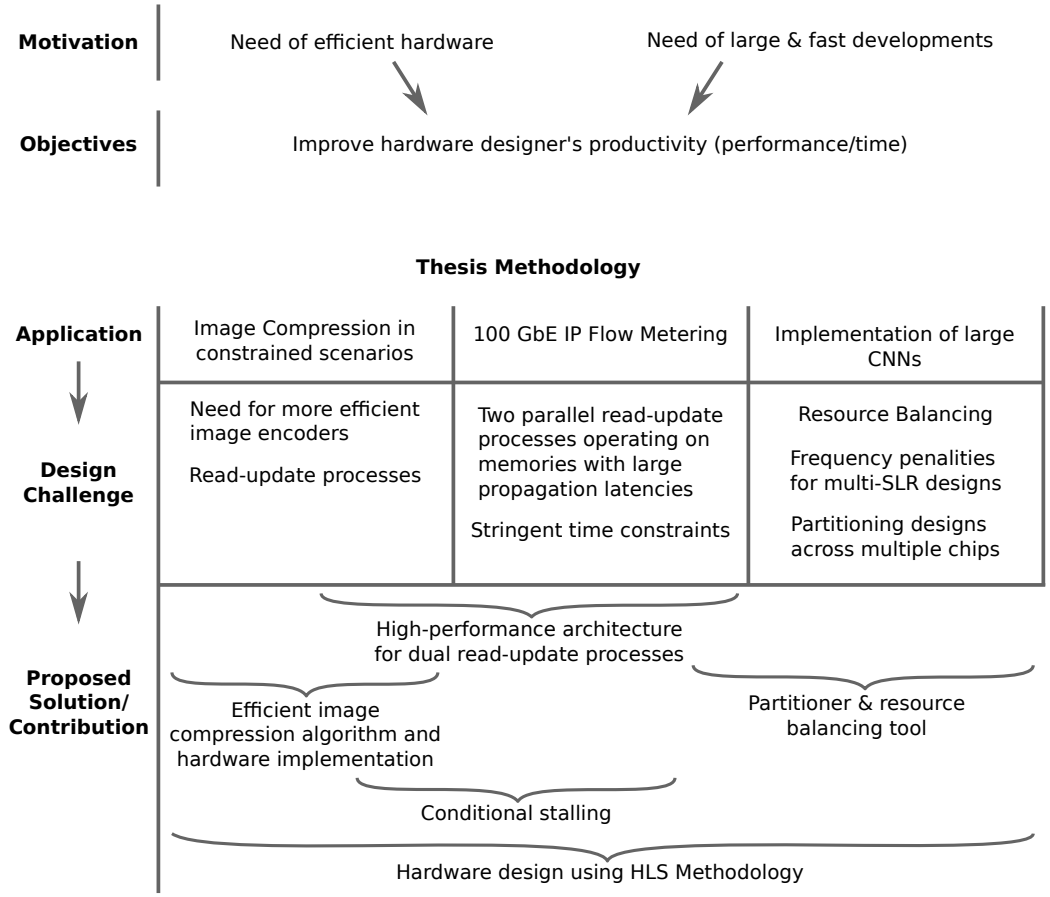


Figure 2.3: Metodología de la tesis

2.3 Estructura de la Tesis

Este trabajo está estructurado en cuatro partes. La primera, *Introduction & Background*, incluye este capítulo y uno adicional que proporciona antecedentes generales sobre el diseño FPGA y la síntesis de alto nivel. A continuación, la parte *Case Studies and Challenges Presentation* está constituida por tres capítulos (uno por área de aplicación), cada uno de los cuales introduce la aplicación, proporcionando antecedentes específicos y análisis de los problemas en cuestión, concluyendo con la determinación de los objetivos específicos de investigación y desarrollo. A continuación, la parte *Addressing the Challenges*, incluye una serie de capítulos, cada uno de los cuales aporta soluciones a los desafíos identificados en la parte anterior. Finalmente, la última parte *Putting All Together*, presenta los aspectos metodológicos clave que nos han llevado a una implementación hardware exitosa utilizando síntesis de alto nivel y proporciona un resumen de las contribuciones de esta tesis. Un resumen breve se encuentra al comienzo de cada capítulo.

CHAPTER 3

BACKGROUND

In this chapter, general background is provided, defining important terms and systems related to FPGA design, from electronic circuits to languages to digital hardware. Application-specific background is introduced in part II of the thesis.

3.1 Electronic Circuit Technology

Electronic Circuits Given its dimensions, compute power, energy efficiency, and robustness, our main data processing technology is *electronic circuits* based on silicon semiconductors. Electronic circuits are systems (machines) that manipulate electric signals for a given purpose. When these signals vary continuously (represented by a real number), we say the circuit is *analog*. For example, we can transform a sound wave into an electric signal using a microphone, amplify it, and finally, convert it back to sound with a speaker. In reality, all circuits are technically analog, but we use the term *digital circuits* for those that can be modeled (think about them) as systems that can be only in a finite set of states.

Finite-State Machines Within digital circuits, there is a subset that uses a special signal, a clock, that tell the systems when it may change its state. These are *synchronous digital systems*. Normally, we describe them as *finite-state machines* (FSM), whose

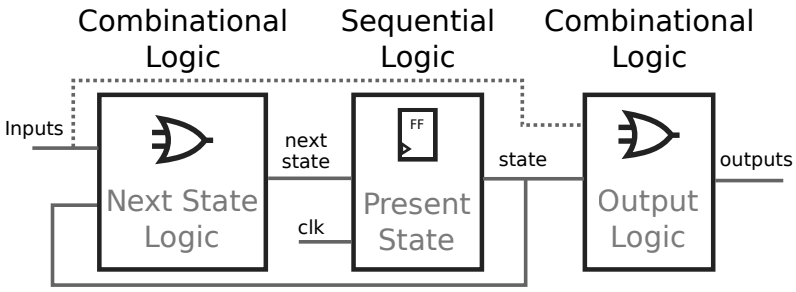
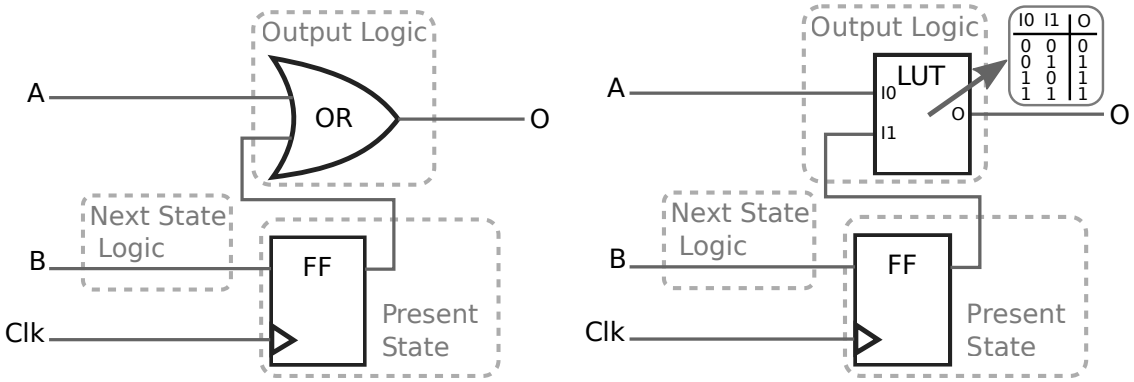


Figure 3.1: Finite state machine structure

structure is presented in fig.3.1. As observed, they have *sequential logic*, which store the state of the system, and *combinational logic*, that manipulates digital signals to determine the next state (state assumed by the sequential logic after the clock ticks again) and the outputs of the system.

Logic Sequential logic is normally composed by *flip-flips* (FF), circuits that store a 0 (low voltage) or a 1 (high voltage) and that only change this state when the clock ticks, and thus are a form of memory. Combinational logic are circuits that react instantly to their inputs, changing their output accordingly after a *propagation delay* (the time it takes for the signal to traverse the circuit). Combinational logic can be constructed in many ways, for example, fig. 3.2 shows two equivalent circuits, one using *logic gates* and the other using *look-up tables* (LUTs). Logic gates are circuits that are designed to implement a particular binary function. For example, in fig. 3.2(a) we observe an *OR gate*, which outputs a 1 if any of its inputs is 1, else it outputs a 0. LUTs are memories that are configured to return the result of a given function when addressed by the input variables of that function. For example, in fig. 3.2(b), the LUT implements an OR function.



((a)) Simple FSM using gate

((b)) Simple FSM using LUT

Figure 3.2: Two equivalent implementations of a simple FSM

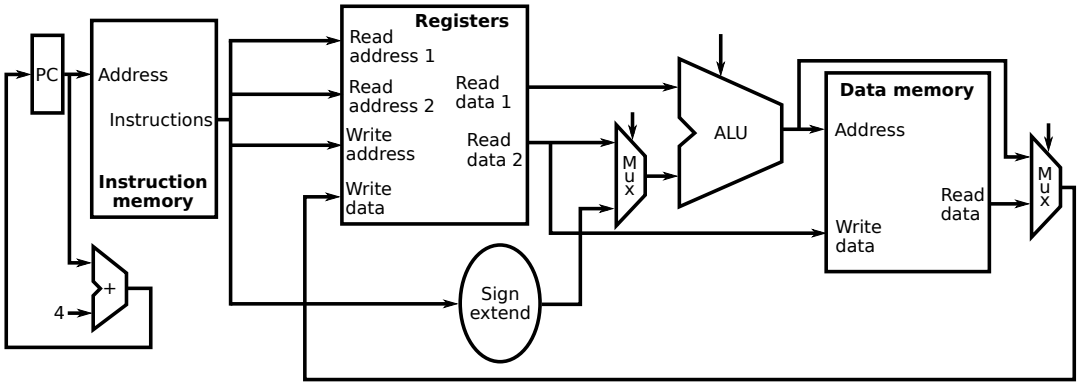


Figure 3.3: Block diagram of a simple processor

Microprocessors With these resources, we can build complex systems. We can group FFs to create *registers*(one-dimensional array of memory cells) or memories (two-dimensional array of memory cells). Also, we can design combinational logic to do compute numerous arithmetic operations (an *arithmetic logic unit* (ALU)). In this way, combining simpler systems to great bigger ones, we can build a microprocessor (μP), like the one depicted in fig. 3.3. This is the most prominent example of a synchronous digital circuit (although they do not need to be synchronous). The behavior of the μP can be generally described as follows: They can access to memory (circuit that allows to store digital information for later access) where they store data and instructions. The μP reads the memory to get instructions and execute them. These instructions, if crafted correctly, will manipulate the data until the desired outputs are computed.

Software & Hardware Table 3.1 shows a simple program that reads two operands from memory, adds them, and then stores them back to memory. Instructions are stored in memory as what is called *machine code*, while we usually analyze this code in *assembly language*, which has a one-on-one equivalence to machine code. The circuits are what we call *hardware*, while the set of instructions are what we call *software*. Note that what the system does depends on both hardware and software.

Table 3.1: Example assembly program.

Instruction description	Assembly	Machine code
Read memory address 0 and store it in reg. 1	lw $r1, 0($zero)	8c000000
Read memory address 4 and store it in reg. 2	lw $r2, 4($zero)	8c000004
Add reg. 1 and 2, and store the result in reg. 3	add $r3, $r1, $r2	00000020
Store the value of reg. 3 in address 8	sw $r3, 8($zero)	ac000008

General-Purpose Hardware The main characteristic of a μP is its *instruction set architecture* (ISA), which tells the software programmer how to use it. It includes, for example, what instructions it can execute. The hardware is harder to design and build, but the software can be changed easily, and in general, as many times as we want. Then, it is interesting to a μP with an ISA that makes it useful for a wide range of applications. Thus, do the hard work once, and then, build many copies of the same circuit, so it can serve multiple applications. These μPs are what we call *general-purpose hardware*.

Abstracting Complexity An *algorithm* is the recipe (list of instructions) that indicates how to manipulate data to obtain the desired results. A great thing about μPs is that they decouple circuit design from algorithm design. The ISA *abstracts* (hides) the details of the circuit and how it is built, as this information is not needed for programming the μP. In addition, given that general-purpose hardware is useful for many people, we can afford to build tools to make their usage even easier. For example, instead of writing a software in assembly language, which needs to specify a very detailed description of what the μP has to do, we may create a tool that translates a simpler language (a *high-level language* (HLL), closer to human language) to machine code that can be then executed. This tool is called a *compiler*. In this way, the compiler rises the *abstraction level*, that is, the level of detail needed to describe the algorithm.

Specific-Purpose Hardware However, a specifically designed circuit will be more efficient for a given problem compared to general-purpose hardware. For example, *graphic processing units* (GPU), which are main designed to operate with images, are capable of processing them at a much higher rate. Also, we may design a circuit for even more specific cases, when extra efficiency is required. These circuits are what we usually call *application-specific integrated circuits* (ASIC) [1].

Reconfigurable Hardware Yet, as suggested before, building an ASIC is expensive and time-consuming. An alternative is to use *reconfigurable hardware*. Reconfigurable hardware are circuits that have a special type of software, the *firmware*, that determines how it behaves. This firmware is not made of instructions, but configurations that make it emulate a given circuit. For example, by modify the contents of the LUT in fig. 3.2(b), we may make it behave as an AND gate, instead of an OR gate. Then, the contents of the LUT is firmware.

[1] Note, that technically, all circuits are ASICs, even μP.

FPGAs Among reconfigurable hardware, *field-programmable gate arrays* (FPGA) have attracted a particular interest given the performance-flexibility-cost trade-off they offer. These are constituted of large arrays of LUTs and FFs connected through configurable nets, providing them the ability of emulating any circuit. In addition, they have resources to interface with many other systems supporting a wide range of communication standards (e.g. DDR memories or PCIe buses). Then, although we still have to design the circuits (lower level of abstraction that instructions), with FPGAs we do not have to incur the ASIC fabrication costs. What is more, with time, FPGAs have acquired more resources (special purpose circuit modules) that enable the FPGA designer to create more efficient systems. Nowadays, for example, they have *random-access memory blocks* (BRAMs) or circuits specialized for signal processing, *digital signal processors* (DSPs). For a more detailed evolution of the FPGA through the decades, refer to [18].

3.2 Design Languages

RTL Traditionally, in the last decades, digital circuits are described using *register-transfer-level* languages, like *Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language* (VHDL) and Verilog. As the name suggests, these languages define the systems in a detailed manner, creating registers, specifying combinational logic and connecting them using wires. RTL is later translated to a *netlist* (wires and basic elements of the target technology) by a process called *RTL synthesis*. Then, to obtain the FPGA firmware (called *bitstream*), the netlist is then mapped to specific elements of the FPGA through a group of processes normally named as *implementation*. The more relevant processes are *placement* and *routing*, and thus, implementation is sometimes referred to as place and route (P&R). Throughout these stages, optimizations towards certain goals are applied to the netlist, for example, to obtain a circuit that needs fewer resources (lower area) or higher performance (being able to use higher frequency clocks).

High-Level Synthesis Describing a circuit with an RTL language involves, among others, to perform the *scheduling* of the circuit (to determine when an operation will happen) and, to some degree, the *binding* (to determine which resource will perform an operation). Conversely, because of the abstraction that the ISA creates, software programmers do not generally deal with these details, which enables them to build larger applications and faster. For this reason, FPGA vendors, EDA companies and the academia are pushing towards abstracting complexities of designing circuits. The idea is

to describe systems with high-level languages, which are then translated into circuits. This process is called *high-level synthesis*, and it is implemented by an *HLS compiler*.

HLS compilers In the past decades, many commercial and academic HLS tools have emerged. Apart from the compiler, these tools generally include mechanisms to analyze and test the designed systems. These tools vary in their chosen source languages (from which these generate the RTL), targets technologies, target applications, among others. Yet, nowadays, commercial HLS tools have mainly converged to SystemC [19] (a set of C++ libraries and macros) or a subset of C/C++ as the source languages. We refer to [13], where a detailed survey on these tools can be found.

HLS directives Given the resources available, in this thesis we have used Vitis HLS (previously Vivado HLS) using C++ as the source language. Apart from the C/C++ code, *directives* (special instructions for the compiler) are used to guide the tool towards the desired *architecture* (the structure of the circuit). These directives, for example, can establish the *initial interval* (II), the number of clock cycles required for a module to be ready to consume a new input. Another common use of directives if to determine the structure of memories and select a specific resource for their implementation.

HLS pros and cons Designing hardware using C++ (plus directives) allows obtaining a working system faster, main because:

- The compiler performs the scheduling and biding process, while the user still can have control over them.
- The code can be validated much faster using a C/C++ program instead of an RTL simulator.
- Directives allow a wide design-space exploration, while using the same algorithmic description.
- After code verification and RTL generation, the output system can be automatically validated using the C/C++ test to perform an RTL simulation.
- The source code is less technology-dependent, then favors code reusability.

However, even though HLS compilers have been improving, their use still tends to establish a trade-off between design time and quality of results (performance and/or footprint) [14].

Part II

Case Studies and Challenges Presentation

CHAPTER 4

IMAGE PROCESSING: COMPRESSION IN CONSTRAINED SCENARIOS

This chapter explores the problem of improving image compression when stringent restrictions on errors, bandwidth, latency and/or power are imposed. An introduction of JPEG-LS lossless and near-lossless image compression standard and asymmetric numeral systems is provided, and later, possible areas of improvement of the former are studied. Although JPEG-LS has proved to be very suitable for these scenarios, the results show that its compression can be significantly increased, and tANS is a good candidate to achieve this with a low computational overhead.

4.1 Introduction

Information compressors allow the reduction of bandwidth requirements and, given that data transmission systems tend to demand much more power than computing systems, they are useful as well when energy or dissipation capabilities is limited. For the

This chapter is based on the works published in [20] and [21]: Alonso, T., Sutter, G., & López de Vergara, J. E. (2021). LOCO-ANS: An optimization of JPEG-LS using an efficient and low complexity coder based on ANS, in IEEE Access, vol. 9, pp. 106606- 106626, 2021, doi: 10.1109/ACCESS.2021.3100747. **and** Alonso, T., Sutter, G.,& López de Vergara, J. E. (2021). An FPGA-Based LOCO-ANS Implementation for Lossless and Near-Lossless Image Compression Using High-Level Synthesis. Electronics, 10(23), 2934.

case of images or videos, apart from lossless compression, we may also introduce errors in a controlled manner to improve the compressibility of the data. A particularly convenient way to perform this is to use near-lossless compression, which is a generalization of lossless compression, where the codec user can set the maximum absolute difference (the error tolerance) between the values of an original pixel and the decoded one (in the space domain). When this limit is set to zero, lossless compression is obtained.

These codecs are particularly useful when the data to compress contains very valuable information or, given the nature of the application, a minimum quality must be ensured. Satellite image acquisition is a prominent application of these systems, which have pushed the development of many algorithms and hardware implementations [22, 23]. Additionally, we can find medical applications such as capsule endoscopy [24–28] or portable image devices [29].

New applications emerge in scenarios where traditionally raw (uncompressed) data was transmitted. Given the rapid increase in the data volume generated, image codecs can reduce costs and development time by leveraging already available transmission infrastructure and standards. An example of this in the video broadcasting industry is the use of intermediate codecs (mezzanine codecs), used between initial acquisition and final distribution [30]. In addition, for the manufacturing industry, we can find high frame per second (FPS) infrared cameras [31] producing information that is subsequently processed by an algorithm that may require limitations on the quantization errors to ensure proper operation. Sometimes these are part of closed-loop control systems, which will additionally demand latency limitations to ensure control loop stability.

Particularly for the more demanding applications (low energy, high throughput, low latency), hardware implementations can be needed to better compete with other products in the market or just to meet requirements while achieving real-time compression of the data stream [32–36]. This tends to be particularly true for the encoder side, as in the case of remote sensing, like satellite applications or portable devices.

A codec well suited for these applications is JPEG-LS [37], based on the LOCO-I (Low Complexity Lossless Compression for Images) algorithm, which is known for its great trade-off between complexity and coding efficiency and amenable hardware implementation [38, 39]. This led to the development of multiple hardware architectures [23, 27, 40–45] and the utilization of an adapted version in the Mars Rover mission (NASA) [22]. An extension of the standard was developed [46], mainly, to improve the compression rate when coding lower entropy distributions like those that arise when the error tolerance is greater than 0. However, this came at the expense of increased

complexity, among other reasons, because it uses an arithmetic coder.

After the inclusion of the JPEG-LS standard extension, a new coding scheme was developed, Asymmetric Numeral Systems (ANS) [47], which presents a coding efficiency that rivals the arithmetic coder, but capable of achieving compression speeds of Huffman coders for both hardware and software implementations [48, 49]. This new coding technology opens a door towards more efficient compression and low complexity compression. For this reason, and given the observation of an increasing need for better codecs, we studied whether, based on the JPEG-LS standard, increased compression at low computational complexity could be achieved employing ANS.

4.2 JPEG-LS

4.2.1 JPEG-LS Baseline Algorithm

JPEG-LS was designed mainly for lossless compression with low complexity in mind and the objective to supersede the previous algorithms like the lossless mode of JPEG [50] and PNG [51]. Fig. 4.1 shows a high-level block diagram of the JPEG-LS encoder algorithm, which is based in LOCO-I [39, 52].

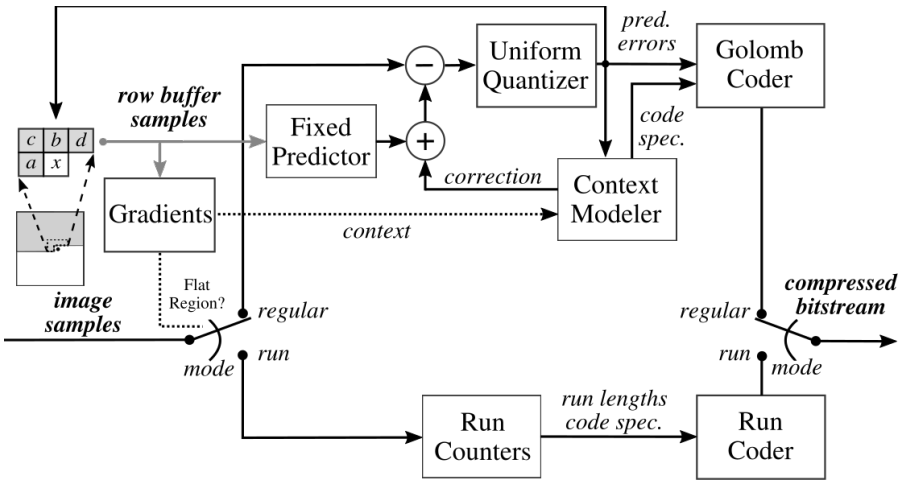


Figure 4.1: High-level JPEG-LS encoder block diagram. Source: Adapted from figure 1 of [52]).

It can be appreciated that it processes image samples using one of two modes, the regular and the run mode. In the regular mode, a prediction is computed and then corrected with an adaptive mechanism, resulting in a prediction error. This error is then quantized using a uniform mid-tread quantizer with a bin size $\delta = 2 * NEAR + 1$,

where $NEAR$ is a parameter chosen by the user, which is equal to the maximum possible error of a pixel value in the decoded image. The quantized error is then coded by a low complexity adaptive block coder based on Golomb codes [53], which the authors call Golomb-power-of-2 () codes.

As the GPO2 coder does not perform well when symbols come from a low entropy source, an adaptive run-length coder is used when smooth surfaces are detected by the gradients surrounding the current image sample. In the run mode, the run-length count is incremented when $|a - x| \leq NEAR$, where a is the pixel value when the count started and x is the new pixel. It is easy to see that, in both modes, lossless compression is obtained when $NEAR$ is set to 0.

To adapt the codes, contexts are used to keep prediction error statistics, which select coder parameters. These contexts are gradient defined. Gradients surrounding the new image sample are computed and then quantized separately, obtaining a vector of integers. The resulting vector is mapped to an identifier, which is used to access and update context statistics. In [52] a detailed description of the codec procedures can be found.

4.2.2 JPEG-LS Extension

An extension of the standard [46] was proposed, based on LOCO-A (presented in [39]), changing the GPO2 and run-length coder used in LOCO-I by a single arithmetic coder and adapting the error distribution estimation procedures. These modifications closed most of the existing gap with CALIC image coder[1] [54, 55] at the cost of increasing the complexity of the system. This extension comes from the authors' recognition of the limitations of the original coder when dealing with low entropy distributions, as those that occur in near-lossless operation. In general, the higher error tolerance (parameter NEAR in JPEG-LS), the lower the entropy of the resulting quantized error distributions.

4.2.3 JPEG-LS Hardware Implementations

Several hardware architectures for JPEG-LS have been published [23, 27, 40, 42–45], but only a few are standard compliant. One of the main reasons for the lack of compliance is not supporting the run mode (which many deem as optional, although it is not [37]). In general, this is done to further simplify the hardware implementation. In [45] it

[1]CALIC was one of the proposed algorithms for JPEG-LS. Although it has a slightly higher compression ratio when employing an arithmetic coder, LOCO-I was finally selected for the standard given its better complexity-efficiency trade-off. with has a slightly higher compression ratio, at the cost of higher complexity

was found complex to implement, while others noted the run mode is rarely used when losslessly compressing images coming from sensors, so decided not to implement it. Although this is generally true for lossless coding, long runs can arise, for example, when sensor saturation occurs. In satellite images, clouds tend to produce this effect. For near-lossless operation, not supporting the run mode greatly impacts compression rates, as this mode is particularly important to complement the main weakness of prefix codes used in JPEG-LS when it comes to low entropy distributions (not able to produce an average code length below 1 bit for any symbol).

Another reason why the implementations did not adhere to the standard was the introduction of algorithm modifications to increase system throughput. Hardware implementations face mainly two bottlenecks: the context update, and the pixel quantization (and reconstruction) procedures. The latter only applies to near-lossless compression. Most implementations try to cope with these limitations by modifying the original algorithm and/or not supporting near-lossless compression (and thus avoiding the second bottleneck). Often, these modifications reduce the compression ratio.

Only two of the mentioned implementations support near-lossless compression [23, 40], but neither is standard compliant. In [23], several modifications are presented to the decorrelation and entropy coding stages, chiefly, the error tolerance ($NEAR$ parameter) is modified within an image according to custom logic and only the GPO2 coder is employed, using a new adaptation algorithm. A close to standard compliant implementation is presented in [40], but it does not support the run-length coder. Although the performance of these two implementations cannot be directly compared, given the great difference in the technologies used in the experiments (0.22 μm process Xilinx XQR4062 in the former versus 40 nm process Xilinx Virtex 6 in the latter), the highest performing implementation supporting near lossless in the literature is the latter (51.68 Mpixels per second).

4.3 Asymmetric Numeral Systems

ANS is a new series of low complexity alternatives to arithmetic coding, initially introduced in [47], and later extended and compared to state-of-the-art compression algorithms, such as Huffman and arithmetic coding, in [48, 49].

ANS provides several possible algorithmic alternatives to implement coders. Particularly, tabled ANS (tANS) has the following properties:

- Suitability for high cardinality symbol sources.

- Capable of being used in adaptive coding settings.
- Able to match arithmetic coder [56] coding efficiencies (having an efficiency-memory resources trade-off).
- Moderate memory resource requirements.
- Has high-throughput implementations. For Field Programmable Gate Arrays (FPGA), encoder architectures were studied in [57] and decoder in [58], which can outperform Huffman decoding [59].

4.3.1 tANS Operation

From a black box perspective, tANS works as a Finite state machine (FSM) where the symbol to encode is the input and the current state is an integer, the ANS state, where ANS stores fractional bits of information. The output of the FSM ROM has the next state and the number of bits to take from the least significant part of the current state, which are then stored in the output bit file. From its design, tANS is meant to be implemented as a microcoded FSM (at least partially), and the FSM ROM is referred to as the *tANS table*. After a block of symbols is finished, the final state needs to be stored in the output bit file.

For the decodification, the binary bits are appended to the ANS state[2], until it is in a certain range (determined by the configuration of ANS). Then, this state is used to address the decoding table, obtaining the encoded symbol and the previous state. As implied by the decodification process, an ANS state is directly matched with a source symbol. Modifying the assignment of states to symbols changes the average number of bits ANS is going to generate for each of the source symbols.

tANS can be used to implement an adaptive coder, given that switching to a different table changes the distribution ANS is tuned to. Using a particular table is referred to as an *ANS mode*. Of course, the decoder has to have the means to choose the same ANS mode that the encoder chose for each symbol. However, more attention has to be paid when using ANS in an adaptive manner, given that symbols are decoded in the opposite order they were coded (the last symbol coded is the first symbol that is decoded).

For more in depth explanation of the ANS algorithm and hardware implementations, refer to [48, 57, 58].

[2] $state \leftarrow (state << 1)|new\_bit$

4.3.2 Coding Efficiency

In general, the more bits used for the state, the more precisely the coder can be tuned to the desired distribution, which leads to a more efficient compression. Fig. 11 of [48] shows simulation results to understand the relationship between the number of ANS states used, the symbol alphabet size and the Kullback–Leibler divergence (KLD), also presenting the approximation $KLD \approx 0.5/(k)^2$ with $k = |S|/|A|$, where S is the set of states (in this work, it is generally assumed to be 2^{state_bits}), A is the set of symbols and $|\cdot|$ denotes the cardinality of the set. Eq. 4.1 summarizes our experience using the simple non-fine-tuning heuristic algorithm provided in the original work to create the tANS tables.

$$0.05/k^2 \lessapprox KLD_{tANS} \lessapprox 0.5/k^2 \tag{4.1}$$

Increasing precision comes at the cost of increasing memory requirements for the FSM ROM. However, the impact of this increment depends on the actual implementation and, as shown in [57], efficient architectures exist for large table configurations.

4.4 Problem Analysis

4.4.1 Test Image Dataset

For this analysis, the 8-bit gray image dataset maintained by Rawzor [60] was used. A description of the dataset images can be found in table 4.1, where the entropy was computed using a slightly modified version of the JPEG-LS baseline model (described in section 4.4.2).

4.4.2 JPEG-LS Optimization Potential

The aim of this section is to establish a theoretical limit on improved compression due to the optimization of the prediction error coder for JPEG-LS given its statistical model.

4.4.2.1 Theoretical Limit of Coder Optimization

To understand the impact that a new coder could have, the average bits per pixel, bpp, obtained by JPEG-LS coder (using the official reference implementation in [61]) was

Table 4.1: Rawzor 8-bit gray dataset [60] description. Entropy estimation based on a modified version of the JPEG-LS baseline model (described in section 4.4.2)

Image	Height x Width	Entropy	Type
artificial	2048 x 3072	0.7625	Synthetic
big_building	5412 x 7216	3.5437	Photographic
big_tree	4550 x 6088	3.6886	Photographic
bridge	4049 x 2749	4.1222	Photographic
cathedral	3008 x 2000	3.5343	Photographic
deer	2641 x 4043	4.6033	Photographic
fireworks	2352 x 3136	1.4218	Photographic
flower_foveon	1512 x 2268	1.9737	Photographic
hdr	2048 x 3072	2.1015	Photographic
leaves_iso_1600	2000 x 3008	4.4505	Photographic
leaves_iso_200	2000 x 3008	3.7648	Photographic
nightshot_iso_100	2352 x 3136	2.0217	Photographic
nightshot_iso_1600	2352 x 3136	3.9333	Photographic
spider_web	2848 x 4256	1.6722	Photographic
zone_plate	2000 x 3000	7.3368	Synthetic

compared against the average symbol entropy using the statistical model employed in the standard to estimate the prediction error probabilities (coder symbols).

Although the average symbol entropy would not consider the effect of the compressed image header file size, this does not have noticeable impact, particularly for the image sizes of the used dataset. In JPEG-LS, the context with all its quantized gradients equal to 0 is handled differently as it is coded using the run-length coder, but in this analysis, given that the same statistical model is used for all contexts, it is treated as the rest.

In the standard, the error, ϵ, probabilities are estimated using a two-sided geometric distribution (TSG) as follows:

$$P(\theta,s)(\epsilon) = C(\theta,s)\theta^{|\epsilon - s|}, \epsilon = 0, \pm 1, \pm 2, ..., \tag{4.2}$$

where θ and s are the distribution parameters and $C(\theta,s) = (1-\theta)/(\theta^{1+s} + \theta^{-s})$ is a normalization factor. $\theta \in (0,1)$ controls the rate of decay of the probabilities and $s \in (-1,0]$ is the fractional bias (the sign of s is inverted compared to [52]).

In JPEG-LS baseline, s was decided to be in $(-1,0]$ given that it was beneficial for their coding procedures. However, when computing the average symbol entropy, the bias cancellation procedure was configured so that $s \in (-0.5, 0.5]$, like in the standard extension. For this reason, the error sign flip applied when $s > 0$ was introduced, also

employed in the standard extension. Additionally, the alternative model and estimators for the TSG proposed in [62] were used. This change does not imply a modification in the distribution, but just a re-parametrization that simplifies the sequential parameter estimation. In this alternative model, each integer ϵ is mapped to a tuple (y,z), where:

$$y = y(\epsilon) \triangleq \begin{cases} 0, \epsilon \geq 0 \\ 1, \epsilon < 0 \end{cases} \tag{4.3}$$

and

$$z = z(\epsilon) \triangleq |\epsilon| - y(\epsilon) \tag{4.4}$$

Then, if $\epsilon \sim TSG(\theta, s)$, the variable $y \sim Bernulli(p)$ (where $p = (\theta^{1+s})/(\theta^{1-|s|} + \theta^{|s|})$) and the variable $z \sim Geometric(\theta)$ with the same θ as ϵ. For sample $t+1$, p is estimated (using $Beta(1/2, 1/2)$ as a prior) as follows:

$$\hat{p} = \frac{N_t + 1/2}{t+1}, \ where \ N_t = \sum_{i=1}^{t} y_i \tag{4.5}$$

In [62] an optimal estimator of the probabilities of z_{t+1} is provided. However, the following estimator was used:

$$\hat{\theta} = \frac{S_t + \alpha}{S_t + t + \alpha + \beta} \tag{4.6}$$

where α and β are the parameters of the $Beta(\alpha, \beta)$ function used as a prior probability distribution. This last estimator, as noted by the authors of the model, is suboptimal, but, in our experiments, it performed almost as well as the optimal one when using the same priors, with the advantage of being computationally simpler. To reflect the fact that as $NEAR$ increases, θ decreases, $Beta$(.5/(1+NEAR/2),.5) was the prior used in the experiments.

The results can be seen in table 4.2, where the column labeled as "Entropy_orig_ctx" was obtained using this model.

It can be seen that the larger the error tolerance, the less efficient JPEG-LS tends to be, having an inefficiency ranging from 1.7% for lossless compression to 9% for an error tolerance of 10.

Table 4.2: JPEG-LS bpp vs TSG models estimated entropy.

Error	JPEG-LS	Entropy_orig_ctx	Entropy_fix_ctx
0	3.32	3.26 (1.7%)	3.26 (1.7%)
1	2.12	2.09 (1.1%)	2.01 (5.1%)
2	1.65	1.63 (1.4%)	1.51 (8.1%)
3	1.40	1.35 (3.2%)	1.24 (11.1%)
4	1.23	1.16 (5.4%)	1.06 (14.3%)
5	1.11	1.03 (6.8%)	0.92 (17.1%)
6	1.01	0.93 (7.9%)	0.81 (19.4%)
7	0.92	0.84 (8.5%)	0.72 (21.1%)
8	0.84	0.77 (8.3%)	0.65 (22.2%)
9	0.79	0.72 (8.4%)	0.6 (24%)
10	0.73	0.67 (9%)	0.55 (25.6%)

4.4.2.2 Optimization by Fixing Gradient Quantization

In JPEG-LS, gradient quantization is a function of the NEAR parameter. As a result, the central quantization bin is expanded and the rest are scaled proportionally. Probably, the quantizer was designed in this manner to be able to use the run-length coder in this lower entropy scenario, but it was not considered necessary for a coder capable of handling low entropy distributions. For this reason, the quantization thresholds were fixed to those computed using NEAR = 0. As a result, the column labeled as "Entropy_fix_ctx" in table 4.2 was obtained. As it can be seen in the table, this change would allow getting better compression ratios as the error tolerance increases. As expected, although this change reduced the estimated symbol entropy, it worsens the performance of JPEG-LS.

Additionally, a hardware implementation of the codec that supports multiple values of NEAR is slightly simplified, resulting in smaller and faster logic for the gradient quantization. Thus, the pixel reconstruction bottleneck is alleviated.

It is worth noting that the changes introduced to the model, particularly to the gradient quantization, did not always result in an improvement in the estimated entropy. For example, the fixed gradient quantization worsens the entropy estimation of the synthetic image "zone plate" for all $NEAR > 0$. However, the changes resulted in reduced entropy estimations in most cases, particularly for the photographic images, which are more relevant given the applications of low complexity lossless and near-lossless compression.

4.5 Research & Development Goals

Many applications push towards higher resolution and higher FPS, but because of link bandwidth, current infrastructure, cost and/or energy limitations they often require more efficient image codecs. Although, new hardware architectures may enable the implementation of more complex compression methods in highly constraint scenarios, the possible performance and resource requirement improvements are bounded by the underlying algorithms.

ANS can match arithmetic coder efficiencies, and it can provide practical increases in compression compared to block coding, particularly in the case of very skewed probabilities (like prediction errors in image compression) [63]. Given the significant JPEG-LS optimization potential and the appearance of this new entropy encoding method, it was considered worth exploring whether an improved codec could be obtained through algorithm-hardware co-design.

LOCO-ANS (Low Complexity Lossless Compression with Asymmetric Numeral Systems) codec emerged as a result of this research and development. Chapter 7 delves deep in the algorithmic description, configuration and evaluation, while in chapter 8 an FPGA implementation is presented and analyzed.

CHAPTER 5

COMPUTER NETWORKS: 100 GBE FLOW METERING

This chapter explores the problem of implementing a flow metering system for 100 Gbps Ethernet (GbE) links in FPGAs. An introduction of IP flow metering and general background on the topic is provided. Additionally, the problem constraints are analyzed, where the main issue is identified, consisting in the implementation of high-throughput and complex read-update processes dealing with significant propagation delays associated with the memory system.

5.1 Introduction

Network traffic monitoring is required for performance assessment, traffic classification and the detection of problems, such as congested or broken links, as well as Denial-of-Service (DoS) attacks. This information, in turn, is used in the high-level management of the network infrastructure (routers, antivirus, firewalls). Yet, given that the link bandwidth has increased faster than the performance of μP, complex real-time analyses using them have become inviable. Then, to be able to deal with increasing

This chapter is based on the work published in [64]: Alonso, T., Ruiz, M., Sutter, G., López-Buedo, S., & López de Vergara, J. E. (2019, April). Towards 100 GbE FPGA-Based Flow Monitoring. In 2019 X Southern Conference on Programmable Logic (SPL), Buenos Aires, Argentina (pp. 9-16).

amounts of data, these analyses have been moving away from packet level and towards *IP flow level*.

According to the IPFIX standard [65], a *flow* is defined as "*a set of packets or frames passing an observation point in the network during a certain time interval. All packets belonging to a particular flow have a set of common properties*". Typically, IP addresses, layer 4 ports and transport protocol are chosen, conforming a tuple that identifies a flow. This is how Netflow v5 defined it [66], the first widespread protocol for flow export, although since 2013 IPFIX has become the Internet standard for this purpose.

For each flow, statistics are collected, creating a *flow record*. From these records, it is possible to know information such as the status of connections, their bandwidth utilization, round-trip time or detect attacks on a server [67–69]. Flow metering, at the expense of losing per packet information, has the advantage of requiring considerably less compute and memory resources. Yet, after detecting an anomaly, for example, a system might proceed to filter the desired traffic for in-depth investigation.

However, to create flows records in high-speed links, we still need to have great computational power and high-performance memory, given that packet level processing is required. For this reason, commodity off-the-shelf servers struggle to deal with this task and, considering current trends, the situation is likely to worsen. What is more, multicore architectures are not very effective to increase the processing power, as in these scenarios the workload balance across cores cannot be assured [70].

In this light, and considering that flexibility is an important factor to be able to incorporate new protocols, we studied the use of FPGAs to offload the flow metering task.

5.2 Flow Monitoring

A flow monitoring system consist of three components, which are shown in Fig. 5.1. The *flow exporter* creates flow records from the traffic going through the observation point and then sends these records to one or more *flow collectors*, where they are stored and processed. Finally, an *analysis application* inquiries the collectors and analyzes the data. Probes can be placed in different nodes of the network, and normally high-aggregation links are chosen.

The *flow table* is the memory that contains the records of active flows in the exporter. Every time a packet is received from a new flow, an entry is created in the table, which will be released when the flow record is transmitted, that is, the flow is exported. Most commonly, flows are exported for the following reasons [71]:

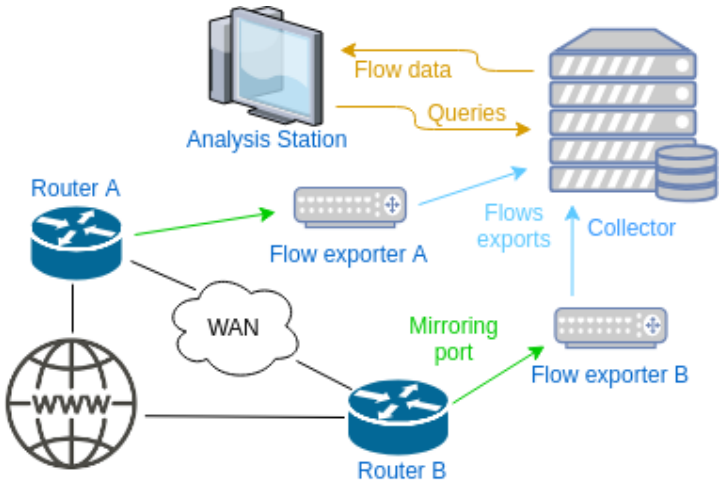


Figure 5.1: Typical flow monitoring system. Source: [64]. © 2019, IEEE

- *Inactive timeout*: A flow is said to be active if at least one packet belonging to it was received in the last T_i seconds. When a flow turns inactive, it is exported. T_i is usually set by default to 15 seconds.
- *Active timeout*: If a flow remains active for more than T_a seconds, it is exported. When setting this parameter, it is important to consider that larger active timeouts favor greater information aggregation, but also reduces the analysis resolution. The flow collector will then be blind of troublesome flows for T_a seconds.
- *Resource constraints*: For example, if a new flow entry needs to be created and there is no place in the memory to store it, one of the flows that generate the conflict is exported. This situation is called a *collision*.
- *TCP flow control*: According to RST or FIN TCP flags the flow may be exported, sometimes, after a waiting period.

For an in-depth discussion on the topic, refer to the tutorial by Hofstede *et al.* [72].

5.3 State-of-the-Art

The aggregation of traffic in flows is usually carried out in routers and switches, taking advantage of the resources already available in them, as is the case of Cisco's NetFlow [73]. However, they usually use packet-sampling techniques, which are a function of their level of congestion. Although there are ways to mitigate it [74], this inevitably leads to data loss. 10 Gbps networks (and beyond) are becoming more frequent, especially in data

centers, making this task even more challenging. Therefore, if a great precision in the analysis is desired, other alternatives have to be considered. Several implementations of flow exporters have been proposed using CPUs, GPUs, FPGAs and hybrid versions, distributing the load of the system in different ways among these devices.

Miravals *et al.* [75] proposed light-weight approximate algorithms to detect TCP retransmissions in a software flow probe using custom driver to capture incoming packets. The system was tested using a high-end E5-2630 processor and stressed using 10Gbit traces, achieving 1.6 million packets per second (Mpps) with a single core. Paula Roquero *et al.* in [76] proposed a CPU-GPU flow exporter for 10 GbE links capable of obtaining complex information related to the TCP protocol, such as the detection of retransmissions with memory at the packet level, achieving a processing rate of up to 4.4 Mpps. Marco Forconesi *et al.* [77] presented an architecture to export flows in 10 GbE networks based on the NetFPGA-10G platform, which is capable of handling the maximum packet rate, 14.88 Mpps without sampling and up to 786,432 concurrent flows using an SRAM memory for the flow cache. Viktor Puš *et al.* in [78] deployed an exporter running on a 20-core server with a custom FPGA (Virtex-7 H580T) based Network Interface Card (NIC), capable of supporting the export of flows from 100 GbE links. Here, according to the result of a hash function applied to header fields of the packets, the NIC injects the packets into 16 queues allocated in main memory (64 GB DDR4) of the two E5-2660v3 CPUs, which implement the flow exporter.

In the latter case, the services provided by FPGA enables the system to support 100GbE, but the CPU continues doing most of the processing, requiring the FPGA to move to main memory information at the packet level, thus utilizing a large PCIe bus bandwidth (approx 22% of PCIe Gen3 x16 considering 23 bytes per packet, as in the TCP session exporter application introduced in section 5.4). Also, a very high-performance hardware (processor, memories, motherboard) is needed to keep up the processing pace. In this line, notice that either a somewhat uniform distribution of packets per core or over-dimensioned CPU cores are required. However, as the mentioned before, for online traffic analyzes the workload balance across cores cannot be assured [70]. This particularly applies when IP address fields are used in the flow definition, given that these addresses tend to be distributed according to a quasi-Zipf law [79], and an even skewer quasi-Zipf fitted the packet per flow distribution in [75] (TCP packets only and using IPs and ports to define the flow). This situation can be exacerbated under situations like some Denial-of-service where a flood of short packets towards a specific IP is sent, which may cause all these to go to a single core.

Hence, it was decided to understand the challenges of offloading part or the whole 100 GbE flow aggregation process within an FPGA. The main goal of the system is to thin down the amount of information forwarded to either a host CPU or the net (network attached and host detached deployment). This also favors a more energy-efficient solution given that, in general, most of the energy is expended in the data movements, at the same time, preventing possible bottlenecks in this area. Notice that, from its definition, a flow can include either all the packets in the observation point (very low memory requirement, high aggregation), just a few packets (very high memory requirement, low aggregation) and everything in between. Then, depending on the number of concurrent flows and the available memory, the metering process is either partly or fully offloaded. This work is also envisioned to pave the way towards flow metering of higher link speeds.

5.4 Problem Analysis

5.4.1 Scenario

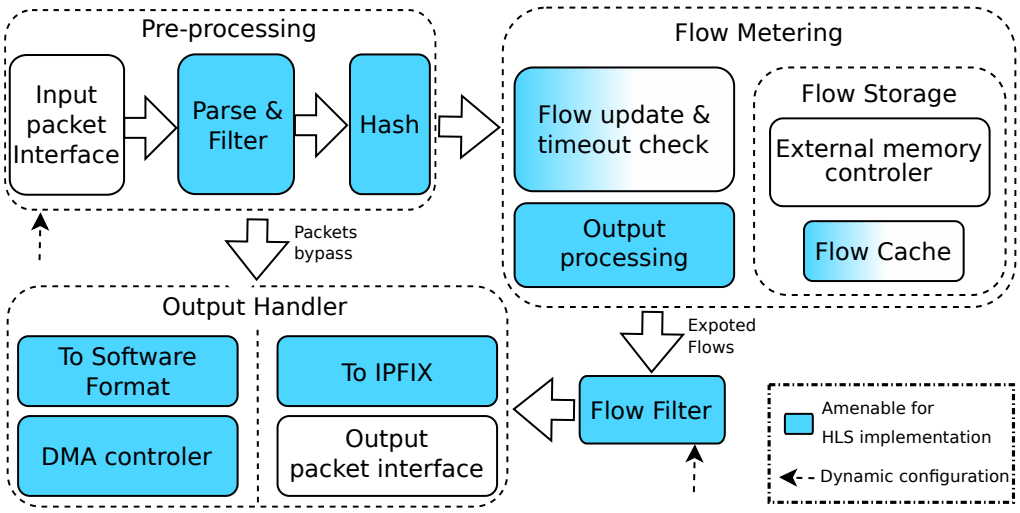


Figure 5.2: Flow metering system high-level diagram.

Fig. 5.2 shows a generic block diagram of a hardware system for flow metering offloading. The Pre-Processing modules include packet level operations like extracting the required packet fields, while Flow Metering ones create flows records, storing them in the flow table. Optionally, flow inspection and filtering might take place before they are sent to the output (exported). The FPGA be a stand-alone network probe exporting flows over the network, or alternatively, it can work as a NIC attached through PCIe to a host machine, which would post-process the exported flows.

In particular, we are going to consider two flow metering applications:

- **TCP flows with retransmission detection:** This case evaluates a somewhat typical flow metering scenario, very similar to [75]. The focus is on the number of TCP retransmissions per flow as the principal Key Performance Indicator (KPI), which is related to packet losses, or increased network and devices latency. To detect retransmissions, the Out-of-Order heuristic algorithm is used [75]. Additionally, for each flow, other volumetry measures are taken.
- **VPN volumetry:** This application was developed as a prototype for an internet service provider, who had its network divided into up to $2^{12} = 4096(4K)$ VPNs, and required to gather statistics for each of them in a highly aggregated node of the network using 100GbE links. Packet and byte counters per EtherType and per layer 4 protocol are kept, and also, packet counters for different combinations of TCP flags. Optionally, histograms of packet size and packet inter-arrival times might be desired. These records need to be exported every $1s$.

5.4.2 System Constraints

Independently of the definition of a flow, the system has to be able to read and update the flow records every 6.72 ns to fulfill the maximum packet rate requirement. Conversely, the capacity of the memory system, it is dependent on the flow definition. However, for a given timeout setting, there is an upper bound on the number of concurrent flows. For example, for the typical $15s$ inactive timeout and 30 min. active timeout, there are up to 2.23 Giga concurrent flows. This corresponds to the case when all the packets in the link have the minimum size and all of them belong to different flow. Depending on the definition of the flow, this bound may be lowered.

5.4.2.1 Memory for TCP Flows

For the TCP flows application, TCP packets are filtered and flows are defined using a 4-tuple: IP addresses (source and destination) and layer 4 ports (source and destination). Given the number of possible 4-tuples and the timeouts used in the practice, the mathematical upper bound on the number of flows will be determined by the timeouts. However, according to the study presented in [80], for 100GbE TCP traffic, the first and third quartiles of the distribution of concurrent flows are 10 Mega and 36 Mega flows (Mflows). Although they indicate the concurrent flows could be above 100 Mflows, they recognize their results tend to overestimate flows in highly aggregated traffic (Multi-Gbps).

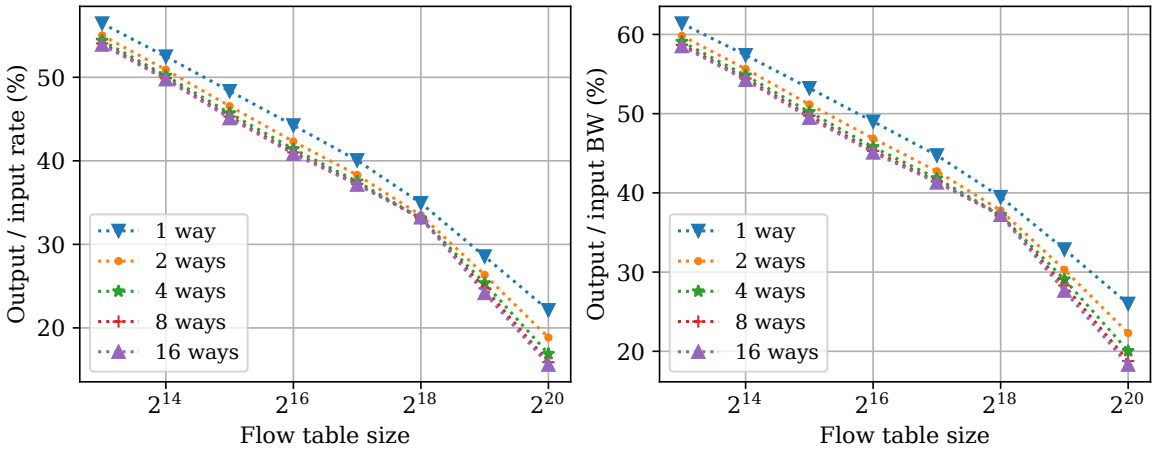
According to these numbers, the memory system capacity should be in the GB range. DDR3-4 memories can provide the required capacities, but they would struggle to achieve the required bandwidth without a memory hierarchy. In addition, the latency of these memories plus that of the memory controller is in the order of hundreds of nanoseconds according to our benchmarks. Then, in this application, we may aim at partially offloading the task with on-chip memories or use latency masking techniques, like caches and prefetching, to deal with external memories.

To assess the impact of using on-chip memories, we conducted simulations for memory sizes ranging from 8 KiFlows (2^{13} flows) to 1 MiFlows (maximum cache size of Cisco's Netflow [81] at the time of doing the experiments). The memory was implemented as a hash table with 1 to 16 ways (typical μP cache). For each new flow, an entry is created in the cache. If there is no slot available (*collision*), one of the flows in the table is expired to make room for the new one. The selection of which flow to expire is based on the RST/FIN TCP flags and the timestamp of the last packet (the oldest flows are expired). Then, for each collision, the system would either export a flow (partial offload scenario) or store it in the external memory (full offload scenario).

Output bandwidth was estimated using 3 output packet structures: single packet flow (23 bytes), less than 40 packets in the flow (31 bytes) and the rest (37 bytes). Notice that using different output packet structures allows eliminating the possibility of bandwidth overhead, as in the worst case where all packet create collisions, the output of the system would be as if no offloading module were in place. As a result, the upper bound for both the output packet rate and bandwidth, are those of the system without any offloading.

As we did not have access to any 100 GbE trace, we created one by merging ten traces provided by CAIDA, captured during 2018 from a 10 GbE link [82]. Timestamps were edited so that the captures started at the same time, and thus, bandwidth and packet rate would add up. This trace should result to be a hard 100 GbE case as the number of concurrent flows would also add up, and present a 10 times worse temporal flow locality, lowering the memory hit rate. The number of estimated concurrent flows in the trace was in the order of 30 Mflows, which is in the previously suggested range, but clearly above the median value (supporting that the used trace could be considered a hard case).

Fig. 5.3 shows the results of the simulations, where it can be appreciated that significant reduction on output packet rate and bandwidth is achieved with table sizes that can be implemented with on-chip memories of Xilinx's Ultrascale+ devices. Also, using associativity levels greater than 2 (probably better in the 4-8 range) allow making the most out of this limited resource.



((a)) Output vs Input packet rate

((b)) Output vs Input Bandwidth

Figure 5.3: Performance of 100 GbE TCP flow metering with on-chip memory as a function of memory size and associativity simulation results

With these results and using DDR memories with the tested controller, the mean supported packet rate would be in the 10 to 40 Mpps. Although this is clearly below the maximum packet rate, the average packet rate (≈ 16 Mpps) falls within the range. To further improve this rate and the capacity to support short packet bursts, other latency masking techniques, such as pre-fetching and out-of-order processing, could be used.

5.4.2.2 Memory for the VPN Volumetry Application

A fundamentally different scenario in terms of required memory capacity is presented in the VPN volumetry application. In this case, the on-chip memory is enough to hold all the flow records (2^{12} flows), requiring a maximum memory capacity of ≈ 600 KB. Notice that in this case, although the task is clearly suitable for a complete offloading on an FPGA, supporting high packet rates might still be a challenge to CPU implementations. For this application, given the small output information volume, a medium-sized chip, like Xilinx ZU5EG, could be suitable to completely offload the application, using the ARM cores for further processing and communication.

5.4.3 Required Hardware

Most of the modules of the system can be implemented with feed-forward logic (no feedback paths), which can be heavily pipelined to support higher link rates. Yet, this is not the case for the core flow metering processes, as they have read-after-write (RAW)

dependencies. As previously explained, the system has to perform up to 148.8 million table updates per second, while, at the same time, it has to scan the table to expire flows. In addition, big on-chip memories are needed, so large propagation delays are expected given that FPGA memory blocks are distributed in the chip. Therefore, and considering the frequency limitations of FPGAs, an efficient architecture is required. What is more, for TCP flows, multi-way caches are desirable, but this increases logic complexity, which stresses the need for an efficient architecture.

Trying to optimize the productivity measure, we assessed the suitability of HLS to implement the hardware components. This was evaluated through algorithm analysis and the use of proof-of-concepts, and, as a result, many modules were considered amenable for HLS implementation, as illustrated by fig. 5.2. Using HLS, however, it would be a real challenge to develop the modules that handle the flow table, and even not possible (given that target frequency) for the cases requiring complex operations and multi-way caches. This is because of the presence of the mentioned data dependencies within and among processes, which can limit frequency if the used architecture is not properly optimized.

5.5 Research & Development Goals

In general, the read-update pattern required to handle the flow table and the increasing need for larger memories is present in other applications, like sorting algorithms, heavy-heater estimation or context-based compression. Also, these may require or benefit from having more than one process doing these read-update operations, as in the case of the flow metering problem. We therefore consider it worth exploring how to best implement these high-throughput and complex read-update systems that need to handle large memories. For this reason, chapter 9 studies, in particular, the capabilities of FPGAs to offload the flow metering task for 100 GbE links and, in general, architectures to better handle processes with RAW dependencies using medium to large on-chip memories, that is, from a few BRAMs to a significant fraction of the available memories.

In addition, the need for larger memories and more complex systems can inevitably lead to increased latencies in the read-update cycle, affecting performance. Therefore, latency masking techniques might be key to keep up with the processing need's pace. Taking this into account, chapter 10 analyzes the conditional stalling technique and how to employ it to increase performance when RAW dependencies are present. As that chapter shows, this dynamic scheduling technique, not present in current commercial HLS compilers, is well suited to the flow metering problem.

CHAPTER 6

AI: Implementation of Large CNN Accelerators

This chapter explores the problem of implementing large CNN accelerators in FPGA accelerator cards. Background on the topic is provided, introducing the application and the employed framework. Finally, it presents the design challenges of maximizing the performance and scalability of such accelerators, particularly for implementations using non-monolithic FPGAs. The most relevant are the need of considering coarse level placement of modules for control signal connection and balancing resources across FPGA regions.

6.1 Introduction

One of the first real-life applications of CNNs was demonstrated by LeCun *et al.* using their LeNet-5 for optical character recognition [84]. Since then, CNNs have been utilized for a myriad of applications in different fields, like advanced driver assistance systems and self-driving cars [85, 86], robotics [87, 88], search engines and ecommerce

This chapter is based on the work published in [83]: Alonso, T., Petrica, L., Ruiz, M., Petri-Koenig J., Umuroglu Y., Stamelos I., Koromilas E., Blott, M. & Vissers K. (2021). Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning. ACM Trans. Reconfigurable Technol. Syst. 15, 2, Article 15 (June 2022), 34 pages. https://doi.org/10.1145/3470567. © 2021 Association for Computing Machinery . This project was done in collaboration with the Xilinx Research Labs in Ireland, working as a research intern and later as a visiting scholar

using content-based image retrieval query (reverse image search) [89–92], computer-aided medical diagnosis [93, 94], , astronomical image processing [95–97] , high-Energy Physics [98–100] and civil engineering [101, 102].

These applications rely on the CNNs capacity to do classification, detection and segmentation of images (understood as N-dimensional arrays of data). Artificial neural networks, and in particular CNNs, have become very popular, not only for achieving the best (or among the best) results in these tasks, but also because of the development of methods to automate the learning process, for example using back propagation [103].

Due to the high compute and memory intensity of high-accuracy CNN, combined with the high-throughput and even real-time and tight power constraints that many applications have, hardware acceleration is often sought. GPUs are a popular alternative for both training and inference. A well-known example of this is AlexNet [104]. Some companies have gone further, developing ASICs of their own for this task, like Google's tensor processing unit (TPU), achieving not only greater throughput than GPUs but also are more energy efficient, thus reducing operational costs [105]. FPGAs are also used for CNN acceleration given their customization capabilities, leading (sometimes jointly with other technologies) in solution size, interfacing capacity, performance per cost, performance per watt and inference latency [93, 100]. Considering recent advancements in FPGAs and CNN architecture, FPGA are major contestants toward being the main CNN acceleration technology in the near future [2, 106].

Matrix of processing engines (MPE), an architecture often employed by ASIC CNN accelerators, is a popular alternative among FPGA implementations [107–110]. However, they can also leverage their reconfigurability to capitalize the streaming nature of CNNs using a feed forward dataflow, like fpgaConvNet [111], FINN [112], and ReBNet [113]. A dataflow accelerator (DFA) implements all the CNN layers in parallel, thus able to run them concurrently. As the implemented hardware for a given layer is not shared, logic can be customized to the particular characteristics of the layer. Additionally, given that in these architectures CNN weights are usually stored using on-chip memory (OCM) close to the layer logic, external memory operations are considerably reduced or eliminated. This favors energy efficiency, as external memory transactions tend to consume two orders of magnitude more energy than compute units [114]. What is more, when increasing the DFA throughput, latency is reduced proportionally, whereas GPUs or MPE often resort to increased batch sizes to trade latency in favor of higher throughput.

However, the sources of the DFA's strengths also determine their limitations. As they tend to rely on OCM, their capacity and count restricts the size of the implementable

CNNs (measured by the bits required by weight parameters). To ease the problem, implementations may resort to an off-chip memory, but generally only a few layers could use it to avoid memory bottlenecks and frequency penalties. In additional, to be able to adapt timely, DFAs need to be automatically generated from CNN high-level language descriptions, like Pytorch, given that a custom pipeline is implemented for each instance. One of such generator systems, is the open-source framework *FINN compiler* [115] provided by Xilinx. However, compared to MPE implementations (often using expert handcrafted RTL), the generated hardware tends to be less optimized, and thus achieving lower operating frequencies. This is particularly relevant for large multi-die FPGAs, as physical constraints have a greater impact.

Given the great performance potentiality of DFAs and with the final goal of finding solutions to them, we explored how to implement large DFAs more efficiently and to target larger networks.

6.2 Background

6.2.1 The FINN Compiler

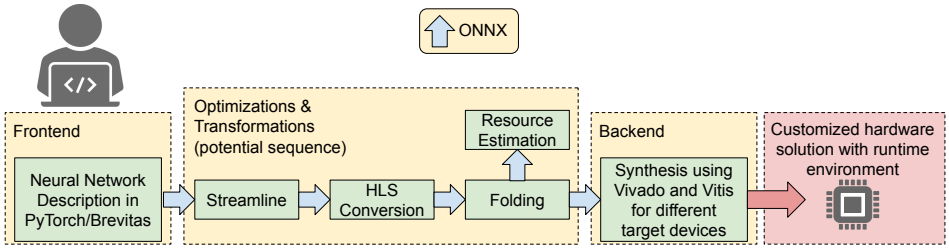


Figure 6.1: FINN Compiler Flow. Source [83]

From PyTorch to FPGA bitstream To create the DFAs the FINN compiler was used. Fig. 6.1 illustrates the stages of FINN flow to reach a DFA FPGA implementation. The process starts with a PyTorch description using the Brevitas library for quantization-aware training [116]. The trained network is exported to the ONNX format, which the FINN compiler uses as the intermediate representation. This graph is first optimized in the *streamline* phase, for example merging operations, minimizing operator size, and removing floating-point computations without changing functionality, and then it is translated to hardware layers. Most of these layers are implemented with HLS using the FINN hlslib [117], but RTL designs may also be employed. Based on the user's

Table 6.1: Neural Networks accelerated in this work. Source [83]

Topology	Precision*	Model Size (MB)	% Top-1 Accu.	GOps	FINN Nodes
ResNet-50 (RN-50)	W1A2	11.25	67.27	6.8	277
MobileNetV1 (MN)	W4A4	2.1	70.39	1.1	115

* *WxAy indicates quantization: x-bit weights, y-bit activations.*

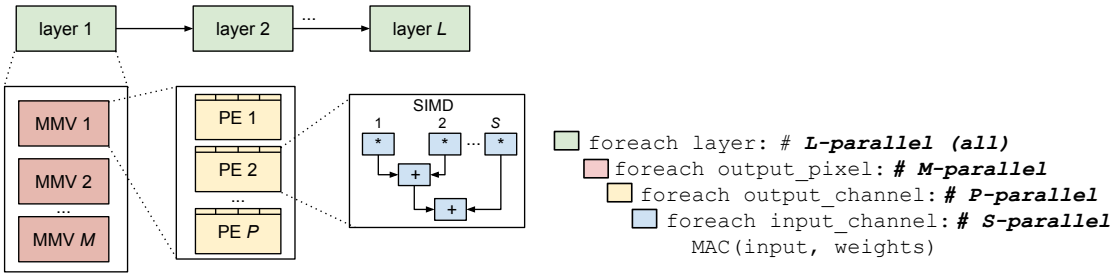
target throughput, the parameters of the hardware layers are tuned to adjust the level of parallelism in a process called *folding* in the FINN framework. In addition, these hardware modules often allow customizations like whether to use DSPs or LUTs for operations, or using LUTRAM, BRAM, or URAM for storage. Finally, the modules are connected either as IPs in Vivado IP integrator or as kernels connected using Vitis, followed by the RTL synthesis and P&R stages to create the FPGA configuration file.

ImageNet Classification support As part of this project, support within the FINN compiler for MobileNetV1 (MN) [118] and ResNet-50-v1.5 (RN-50) [119] was added. Quantized versions of these networks were described and trained using Brevitas, resulting in the CNNs summarized by table 6.1 and available from [120]. We consider these CNNs relevant because of their widespread use in modern AI applications. Additionally, these large networks, the largest implemented by the FINN compiler to date of development, allow us to explore the designed challenges previously stated. Finally, notice that unlike MN, RN-50 has a non-linear topology, adding some additional complexity to its graph manipulation and implementation.

6.2.2 Scaling Up CCN Performance

6.2.2.1 Adjusting the Accelerator Parallelism

CNNs are highly parallelizable, as there are normally several dimensions of independent operations. Unlike MPEs, which time-multiplex layers, DFA execute them all in parallel. Given its pipeline architecture, the throughput of the DFA is determined by the throughput of the slowest layer, and thus it is important to balance them not to waste resources. To achieve this, a DFA tunes the level of parallelism of the other dimensions, illustrated in fig. 6.2 for dense convolutions as supported by FINN. Each MMV compute an output pixel, each PE (MMV component) compute different channels of a given pixel, and each PE performs SIMD multiplications in parallel. As previously mentioned, al-



((a)) Hardware structure. ((b)) Pseudocode view.

Figure 6.2: FINN parallelism dimensions for dense convolutions. Source [83]

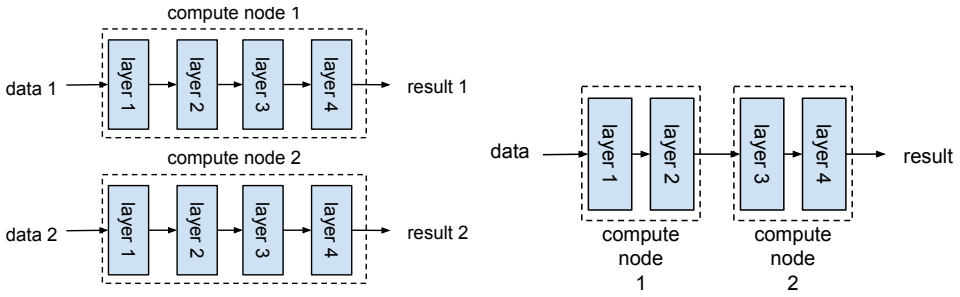
though compute resources may be scaled, memory requirements for parameters stay relatively constant, and thus limiting the CNNs implementable in a given FPGA.

6.2.2.2 Scaling Out

When resources within a chip do not allow us to scale-up (or even to implement) a DFA, we may distribute the compute load among different chips. Although there are several ways of doing this [121], here we consider the two basic alternatives shown in fig. 6.3, data- and model-parallelism.

Data-parallelism simply replicates the single node solution on several other nodes, requiring a server to distribute the inputs among the nodes, which execute a whole instance of the CNN, and then, the server gathers all the outputs. From the hardware design point of view, this option is simple, as the same configuration is used in all nodes, and for software, it is also convenient as the interface is always the same. If n nodes are used, this solution increases throughput by n, keeping latency relatively constant.

Conversely, model-parallelism distributes a single CNN instance among different nodes, allowing to use scaled-up versions of the DFA. The system feeds the accelerator



((a)) Data parallelism. ((b)) Model parallelism.

Figure 6.3: Two types of parallelism to scale out systems. Source [83]

through a single input, intermediate results are transmitted between nodes, and outputs are gathered from a single interface. Despite implying a more complex hardware design process, the scaled-up DFAs allowed by this alternative reduce latency and distribute weight parameters among different nodes, relaxing the memory requirements, and thus allowing larger nets to be implemented. Depending on how it is implemented, this option may also require adapting the software that controls the accelerators as the interface may change (e.g., providing inputs through one card and getting outputs through another). Although, performance improvements are harder to estimate in this case, which eventually saturate, for n nodes (a few), throughput may approx. increase by n and latency be reduced by n.

6.3 Challenges in Scaling Up Dataflow Architecture Performance

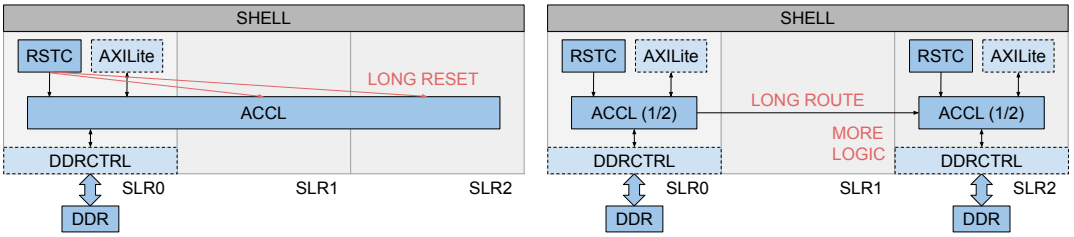
To maximize the implementation efficiency, we aim to maximize operating frequency, while also maximizing parallelism. In this section, we describe the main obstacles found when striving to those goals.

6.3.1 Considerations for Large FPGA Designs

The larger high-performance Xilinx FPGAs are composed of multiple silicon dies (called super logic regions, SLRs), mainly to optimize production cost, as yield[1] decreases fast with the chip size [122]. Multiple connections between SLRs exist, but these are more limited and slower compared to the routing within an SLR. Therefore, special attention has to be paid to these SLR crossings to achieve a high frequency. In particular, we highlight the following:

- **Critical path determined by high fan-out control signals:** it is well known that high fan-out nets can limit clock frequency in synchronous digital circuits [123]. This matter is worsened when these go through SLR crossings, given the large delay they suffer. Fig. 6.4(a) illustrates a monolithic IP accelerator that requires multiple SLRs to be implemented, and as a consequence, the reset nets need to go through one or more SLR crossing, which in our implementations,

[1] fraction of dies produced without defects



((a)) Frequency penalties due to long resets ((b)) Frequency and area penalties due to bad floorplaning

Figure 6.4: Bad design decisions for large accelerators on multi-SLR FPGAs. Source [83]

limits the maximum frequency. Therefore, this matter needs to be considered when implementing large DFA in these devices.

- **Data SLR crossings:** Not only long fan-out needs may be affected by SLR crossings. In general, to avoid frequency penalties, it is good practice to limit the width of the buses crossing SLRs and to optimize these crossings using the registers available for this purpose [124].
- **Logic placement:** Fig. 6.4(b), illustrates potential pitfalls when placing accelerator modules. To address the previously mentioned problems, we may partition the accelerator in several modules to allocate them in different SLRs. When doing so, it is important to avoid nets to go through multiple SLR crossings (or pipeline them appropriately). Additionally, to avoid redundant logic, we should try to place in the same SLR the modules that require access to a particular resource, like external memory.

6.3.2 Maximizing Compute Density

6.3.2.1 Resource Balancing

As opposed to ASICs, which only implement what has been designed (determining the silicon area), FPGAs chips have a fixed amount of any given resource. Then, a heavily used resource degrade the utilization efficiency of the device. However, in FPGAs, a given functionality may be implemented with different resources. For example, multiplications can be implemented with LUTs, DSPs, BRAMs (using them as look-up tables) or a mix of them. Employing only DSPs for multiplication may limit DFA throughput, even when LUTs are under-utilized. Therefore, through resource balancing, increased parallelism can be achieved.

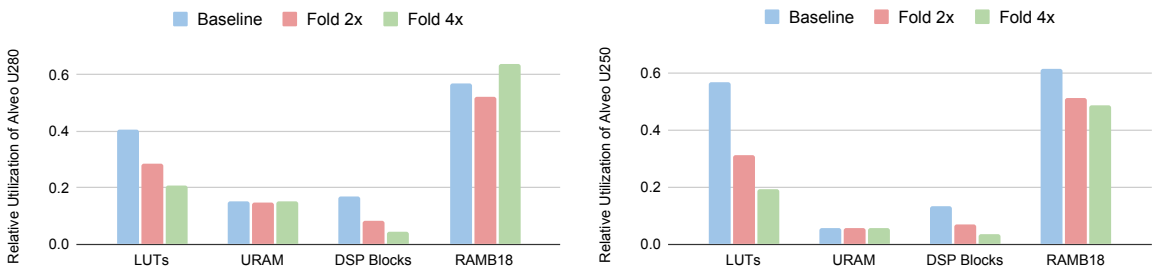
Table 6.2: Vivado resource utilization guidelines to avoid frequency penalties

Resource	LUT	FF	BRAM	URAM	DSP	Avg. of DSP, BRAM, URAM
Guideline (%)	70	50	80	80	80	70

In addition, over-utilization of a given resource can lead to frequency penalties, given that it hinders an appropriate placement and routing. Thus, balancing resources also allows maximizing the operating frequency. Although some designs may allow higher utilization rates while still accomplishing high frequencies, Xilinx recommends not surpassing the rates shown in table 6.2 to avoid frequency penalties. In the past, the resource balancing and the partitioning processes have been performed separately [125], doing the balancing as a fine-tuning step, but we notice that these should be optimized together to have a significant effect on performance given their interdependency. In general, partitioning would avoid surpassing utilization rates that cause frequency penalties. Given that reducing the utilization of a resource below these limits has little impact on frequency (confirmed by our experiments with DFAs), the usefulness of performing resource balancing after partitioning is limited. Also, in this way, potential reduction in the number of required SLRs would not be exploited.

6.3.2.2 Resource Efficiency as a Function of Accelerator Scale

Although the level of parallelism can be scaled, the performance per area metric is not constant for DFAs. Fig. 6.5 show the resource utilization rates required by MN and RN-50, using 3 different folding solutions: *Baseline*, *Fold 2x* (2× the II of Baseline), and *Fold 4x* (4× the II of Baseline). It is possible to observe that most resources are not scaled down by the same factor FPS are decreased. This is particularly true for memory resources, as previously noticed and thus, larger DFA are more resource efficient. Yet, this



((a)) MN Profile ((b)) RN-50 Profile

Figure 6.5: Resource utilization as a function of folding. Source [83]

may require more resources than what is available in the FPGA chip. To deal with this, we may resort to model-parallelism. However, and most importantly for DFAs which are custom-built implementations for a particular network, this accelerator partitioning task should be automated to enable an easier deployment and improve designer's productivity.

6.3.2.3 Folding Limitations

When scaling-up any accelerator, there exist limitations to granularity of the level of parallelism steps, and also we will eventually reach upper bounds on parallelism. These limitations are dictated by the algorithm itself, or the chosen architecture or framework. For example, FINN is able of a somewhat coarse-grained folding, as it forces that resources allocated for a given dimension must be an integer factor of the dimension cardinality (see section 6.2.2.1). For example, if a convolution has 64 output channels, FINN only allows 2^i PEs, where i is an integer.

These limitations will tend to avoid the system designed to achieve the desired level of device utilization. To deal with this issue, we may use data-parallelism to increase the computational density of a device, instantiating an additional DFA (with a possibly different folding). What is more, this idea can be generalized for multi-FPGA deployments, allowing the combination of unused logic of different devices to implement more and/or larger (thus efficient) DFAs. As a result, the average FPS/FPGA will increase.

6.4 Research & Development Goals

Addressing the design challenges outlined requires physical considerations in the accelerator design. Partitioning the accelerator for non-monolithic FPGAs, dividing an accelerator among different FPGAs, placing partitions, and balancing resources need to be addressed globally to optimize efficiency. Although the focus here is on large CNN dataflow accelerators, these challenges are, in general, not application specific, but result from designing large systems and/or aiming at maximizing the performance that can be extracted from a given FPGA chip. Yet, this task is NP-hard, and thus, it rapidly gets out of reach for humans, and eventually, also for classical computers. For this reason, in chapter 11 we present a tool to efficiently perform this optimization task automatically.

Part III

Addressing the Challenges

CHAPTER 7

LOCO-ANS IMAGE CODEC: ALGORITHM

In this chapter, we present enhancements to the JPEG-LS standard, aimed at improving its coding efficiency at a low computational overhead, particularly for hardware implementations. The main contribution is a low complexity and efficient coder, based on Tabled Asymmetric Numeral Systems (tANS), well suited for a wide range of entropy sources and with simple hardware implementation. This coder enables further optimizations, obtaining LOCO-ANS. Targeting photographic images, the codec achieves up to 1.6%, 6%, and 37.6% better compression compared to JPEG-LS, for error tolerances of 0, 1, and 10, respectively. Allowing an increase in the context size and image tiling, a 2.3% lower bandwidth is obtained for lossless compression, also improving near-lossless compression. The results also show that our proposal compares favorably against state-of-the-art codecs like JPEG-XL and WebP, particularly in near-lossless, where it achieves higher compression ratios with a faster coding speed.

This chapter is based on the work published in [20]: Alonso, T., Sutter, G., & López de Vergara, J. E. (2021). LOCO-ANS: An optimization of JPEG-LS using an efficient and low complexity coder based on ANS, in IEEE Access, vol. 9, pp. 106606- 106626, 2021, doi: 10.1109/ACCESS.2021.3100747.

7.1 Introduction

As presented in chapter 4, JPEG-LS standard has a significant optimization potential. In particular, we showed that that entropy encoder subsystem (composed of a GPO2 and a run coder) generated up to 9% larger compressed output streams than the minimum established by the statistical model. However, this subsystem imposed other restrictions on the codec, like how to quantize context gradients. When this limitation is lifted, greater compression is unlocked, increasing the coding inefficiency to up to 25%, which grow further if only considering photographic images (more relevant to the target applications).

To improve JPEG-LS, we first developed an efficient and low complexity adaptive coder for sources with a geometrical distribution, which uses Tabled Asymmetric Numeral Systems (tANS) as the underlying technology [47], with a complexity similar to a Huffman coder but with efficiencies that closely approach to the model's entropy [48, 49]. This coder is a key part of the adaptive system to code sources with a two-sided geometrical distribution, that is, to code the prediction residual.

JPEG-LS codec was adapted to work with the designed coder allowing a better com-

Table 7.1: Notation for LOCO-ANS description and analysis

Notation	Description
$NEAR$	Codec input that determines maximum absolute difference between the original pixel and the decoded one.
ϵ	Two-sided geometrically distributed variable (prediction error)
z	Geometrically distributed component of the prediction error
y	Bernoulli distributed component of the prediction error
s	Fractional bias of the Two-sided geometric distribution
θ	Shape parameter of geometric and Two-sided geometric distribution
p	Bernoulli distribution parameter
$\hat{\theta}_q$	Quantized estimation of θ
$\hat{p}_q$	Quantized estimation of p
Q_θ	θ quantization function
Q_p	p quantization function
C	Symbol alphabet cardinality of an ANS table, given a $\hat{\theta}_q$
S_t	Accumulator storing $\sum_{i=1}^{t} z_i$
$\overline{S_t}$	Mean of the z geometric variable ($\sum_{i=1}^{t} z_i/t$)
N_t	Accumulator storing $\sum_{i=1}^{t} y_i$
St_p	Number of fractional bits used by S_t
Nt_p	Number of fractional bits used by N_t
NI	Geometric coder maximum number of iterations

pression, particularly, for lower entropy distributions, more common in near-lossless operation. A key aspect of this integration resides in the distribution parameter estimation, where low complexity and more precise quantizers were studied and developed. The resulting LOCO-ANS codec is capable of diverse trade-offs between resources and compression and was evaluated in depth to understand the sources of inefficiencies and how these relate to the coder parameters, resulting in a methodology to tune them. And, what is more, from their conception, the proposed coder and modifications are amenable for hardware implementation. The system prototype plus auxiliary code to create tables and run experiments are open sourced to the community [126].

As a reference, table 7.1 contains the notation used in this chapter to describe and analyze LOCO-ANS.

7.2 LOCO-ANS Overview

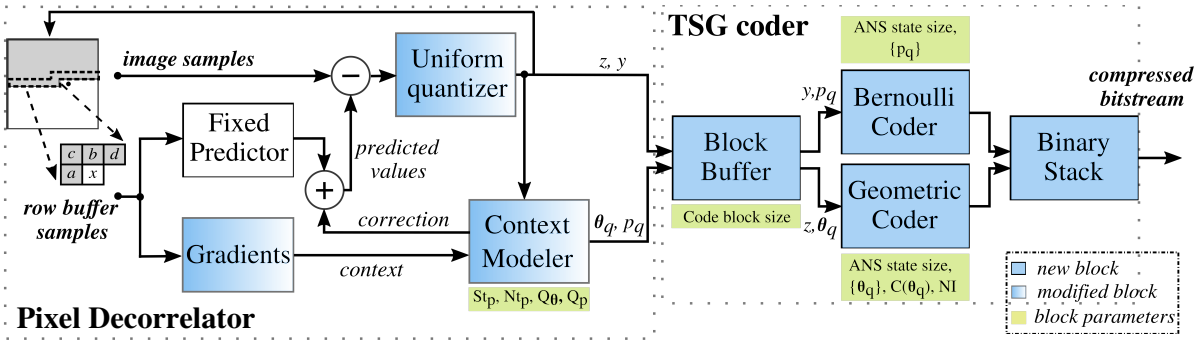


Figure 7.1: LOCO-ANS block diagram.

7.2.1 High-Level Description

Fig. 7.1 shows the LOCO-ANS algorithm block diagram, where two main subsystems can be appreciated, the Pixel Decorrelator and the TSG Coder, which substitutes the GPO2 and run coders used in JPEG-LS. The former processes the input pixels with the aim to turn them into a stream of statistically independent symbols along with their estimated distribution parameters, which the latter will code. These symbols are errors made by the adaptive predictor, which are then quantized according to the error tolerance ($NEAR$ parameter) as shown by equation 7.1. This quantization ensures that the absolute difference between the original value of a pixel and the decoded one is less

or equal to $NEAR$. Note that if $NEAR = 0$, then lossless compression is obtained. Other reversible operations are then applied to ϵ_q to improve compression.

$$\epsilon_q = round(\epsilon/(2 * NEAR + 1)) \tag{7.1}$$

The adaptive predictor is composed of a fixed predictor plus an adaptive bias correction. The adaptive correction is computed for each context, which is a function of the gradients surrounding the pixel currently processed. To obtain the context, gradients are first quantized, and given the results obtained in chapter 4, gradient quantization was modified to use a fixed quantizer function (thresholds computed using $NEAR = 0$).

Prediction errors are modeled using the Two-Sided Geometric (TSG) distribution, that is, an error ϵ_q is assumed to have the following probabilities:

$$P(\theta, s)(\epsilon_q) = C(\theta, s) \cdot \theta^{|\epsilon_q - s|}, \epsilon_q = 0, \pm 1, \pm 2, ..., \tag{7.2}$$

where θ and s are the distribution parameters and $C(\theta, s) = (1-\theta)/(\theta^{1+s} + \theta^{-s})$ is a normalization factor. As in the standard extension, the adaptive predictor is configured so that the fractional bias, s, tends to stay in $(-0.5, 0.5]$. However, to simplify the modeling and coding of this error, the alternative TSG model is used:

$$y = y(\epsilon_q) \triangleq \begin{cases} 0, \epsilon_q \geq 0 \\ 1, \epsilon_q < 0 \end{cases} \sim Bernoulli(p) \tag{7.3}$$

and

$$z = z(\epsilon_q) \triangleq |\epsilon_q| - y(\epsilon_q) \sim Geometric(\theta) \tag{7.4}$$

where $p = (\theta^{1+s})/(\theta^{1-|s|} + \theta^{|s|})$ and θ is the same parameter as in eq. 7.2 [62]. These distribution parameters are estimated by the Context Modeler for each context, generating the estimated quantized versions, $\hat{\theta}_q$ and $\hat{p}_q$.

As seen in the block diagram, the TSG coder uses two different coders to handle y and z, both based on tANS. As mentioned, ANS output bitstream acts as a LIFO, but the decoder needs to obtain the errors in the same order the decorrelator processed them, to be able to mimic the model adaptations. For this reason, the Block Buffer groups symbols in blocks and inverts their order. The output bits of a block are packed in the Binary Stack and stored in the inverse order, so the decoder can recover pixels in the same order the encoder processed them without additional metadata.

7.2.2 Encoding Algorithm Summary

The encoder algorithm can be summarized as follows, where (*) denotes a modified or new procedure with respect to JPEG-LS and (†) one taken from the standard extension.

Scanning the image (with an $ibits$ pixel depth) sequentially from left to right and from top to bottom:

1. Read the first pixel and store it directly, also updating the row buffer (*).
2. Read a new pixel.
3. Compute the gradients, quantize them (*) and obtain the pixel context.
4. Compute the fixed prediction.
5. Get the prediction bias and the TSG quantized parameters estimations, $\hat{p}_q$ and $\hat{\theta}_q$ for the context (*).
6. Correct the prediction using the bias and compute the prediction error.
7. Invert the sign of the prediction error if either the context is negative or $s > 0$ (†).
8. Obtain the quantized error using the NEAR parameter and reduce it modulo α, where $\alpha = 2^{ibits}$ if $NEAR = 0$ else $\alpha = (2^{ibits} - 1 + 2 * NEAR)/(2 * NEAR + 1)$.
9. Compute z and y and store it in the coder input buffer with their distribution parameters (*).
10. Check if the symbol block is complete, and if so, use the coder presented in section 7.3 to process the whole block and append the resulting binary stack to output bit stream (*).
11. Reconstruct the pixel and store it in the row buffer.
12. Update the prediction bias (†) and the TSG parameters estimations (*).
13. If there are more pixels in the image, return to step 2.

Although presented as an ordered list, notice that some of these steps can be done completely or partially in parallel.

7.3 An ANS-based Coder for TSG Sources

To use tANS in an adaptive setting, in general, one table per symbol distribution is required, so there is a trade-off between table resources and KLD. Additionally, more tables can also imply a reduction in the coder throughput.

To simplify the parameter estimation procedures and the coding efficiency of ANS, the proposed system encodes the (y,z) tuple components separately, instead of the ϵ TSG distributed error. Notice that choosing to code the (y,z) tuple components independently, allows having tables tuned to the distributions of each component instead of tuned to the tuple join distributions, which is needed if using the TSG model described by eq. 7.2. In this way, the number of required tables is equal to $|\{\hat{\theta}_q\}| + |\{\hat{p}_q\}|$ instead of $|\{\hat{\theta}_q\}| \cdot |\{\hat{p}_q\}|$ tables, where $|\{\hat{\theta}_q\}|$ and $|\{\hat{p}_q\}|$ are the number of reconstruction values supported for θ and p, respectively.

In this section, the codification procedures for y and z variables are presented. For a more clear explanation and given that most algorithms do not strictly depend on ANS, it is first assumed that the encode and decode order are the same (which is not true for ANS), addressing the codification order required by ANS later on.

7.3.1 Adaptive Bernoulli Coder

Coding the y binary variable with tANS is simple. Given a quantized estimation of the Bernoulli parameter $\hat{p}_q = Q_p(\hat{p})$, where Q_p is the chosen quantization function for the p parameter, a unique index is assigned to it, which is used to select the ANS table tuned to $\hat{p}_q$. To half the number of required tables, if $\hat{p}_q > 0.5$, then y is inverted and $\hat{p}_q$ is set to $1-\hat{p}_q$. On the decoder side, when $\hat{p}_q > 0.5$, then $\hat{p}_q \leftarrow 1-\hat{p}_q$ is used to select the decodification table, and the obtained symbol is inverted. Note that assuming the TSG distribution hypothesis holds and that the bias cancellation procedure works well, and given that $p = (\theta^{1+s})/(\theta^{1-|s|} + \theta^{|s|})$ and $s \in (-0.5, 0.5]$ then $p \in [\theta/2, 0.5]$. In practice, using the Rawzor dataset, limiting the $\hat{p}$ to that range does not increase the bpp, except for the "zone plate" artificial image.

7.3.2 Basic Geometric Coder

Given a symbol z coming from an infinite alphabet source with a geometric distribution and a quantized parameter estimation $\hat{\theta}_q = Q_\theta(\hat{\theta})$, where Q_θ is the chosen quantization for the θ parameter, the probabilities of z are computed as:

$$P_(\hat{\theta}_q)(z) = (1-\hat{\theta}_q) \cdot \hat{\theta_q}^z \tag{7.5}$$

To code this type of symbol source using tANS, one main challenge had to be overcome. Taking into account the maximum possible value of z for the image compressing application, the cardinality of the symbol source is large, which leads to high resource

requirements for the tANS tables. Additionally, eq. 7.5 shows that the probabilities of z can decrease very fast. So, as seen in section 7.5.1.2, ANS would require an impossibly large state to cover the whole z range, which, in turn, exponentially increases the memory requirements. This could be addressed with binarization, but there is an alternative enabled by the memoryless property of geometric distribution, which allows a simpler, scalable and generally higher throughput system.

Both the large cardinality and high probability precision problems can be addressed by using conditional probabilities when the symbol z is larger than an implementation defined threshold. Symbols in the range $[0..(C-1)]$ are coded directly, choosing the ANS mode (ANS table computed for a certain distribution) according to the provided $\hat{\theta}_q$, which also determines the C constant. For larger symbols, the coder inserts C, which stands for "$z \geq C$". Applying the memoryless property, it can be seen that the distribution of $(z-C)$ given that $z \geq C$ is the same as z. For this to be strictly true, z should come from an infinite set not a constrained one, as in the case of error residuals, but the set is large enough, so there is no significant difference, at least, for the $\hat{\theta}$ seen in practice. Then, using the same ANS mode (as they have the same distribution), the system tries to code $(z-C)$ and, again, if it is greater or equal to C, it inserts C. This process is repeated until a symbol different that C is coded.

Notice that in this way, without the need of deriving probabilities for the decomposed symbols or any additional statistics gathering process, and using a stateless coder with $C+1$ symbols, any number originated from an infinite alphabet source with the memoryless property can be optimally encoded. In this way, for each supported $\hat{\theta}_q$, just one tANS table tuned to a $C+1$ symbol source is required.

7.3.3 Codification Order for ANS

If ANS is used to code the symbols, then for the bitstream to be decodable, the codification order must be inverted.

7.3.3.1 Symbol Block Codification Order

As mentioned before, the ANS output binary acts as a Last In, First Out (LIFO) memory, so prior to coding, the symbols are stored with the necessary adaptation parameters and coded in reverse order, as proposed in [48]. In this case, $\hat{\theta}_q$ and $\hat{p}_q$ parameters should be stored alongside the (y,z) tuple. In some cases, it is not possible or desirable (added latency) to store these variables for the whole image, so smaller blocks can be used at

the cost of some additional bits (the final ANS state needs to be sent after each block and small inefficiencies can arise due to word alignment) effecting slightly the overall coding efficiency. The decoder needs to know the block size, which can be included in the compressed image header.

In general, the additional bits per symbol due to the need of transmitting the final ANS state at the end of each block and the requirement of aligning a new block to a certain word size will be, on average:

$$\overline{KLD} = (state\_bits + (word\_bits - 1)/2)/block\_size \tag{7.6}$$

For example, for a 6 bit ANS state, aligning binary blocks to bytes and using a block size of 2048 pixels, $\overline{KLD} = (6 + (8-1)/2)/2048 \ = 0.0046\, bits/pixel$.

As suggested in [48], the initial state of the ANS coder can be used to carry information, but, in the system prototypes, the initial ANS state is used as a sanity check of each block. That is, the encoder always sets the initial state to 0 (actually to 2^{state_bits}), and the decoder checks after each block that the final ANS state is 0 (corresponding to the first in the encoder side).

In hardware implementations, to avoid stalls, a ping-pong buffer should be used. In this manner, a block can be processed, while the next one is being generated.

7.3.3.2 Sub-Symbol Codification Order

If y is coded before z, then z is decoded before y. Additionally, for each symbol z, the order of operations described in section 7.3.2 is as the decoder would see them. The encoder should proceed in the reverse order, inserting first the last sub-symbol the decoder should see. It is not hard to see that the value of that sub-symbol is $z \mod C$ (trivial to implement if C is chosen to be a power of 2). After, if required, it inserts a sequence of n C sub-symbols, where $n = (z - (z \mod C))/C$.

Finally, the codification of a single z symbol could be implemented as seen in alg. 7.1. There, *store_in_binary_stack* function call deals directly with the output binary and its arguments are an integer variable with the bits to store and the number of bits to take starting from the least significant bits. The ANS tables are stored in the array ANS_table, which is addressed by the quantized distribution parameter ID, the current state of the ANS coder and the new symbol to encode. Each element of the table is a structure with the number of bits that should be sent to the output and the next ANS state. Note that ANS operation can be implemented differently [49, 57].

Algorithm 7.1: Codification procedure of a geometrically distributed symbol using tANS.

Require: z
Require: $param$
1: $c \leftarrow get\_cardinality(param)$
2: $remaining\_sym \leftarrow z$
3: $subsym \leftarrow z \mod c$
4: **repeat**
5: $remaining\_sym \leftarrow remaining\_sym - subsym$
// tANS coding
6: $obits \leftarrow ANS\_table[param][state][subsym].bits$
7: store_in_binary_stack($state, obits$)
8: $state \leftarrow ANS\_table[param][state][subsym].nx\_st$
9: $subsym \leftarrow c$
10: **until** $remaining\_sym = 0$

7.3.3.3 Binary Store Order

As the decoder reads the binary bits in the inverse order, the encoder generates them and to avoid the need of appending a header to each binary block, bits should be stored in the reverse order as they are produced. This can be easily implemented storing coder output bits in a stack, and then copying the whole binary block below the previous binary block.

7.3.4 Geometric Coder Iterations

Although this algorithm may appear to be slow for its iterative nature, even with small $C \in [1,16]$ and for the θ observed in 8-bit images, it is not. This can be appreciated in fig. 7.2, where the expected iterations per symbol ($\bar{i}$) were plotted, which is computed as follows:

$$\bar{i} = \frac{1}{1-\theta^C} \tag{7.7}$$

Note that the equation can be approximated with $\bar{i} \approx \overline{S_t}/C + 1$ for high $\overline{S_t}/C$, where $\overline{S_t} = \sum z_i/t = \theta/(1-\theta)$. As a reference, fig. 7.3 shows the distribution of $\overline{S_t}$ for different values of $NEAR$ for the Rawzor dataset.

Notice that the larger $\overline{S_t}$, the smaller rate at which $P(z)$ decreases. Additionally, C can be a function of $\hat{\theta}_q$. So, in general, although $\bar{i}$ increases almost linearly with $\overline{S_t}$, the maximum C value for a given an ANS state size tends to increase with $\overline{S_t}$. Of course,

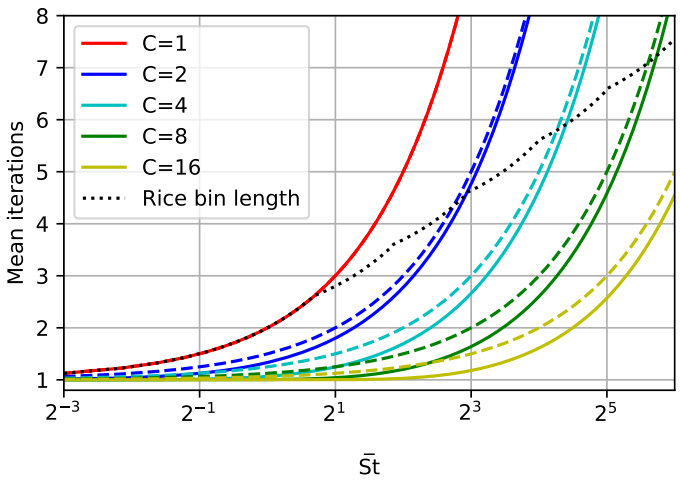


Figure 7.2: Geometric coder mean iterations as a function of $\overline{S_t}$ compared to number of iterations resulting from a Rice-based binarization strategy. The approximations are shown with dashed lines

the state size has to be large enough to be able to code θ and $(1-\theta)$ (the two symbols for $C=1$).

An alternative strategy to the one presented in alg. 7.1 would be to binarize the symbol and then proceed with a binary coder. Instead of a trivial binarization, this procedure could consist in using Rice-codes [127] for the symbol, which is a similar method to the one employed in the JPEG-LS standard extension. Fig. 7.2 allows comparing the iterations required by the proposed method with the average number of bits (and thus iterations of the binary coder) resulting from a Rice coding binarization strategy. There, the k rice parameter was chosen as the closest integer to $-log_2(-log_2(\theta))$. It can be seen

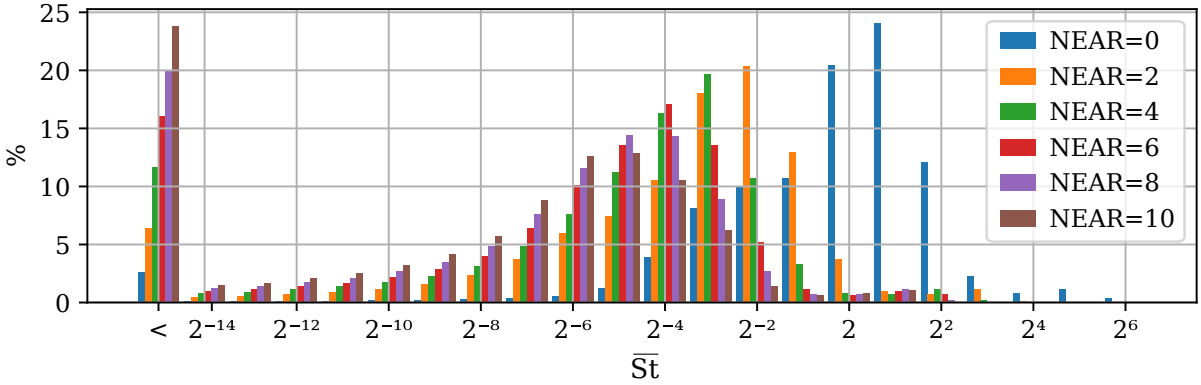


Figure 7.3: $\overline{S_t}$ Histogram for Rawzor dataset using different NEAR parameters. Bins bounds are placed at $(2^x, 2^{x+1})$ with 2^x representing the bin.

that in range of interest, with small values of C, the proposed method requires fewer iterations. Moreover, no binarization or bit probability modeling is required.

Although the coder can be configured so that $\bar{i}$ stays within some desired bounds, the maximum possible iterations are higher, which can lead to data loss if buffers are not correctly sized. Given that there are many situations in which buffer sizes and/or latency are highly constrained, this issue is addressed in the next section.

7.3.5 Limitation of Coder Iterations and Symbol Expansion

One of the concerns that arises when analyzing the proposed coding algorithm is the possibility of bursts of symbols requiring many cycles to code them and, particularly for the smaller θ, the possibility of local expansion. Both burst of long iterations and expansions have to be considered when sizing buffers before and after the coder, respectively. For this reason, it would be desirable to have a direct mechanism to limit them.

A way of limiting both the expansion and the number of iterations would be the following: The maximum number of iterations (NI) is chosen. Then, NI consecutive sub-symbols C will act as an escape mechanism, after which z is stored directly using $z\_bits = \lceil log_2(max(z)+1) \rceil$. Given the modulo reduction applied to the prediction error in JPEG-LS, z can be coded with $(ibits-1)$ bits, where $ibits$ is the pixel depth of the input image.

Alternatively, the residual $(z - NI \cdot C)$ could be stored, which in some configurations might require fewer bits to code. What is more, the GPO2 coder could be used to code this residual, selecting k from a small array indexed by the distribution parameter. These codes are more efficient for high θ geometric distributions, which at the same time are the most likely to require this mechanism for a given C.

For an ANS implementation of the coder, the order is reversed. If $z \geq NI \times C$, then z (or the residual) is coded and, after that, NI consecutive C sub-symbols are coded using the ANS encoder as usual. On the receiver side, upon seeing NI C sub-symbols, it will get out of the loop and proceed to get z directly (or through the Golomb decoder). Note that, as before, if C is chosen to be a power of 2, $NI \times C$ is just a binary shift. Alternatively, those values can be stored, and then, retrieved using the parameter distribution identifier. Finally, for the simpler case where the GPO2 coder is not used to code the residual, the algorithm can be expressed as in alg. 7.2, where a new input is required, z_bits.

Algorithm 7.2: Codification of single z limiting the iterations.

Require: z
Require: $param$
Require: z_bits
1: $c \leftarrow get\_cardinality(param)$
2: $remaining\_sym \leftarrow z$
3: $subsym \leftarrow mod(z,c)$
4: **if** $z \geq NI * c$ **then**
5: store_in_binary(z,z_bits)
6: $remaining\_sym \leftarrow NI * c$
7: $subsym \leftarrow c$
8: **end if**
// code z or escape symbol with tANS
9: **repeat**
10: $remaining\_sym \leftarrow remaining\_sym - subsym$
// tANS coding
11: $obits \leftarrow ANS\_table[param][state][subsym].bits$
12: store_in_binary_stack($state$,$obits$)
13: $state \leftarrow ANS\_table[param][state][subsym].nx\_st$
14: $subsym \leftarrow c$
15: **until** $remaining\_sym = 0$

7.3.5.1 Implications on Coding Efficiency

The implementation of this iteration limitation mechanism will tend to decrease the coding efficiency of the coder. In its simpler version, it forces all symbols equal or above $NI \times C$ to be coded using a fixed number of bits (z_bits). The KLD can be obtained as:

$$KLD(L,\theta) = \theta^L \cdot \left(z\_bits - Entropy(z|z \leq max(residual))\right) \tag{7.8}$$

where $L = NI \cdot C$.

The code inefficiency ($KLD/Entropy$) due to the use of the simple coding of z or its residual can be observed in fig. 7.4, setting NI to 7 and using small values of C, for $\overline{S_t}$ in the range observed in dataset. As it can be seen, KLD can be relatively small even for the simplest codification and using small numbers of C.

7.3.5.2 Upper Bound on the Code Length

As mentioned before, apart from limiting the iterations, it would be useful to obtain an upper limit on the code length. For each $\hat{\theta}_q$, if $Max(z) \geq NI \times C$, there are two symbols

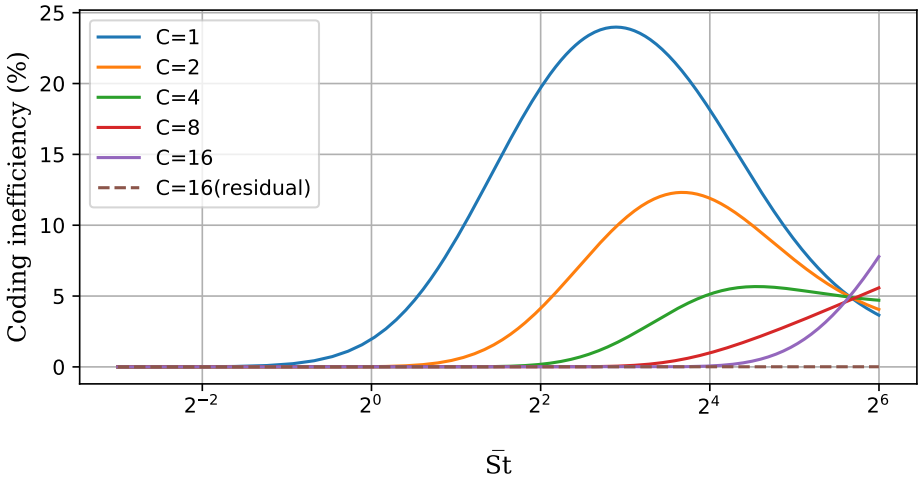


Figure 7.4: Coding inefficiency ($KLD/Entropy$) caused by the iteration limitation mechanism using the direct z codification after the escape mechanism. Curves for $NI = 7$

that could have the maximum code length. These are either $z \geq NI \times C$ (all symbols in this set are coded with the same length) or $z = NI \times C - 1$. Then, if $NI > 0$, an upper limit for a single symbol and a given $\hat{\theta}_q$ would be:

$$\begin{aligned} Max\,code\,length &\leq \max_{state}(tANS_{\hat{\theta}_q}[C]) \cdot (NI-1) + \\ \max\left(\max_{state}(tANS_{\hat{\theta}_q}[C]) + z\_bits, \max_{state}(tANS_{\hat{\theta}_q}[C-1])\right) & \\ &\leq ANS\_state\_bits \cdot NI + z\_bits \end{aligned} \tag{7.9}$$

Here, $tANS_{\hat{\theta}_q}$ is the ANS table for $\hat{\theta}_q$ storing the number of bits to send to the output, which is addressed by the symbol and ANS state (omitted in the equation). This is an upper bound as, after an ANS symbol is coded, only a subset of the state domain is possible. To have the exact maximum code length for a given $\hat{\theta}_q$, the sub-symbol sequences used in eq. 7.9 can be coded, iterating over the state domain to set the initial state.

However, upper bounds on long sequences of symbols (like the coder block size) are more useful to size the output buffer. Then, if the ANS tables are already generated, using a simulation, the buffer could be computed such that there is no possibility of exceeding its size. A complete block of the symbol that produces the maximum code length should be coded for each $\hat{\theta}_q$, taking the largest binary block to size the output buffer. Compared to eq. 7.9, this procedure would produce tighter upper bounds.

If the ANS tables are not yet generated, the worst case could be assumed, in which all

the max functions applied over $table_{ANS}$ in eq. 7.9 are equal to the ANS state number of bits. This would be accurate if the entropy of the C sub-symbol, is close to the state bits. If that is not the case, entropy values plus safety margins (see section 4.3) could be used instead of the $\max_{state}(tANS_{\hat{\theta}_q}[\,][\,])$ functions to do the estimation.

7.3.5.3 Interactions Between the Coder and the Rest of the System

Something that should be noticed is that there is a negative feedback loop in place. Code expansion and long iterations are due to a large z given the estimated $\hat{\theta}_q$ for the context and $C = f(\hat{\theta}_q)$. If this situation persists, the large errors are going to drive $\hat{\theta}_q$ up, and subsequent large z that belong to this context would produce fewer bits, and fewer iterations as, in general, C can be increased with $\hat{\theta}_q$ for a given ANS state size. If C is not increased, as it might be limited for resource requirement reasons, the number of iterations would remain the same, but maximum code length will tend to decrease with increasing $\hat{\theta}_q$ estimations, as the number of bits used to code the sub-symbol C decreases.

For this reason, the actual largest binary output depends on the relative values of the block size, context domain size (number of context defined by surrounding quantized gradients), context $\hat{\theta}_q$ parameter estimation inertia, NI, C for each $\hat{\theta}_q$. The smaller the block size, the bigger context domain size, and the higher $\hat{\theta}_q$ parameter estimation inertia, the closer it gets to the limit established by eq. 7.9.

7.4 Distribution Parameters Estimation

To integrate the presented coder with JPEG-LS, the parameter estimation procedures to obtain $\hat{\theta}_q$ and $\hat{p}_q$ need to be introduced. These procedures not only estimate the distribution parameters, but also define the Q_p and Q_θ parameter quantization functions.

7.4.1 p Parameter Estimation

An approximation of eq. 4.5 can be used to obtain $\hat{p}_q$. For this, the Nt sum is kept for each context and the bias cancellation procedure with some minor modifications can be employed, implementing a quantizer with uniform bin sizes. The reconstruction values can be chosen to minimize the KLD within each bin.

In alg. 7.3, N is the context counter also used for the bias cancellation and p_{id} is $\hat{p}_q$ ID number, which is also kept for each context. The parameters of this algorithm are

Algorithm 7.3: Estimation procedure of the p Bernoulli parameter for a given context.

Require: y, N_t, N, p_{id}
1: $N \leftarrow N + 1$
2: $N_t \leftarrow N_t - p_{id} + (y << Nt_p)$
3: $Li \leftarrow fi(N)$
4: $Ls \leftarrow fs(N)$
5: **if** $(Li > N_t)$ **then**
6: $\quad p_{id} \leftarrow p_{id} - 1$
7: $\quad N_t \leftarrow N_t + N$
8: **else if** $(N_t >= Ls)$ **then**
9: $\quad p_{id} \leftarrow p_{id} + 1$
10: $\quad N_t \leftarrow N_t - N$
11: **end if**

Nt_p and the bound functions, $fi(N)$ and $fs(N)$. Nt_p determines the number of fractional bits stored in the N_t register and, as a consequence, the size of each quantization bin is 2^{-Nt_p} and $L_s - L_i = fs(N) - fi(N) = N$.

There are several ways (L_i, L_s) can be set. The extreme cases are analyzed, that is, the case $(-N/2, N/2)$ (where $p_{id}/2^{Nt_p}$ is centered within bin bounds) and $(0, N)$ (where $p_{id}/2^{Nt_p}$ is not centered, but equal to the lower bound). Fig. 7.5 shows the KLD for each of these cases for $p \in [0, 0.5]$ and $Nt_p = 4$. Here, the reconstruction values were chosen to be in the center of the bin, except for bin 0 of the centered case, where the reconstruction is computed taking $p = 0$ as the lower bound.

Although slightly more complex, choosing the $(-N/2, N/2)$ bounds allows a lower KLD for a given precision as it has a smaller bin (half the size) in the lower end of the p range, where the KLD is more sensitive. This comes at the cost of an additional bin in the $p \in [0, 0.5]$ range (resulting in 2^{Nt_p-1}+1 bins). However, the optimal reconstruction value of the upper bin (bin 2^{Nt_p-1}) is $p = 0.5$, so the tANS coder can be bypassed as y does not need to be coded (entropy = 1). Then, the number of tables required for both configurations is 2^{Nt_p-1}. It has to be noticed that low $\hat{p}$ ranges are used for low θ, so the impact of choosing one or the other would be appreciated for lower entropy cases (high $NEAR$ and/or images that are accurately predicted, like smooth surfaces).

For photographic images, as forcing $\hat{p}_q <= 0.5$ does not increase the bpp, the condition $p_{id} < (2^{Nt_p-1} - 1)$ can be added to the `else if` (alg. 7.3), avoiding the need to implement the logic to code y for the rare cases when $\hat{p}_q > 0.5$ (as indicated in section 7.3.1).

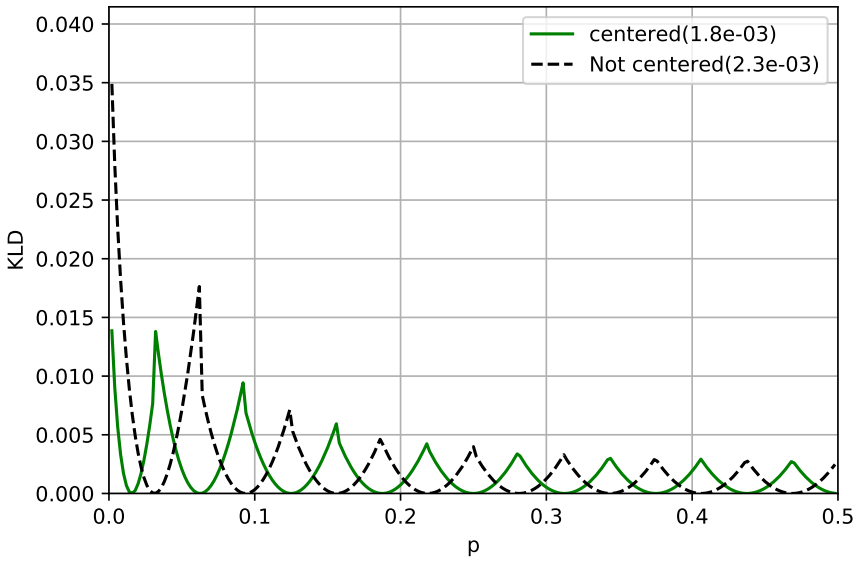


Figure 7.5: KL Divergence result of the quantization of the $\hat{p}$ estimated parameter for "centered" bin bounds $((id-1/2)/2^N t_p,(id+1/2)/2^N t_p)$ and "not centered" bounds $(id/2^{Nt_p},(id+1)/2^{Nt_p})$ using $Nt_p = 4$ (accumulator precision). The average KLD, in bits, are shown between parentheses

7.4.2 θ Parameter Estimation

Unlike the GPO2 coder used in LOCO-I, where the quantization of the TSG distribution parameters has to be adapted, particularly, to the k Rice parameter, with the proposed coder any quantization can be chosen. However, it is necessary to find a good trade-off between coding efficiency and coder resources.

An approximation of equation 4.6 is used to estimate θ. To implement it, a S_t sum register is stored for each context. Then, $\overline{St}$ needs to be computed and quantized, obtaining indexes which can be directly mapped to θ using eq. 4.6 with α and β set to 0. Given this direct relationship between $\overline{St}$ and θ, these two terms are used interchangeably.

7.4.2.1 Constant Ratio Quantizer

The *constant ratio quantizer*, CRQ, is defined here as having the lower and upper bounds of each bin computed as $(Li,Ls) = (\overline{St_x} * r, \overline{St_x}/r)$, where $\overline{St_x}$ is the reconstruction value of the bin x and $r \in (0,1)$ is a constant the regulates the size of the bins. This quantizer tends to keep the average KLD per bin constant when it is applied to $\overline{S_t}$. Once $\overline{St_0}$ and r are set, all bins bounds and reconstruction values can be determined. The bin bounds can be placed at $2^i, i \in \mathbb{Z}$, to obtain the quantization function used in LOCO-I for

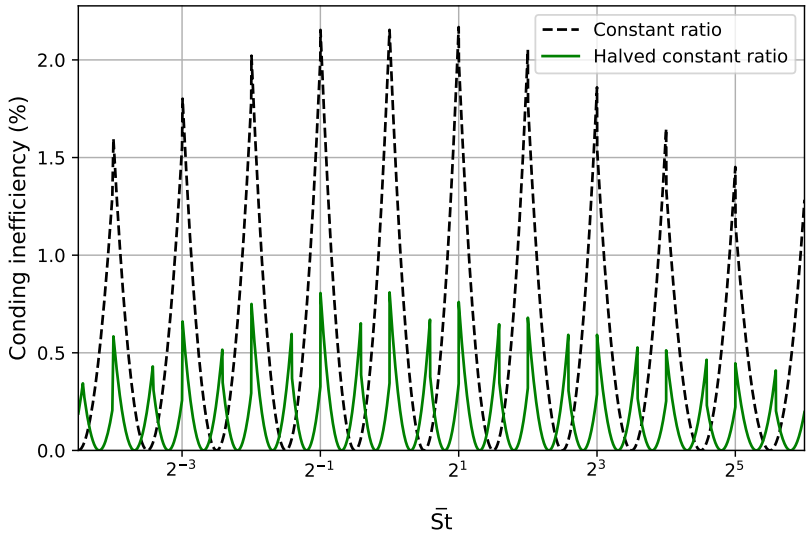


Figure 7.6: Coding inefficiency due to the quantization of $\overline{S_t}$ for two simple quantizers

the average absolute error. However, as the presented coder can handle lower entropy distributions, the precision of the St register can be increased (as it was done for the N_t register) to support the quantization of $\overline{S_t} < 1$, which has more impact as $NEAR$ increases.

The inefficiency ($KLD/Entropy$) due to the quantization of $\overline{S_t}$ can be observed in fig. 7.6, where θ is computed as $\overline{S_t}/(\overline{S_t}+1)$ and reconstruction values are computed using the rule stated above. Although those are not the optimal reconstruction values, they are close to them. Assuming a uniform distribution of $\overline{S_t}$ in the observed range, this simple quantization has on average inefficiency of 0.48%.

The division and quantization procedures can be carried out in several ways. LOCO-I presents an iterative method, implemented with a one-line for loop. Alternatively, the procedure in alg. 7.3 can be adapted to accomplish the same quantization. For this, L_i is set to 0, $Li = N << f(\theta_{id}, St_p)$ (where St_p is the precision of the S_t register) and in line 2, instead of subtracting θ_{id}, the bin lower bound is computed based on θ_{id} and St_p.

These two procedures will not always output the same result, as the latter can only produce a decrement/increment of θ_{id} of 1 with respect to the previous ID (this can be extended at the cost of more logic). In addition, this method requires storing θ_{id} in the context, although the size of the S_t register will be small as it will contain only the division residual.

In a software implementation, particularly a single thread one, this procedure will tend to be faster compared to the iterative one. However, from the hardware perspective,

despite its iterative nature, the first alternative is appealing as it can be carried out outside the error quantization loop, and then the system throughput will tend to be higher. In this case, S_t and N are sent to the next stage where a possibly highly pipelined module obtains θ_{id}, while the context is being updated and a new sample is processed by the image quantizer. Whereas for the second alternative, the quantization procedure and update of the S_t index in the context needs to be completed to continue with next image sample.

7.4.2.2 Finer Grain Quantizers

If higher coding efficiency is required, maintaining a simple quantization logic, the previously obtained quantization bins can be uniformly divided. This can be implemented in several ways, for example, see alg. 7.4. This algorithm halves each bin, generating the inefficiency curve labeled "Half constant ratio" in fig. 7.6 when using optimal reconstruction values. It achieves an average inefficiency of 0.12% at the cost of doubling the number of required tables for a given $\overline{S_t}$ range.

Algorithm 7.4: Procedure to obtain the quantized estimation parameter of the geometric distribution, $\hat{\theta}_q$, using the halved constant ratio quantizer.

Require: S_t
Require: N
Ensure: θ_{id}
$S_t \leftarrow S_t + (z << St_p)$ // In the update phase
$\theta_{id} \leftarrow 0$
$l \leftarrow N$
while $S_t > l$ **do**
  $\theta_{id} \leftarrow \theta_{id} + 2$
  $l \leftarrow l << 1$
end while
if $S_t > l - ((l+2) >> 2)$ **then**
  $\theta_{id} \leftarrow \theta_{id} + 1$
end if

7.4.3 Resets

As done in JPEG-LS, the context count N and accumulators (in this case S_t and N_t) are halved when N reaches $N_0 = 2^i, i \in \mathbb{N}$ to limit the size of the registers and better adapt to changes in the context statistics.

7.5 Selection of Coder Parameters

Different scenarios might need different trade-offs between resources, code efficiency, throughput and latency, and requiring support for a variable set of $NEAR$ settings and types of images. However, it is not an easy task to establish the best configuration because of the strong coupling between the parameters of the coder.

These parameters are:

- The ANS state size, which sets limits to the ranges of possible $\hat{\theta}_q$ and $\hat{p}_q$ values, as well as, the maximum C for a given $\hat{\theta}_q$.
- The precision of the S_t accumulator, St_p, which sets a lower bound to the $\hat{\theta}_q$ values. If St_p is such that the lower bound it sets is equal or below the one set by the ANS state size, increasing St_p has almost not impact. The only effect it would have is that the accumulator will have an additional memory of past errors.
- The maximum $\hat{\theta}_q$ value.
- The Q_θ quantization function
- The precision of the N_t accumulator, Nt_p, which sets a lower bound on the $\hat{p}_q$ range. If only considering the centered uniform quantizer presented in section 7.4.1, Nt_p also determines the Q_p quantization function.
- The ANS table cardinality, C, for each $\hat{\theta}_q$.
- The code block size.
- The geometric coder maximum number of iterations, NI.

All system performance measures are effected by all or most of the above parameters.

7.5.1 ANS Tables Limitations and Generation

To select the coder parameters, it is necessary to keep in mind some considerations regarding tANS.

7.5.1.1 Algorithm to Create the ANS Tables

To create the ANS tables, a slightly modified version of the heuristic algorithm (mentioned in section 4.3) was used. The goal of this modification is to ensure that the resulting table is a valid one, given the cardinality of the symbol source. This is done detecting if the original algorithm fails to assign at least 1 state to each symbol and

then forcing it. In these cases, the KLD is expected to be higher than what eq. 4.1 states, given that the tuning of the table to the set of symbols probabilities would tend to be worst. These tables are referred to as *suboptimal tables*. Note that this table generation algorithm can be improved.

7.5.1.2 ANS State Size and Small Symbol Probabilities

In general, the tANS tables can be fine-tuned to obtain better results than the heuristic algorithm. However, there is a minimum symbol probability below which the table cannot be tuned to, and that minimum is achieved by the suboptimal tables, although, in general, these do not need to be suboptimal. In the case of 2-symbol sources, these tables are constructed by assigning to the higher probability symbol the first $2^{state\_bits} - 1$ states and the last state to the other symbol.

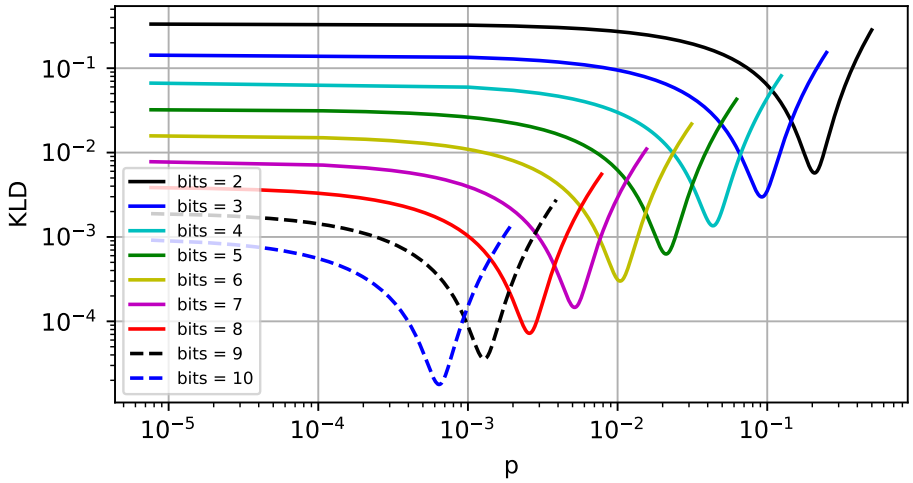


Figure 7.7: KL Divergence of 2-symbol tANS tuned to the minimum symbol probability as a function of the $P(0) = p$ probability for several ANS state bits.

Figure 7.7 shows the KLD achieved by 2-symbol suboptimal tables as a function of the p Bernoulli distribution parameter for different ANS state sizes. Notice that 2-symbol sources would minimize the KLD for a given state size (see eq. 4.1). From these curves, table 7.2 was obtained, which shows the minimum KLD achieved and at which p symbol probability.

This has to be considered when sizing the state size of the coder. For example, in an adaptive codification of a Bernoulli source, trying to tune tables to a p parameter equal or below the minimum probability observed in table 7.2 will always result in the

Table 7.2: Performance of 2-symbol tANS tuned to the minimum symbol probability as a function of ANS state bits.

Address size	min KLD	$argmin_p KLD$	min prob/(1/|S|)
2	5.7273e-03	2.0750e-01	0.8300
3	2.9764e-03	9.2050e-02	0.7364
4	1.3537e-03	4.3425e-02	0.6948
5	6.2724e-04	2.1112e-02	0.6756
6	2.9950e-04	1.0406e-02	0.6660
7	1.4602e-04	5.1631e-03	0.6609
8	7.2063e-05	2.5766e-03	0.6596
9	3.5799e-05	1.2883e-03	0.6596
10	1.7833e-05	6.4114e-04	0.6565

minimum entropy table. Then, adding more than one of these tables would not improve the coding efficiency, thus wasting resources.

7.5.2 Selection Methodology

7.5.2.1 Preliminary Considerations

A design methodology was derived from the mentioned relationships between coder parameters and the experience obtained when creating the prototype configurations for the experiments. For them, given an ANS state size, the main objective was to obtain configurations for a wide range of prediction error entropy. That is, to aim at a wide range of images and $NEAR$ values. Additionally, for each ANS state size, a good trade-off between code efficiency and resources was sought. Then, the methodology intends to support the widest range of $\hat{\theta}$ and $\hat{p}$ for a given state size.

In addition, Nt_p is set so that it does not limit the range of $\hat{p}$, but not increasing it beyond that point as the impact on efficiency tends to be minimal while the number of tables doubles for each additional bit of precision (if the quantizer is configured to obtain the maximum number of quantization bins given the selected precision). Also, by default, the constant ratio quantizer is used for $\overline{S_t}$ and the centered uniform quantizer for p.

The maximum $\hat{\theta}$ that has practical implications to code efficiency is affected by the minimum $NEAR$ supported, the pixel depth and the type of images to encode (classifying them according to their entropy, given the chosen model). Assuming that the actual z distribution as a geometric conditioned with the maximum possible value ($2^{ibits-1}-1$), then as θ tends to 1, then $\overline{S_t}$ will tend to ($2^{ibits-1}-1$)/2 (half of the range). Fig. 7.3 shows that for the 8-bit gray images of the dataset, some pixels reach this maximum (less than

0.4% of them). However, if only photographic images are considered, just 0.16% of the pixels reach a $\overline{S_t} > 16$. Then, to achieve high coding efficiency, the maximum $\hat{\theta}_q$ should correspond to a quantization bin that covers or is above $\overline{S_t} = 32$, in the general case, and $\overline{S_t} = 16$ for photographic images. If the minimum $NEAR > 0$, these values would be approximately scaled down by $\delta = 2 \cdot \min(NEAR) + 1$.

7.5.2.2 Methodology

The methodology is as follows:

1. Choose the ANS state bits.
2. Set St_p and Nt_p such that they do not increase the lower bound on the range of the distribution parameter estimations they affect ($\hat{\theta}$ and $\hat{p}$, resp.), given the selected ANS state size. For this, start with small precision, for example, set both St_p and Nt_p to 0. Then, using $C = 1$, try to generate the ANS tables for the first bin of each quantizer (smallest $\hat{\theta}_q$ and $\hat{p}_q$). If it succeeds, increment the corresponding accumulator precision. If it returns a suboptimal table, stop.
3. Set the maximum $\hat{\theta}_q$. Choose the minimum $\hat{\theta}_q$ between the maximum one that has practical implications to compression and the maximum supported by the ANS state size. To check this latter maximum, proceed similarly to step 2, iterating over the order set of $\hat{\theta}_q$ until a suboptimal table is returned. Particularly for hardware implementations where ANS tables would be stored in on-chip memory, using a number of $\overline{S_t}$ quantization bins different from a power of 2 will result in unused resources. Then, if the ANS state size allows it, increasing the number of $\overline{S_t}$ quantization bins up to a power of 2 might provide some additional compression without requiring more resources.
4. For each $\hat{\theta}_q$, choose the maximum allowed C. For this, proceed similarly to step 2, but in this case, start with $C = 1$ and iterate over power of 2. In the experiments, the maximum allowed C (8) was not big enough to have a significant intrinsic ANS KLD (see eq. 4.1). However, depending on the implementation, the maximum used C can significantly affect memory resources and, particularly in the case of hardware implementations, coder throughput. For this reason, an upper limit to C may be set using the ANS state bits and the number of required ANS tables to do resource and performance estimations.

Initially, to choose the ANS state bits, it can be assessed the number of bits that can be afforded given the memory resources and performance requirements. This should be done assuming that 16-64 ANS tables would be employed. For FPGA implementations, the results in [57] can be used as a guide to understand the impact of resources on performance.

7.5.2.3 Setting the Code Block Size and NI

The code block size is relatively decoupled from the rest of parameters. The larger it is set, the better compression ratio achieved. However, if the binary is aligned to bytes and the ANS state bits is below 10, no significant improvement will result increasing the code block size above the tens of thousands of symbols. Increasing its size comes at the cost of more memory resources and some impact on latency, although, in practical scenarios, this does not represent a major problem to achieve high efficiency and low latency. For most of the experiments, presented in section 7.6, the block size was set to 2048 as it results, on average, in a KLD of 0.005 bits or less when binary blocks are aligned to bytes (ANS state bits $\in$ [4..7]). Moreover, for an FPGA implementation processing 8-bit images and using 32 $\hat{p}_q$ tables and 32 $\hat{\theta}_q$ tables, one block of symbols with their distribution parameter estimations can be stored in 1 Xilinx 36K BRAM or 2 Intel M20K.

Regarding NI, a larger value tends to reduce the bpp. However, as in the case of the code block size, increasing NI has diminishing returns. Also, the worst-case throughput worsens linearly (initially) with NI. Most test were run with NI set to 7, the number of bits required to represent z variable for 8-bit images. Then, the worst-case throughput is the same as the resulting from a coder using trivial binarization. Using this value, little negative impact on compression is seen in general, and almost no impact for photographic images.

7.5.3 Tuning the Coder Parameters

If the application needs to support a limited number of $NEAR$ values and/or type of images, then better trade-offs between code efficiency and resources could be obtained.

Using the halved constant ratio quantizer for $\overline{S_t}$ (doubling the number of bins covering the same range) can have a greater impact on code efficiency than increasing the ANS state bits, while in former equal or less memory resources are required. This effect can be observed in the experimental results shown in the following section.

Similarly, for $\hat{p}$, changing the centered uniform quantizer with the non-centered one and incrementing Nt_p in one, would result in the same minimum $\hat{p}_q$, but the rest of the $\hat{p}$ range would have bins that are half the size, resulting in a smaller KLD.

If performance models and resource restrictions are available, the optimization task could be handled algorithmically.

7.6 Experimental Results

A prototype of LOCO-ANS was implemented using C++, which was tested with a set of different configurations with the goal of exploring the design space. The tested configurations can be seen in table 7.3. The name of the configurations indicates the most relevant parameter settings, following the format:

Nt{Nt_p}_St{Q_θ}{St_p}_ANS{State bits}

, where Q_θ is "cg" for the (coarse grain) constant ratio quantizer or "fg" for the (finer grain) halved constant ratio quantizer described in section 7.4.2. In all cases, N_0 was set to 64 (JPEG-LS default), the centered uniform quantizer was used to quantize the distribution parameter estimation $\hat{p}$ and the maximum C was set to 8.

Apart from the configuration parameters, table 7.3 shows the total number of rows (total number of tables by the number of states), which provides a measure of the memory resources required by each configuration (the actual memory utilization depends on the implementation).

Table 7.3: Prototype configurations used in the experiments

Conf. name	State bits	St_p	Max $\overline{S}_{tq}(\hat{\theta}_q)$	Q_θ	Nt_p	C range	# of θ tables	# of p tables	total # of rows
Nt4_Stcg5_ANS4	4	5	22.63 (0.958)	CG	4	1-8	11	8	304
Nt5_Stcg6_ANS5	5	6	90.51 (0.989)	CG	5	1-8	14	16	960
Nt5_Stfg6_ANS5	5	6	55.62 (0.982)	FG	5	1-8	26	16	1344
Nt6_Stcg7_ANS6	6	7	90.51 (0.989)	CG	6	1-8	15	32	3008
Nt6_Stcg8_ANS7	7	8	90.51 (0.989)	CG	6	1-8	16	32	6144
Nt6_Stfg8_ANS7	7	8	111.24 (0.991)	FG	6	1-8	32	32	8192

Table 7.4: Mean bpp and iterations using the prototype configurations with NI=7.

	Nt4_Stcg5_ANS4		Nt5_Stcg6_ANS5		Nt5_Stfg6_ANS5		Nt6_Stcg7_ANS6		Nt6_Stcg8_ANS7		Nt6_Stfg8_ANS7	
Error[1]	bpp [2]	$\bar{i}$	bpp [2]	$\bar{i}$	bpp [2]	$\bar{i}$	bpp [2]	$\bar{i}$	bpp [2]	$\bar{i}$	bpp [2]	$\bar{i}$
0	3.41 (-2.8)	1.3	3.31 (0.4)	1.3	3.3 (0.7)	1.3	3.29 (0.9)	1.3	3.29 (0.9)	1.3	3.28 (1.2)	1.3
1	2.04 (3.5)	1.2	2.03 (4.2)	1.2	2.02 (4.5)	1.1	2.02 (4.4)	1.1	2.02 (4.5)	1.1	2.01 (4.8)	1.1
2	1.55 (5.9)	1.1	1.53 (7)	1.1	1.53 (7.2)	1.1	1.53 (7.3)	1.1	1.53 (7.4)	1.1	1.52 (7.7)	1.1
3	1.28 (8.4)	1.1	1.26 (9.9)	1.1	1.26 (10.1)	1.1	1.25 (10.4)	1.1	1.25 (10.4)	1.1	1.25 (10.7)	1.0
4	1.1 (10.9)	1.1	1.07 (12.8)	1.1	1.07 (13)	1.1	1.07 (13.5)	1.0	1.06 (13.5)	1.0	1.06 (13.8)	1.0
5	0.96 (13.2)	1.1	0.94 (15.4)	1.1	0.94 (15.6)	1.1	0.93 (16.3)	1.0	0.93 (16.4)	1.0	0.92 (16.6)	1.0
6	0.86 (14.8)	1.1	0.83 (17.4)	1.1	0.83 (17.6)	1.1	0.82 (18.4)	1.0	0.82 (18.6)	1.0	0.82 (18.9)	1.0
7	0.77 (15.9)	1.0	0.75 (18.9)	1.0	0.74 (19.1)	1.0	0.73 (20.1)	1.0	0.73 (20.3)	1.0	0.73 (20.5)	1.0
8	0.7 (16.6)	1.0	0.67 (19.7)	1.0	0.67 (20)	1.0	0.66 (21.1)	1.0	0.66 (21.4)	1.0	0.66 (21.7)	1.0
9	0.65 (17.5)	1.0	0.62 (21.3)	1.0	0.62 (21.6)	1.0	0.61 (22.9)	1.0	0.6 (23.2)	1.0	0.6 (23.4)	1.0
10	0.6 (18.2)	1.0	0.57 (22.7)	1.0	0.56 (23.1)	1.0	0.55 (24.5)	1.0	0.55 (24.8)	1.0	0.55 (25)	1.0

The results presented here were obtained averaging the bpp computed for each image so that all images have the same weight in the average. Code block size set to 2048 symbols.

[1] By definition, the maximum peak error is equal to the NEAR input parameter

[2] The improvement percentage compared to JPEG-LS is shown between parentheses.

7.6.1 Analysis of LOCO-ANS Configurations Performance

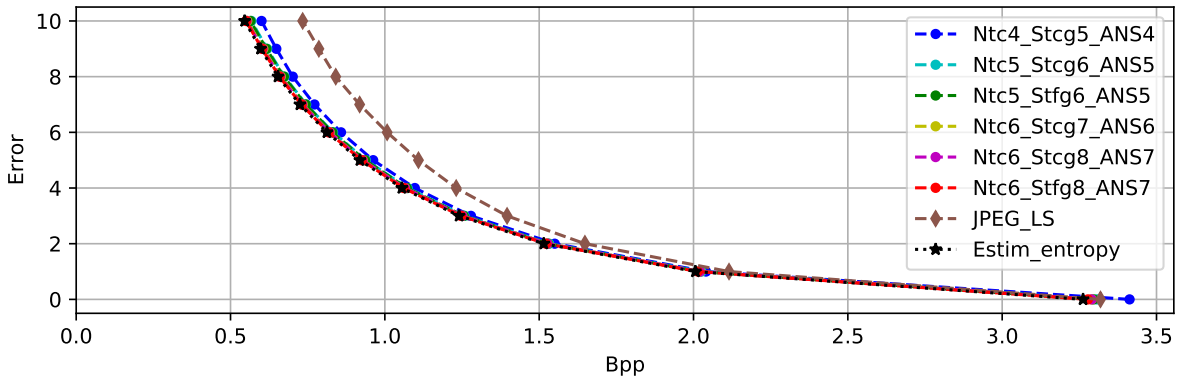


Figure 7.8: Mean bpp using the prototype configurations with NI=7 and JPEG-LS.

The average compression results (over the whole dataset) can be seen in table 7.4 and plotted in fig. 7.8. Additionally, the entropy estimation (according to the model) is shown in the figure to appreciate the efficiency of the configurations.

For these experiments, the code block size was set to 2048 and NI to 7. All configurations surpass JPEG-LS mean compression ratios for all the tested $NEAR$ settings, except for the Nt4_Stcg5_ANS4 configuration for lossless. The highest performing configuration, Nt6_Stfg8_ANS7, achieves a 1.2% mean bpp improvement for lossless, which increases with $NEAR$. Interestingly, even the lighter version, Nt4_Stcg5_ANS4, can obtain remarkable reductions of bpp for near-lossless compression, with improvements ranging from 3.5% for $NEAR = 1$ to 18.2% for $NEAR = 10$. However, for $NEAR > 10$ the improvement percentage for this particular configuration starts to decrease, as the lower entropy distributions require larger ANS state sizes and higher precision estimations.

7.6.1.1 Compression of Photographic Images

It is worth noting that when only considering the photographic images of the dataset, the bpp improvements are greater. In this case, as observed in table 7.5, even the configuration using 4 bits for the ANS state size outperforms JPEG-LS for all the tested $NEAR$ values, including lossless.

7.6.1.2 Effect of Iterations Limitation

The results in table 7.4 and 7.5 correspond to configurations with $NI = 7$. When the number of iterations of the geometric coder are not limited, the compression ratio slightly

Table 7.5: Mean bpp for dataset photographic images for a set of prototype configurations

Error	JPEG_LS	Nt4_Stcg5_ANS4	Nt6_Stfg8_ANS7
0	3.20	3.18 (0.5)	3.15 (1.4)
1	1.95	1.86 (4.9)	1.84 (5.8)
2	1.48	1.36 (8.0)	1.33 (9.7)
3	1.22	1.08 (11.5)	1.05 (13.9)
4	1.06	0.89 (15.5)	0.86 (18.6)
5	0.94	0.76 (18.9)	0.73 (22.7)
6	0.84	0.66 (21.6)	0.62 (26.1)
7	0.76	0.58 (23.9)	0.54 (29.1)
8	0.69	0.51 (25.6)	0.47 (31.7)
9	0.63	0.46 (26.9)	0.41 (34.1)
10	0.58	0.41 (28.2)	0.36 (36.8)

Results for NI=7. The improvement percentage compared to JPEG-LS is shown between parentheses.

increases for lossless compression of the complete dataset, allowing the Nt6_Stfg8_ANS7 configuration to reach a 1.3% improvement over JPEG-LS. However, for near-lossless compression or for the photographic images (including lossless), there is not a practical difference in compression when setting $NI = 7$ compared to not limiting the iterations.

7.6.1.3 Analysis at the Image Level

A comparison at the image level is presented in table 7.6, which shows the number of images of the dataset (and of the photographic image subset in parentheses) JPEG-LS obtains better compression ratios for different error tolerances. The numbers observed for $NEAR = 5$ repeat exactly up to $NEAR = 12$. From that point, configurations with smaller ANS states start to struggle with lower entropy images, which can also be appreciated in fig. 7.8.

The synthetic image "zone plate" is the hardest to compress (according to the model) and the one where JPEG-LS tends to outperform LOCO-ANS. As mentioned in section 4.4.2, because of the change in the gradient quantization function, the entropy estimation for all $NEAR > 0$ worsens for this particular synthetic image. This results in JPEG-LS obtaining a bpp below the estimated entropy, according to the modified model for most $NEAR > 0$. For the best performing LOCO-ANS configuration in table 7.6, this situation occurs for all the cases in which JPEG-LS obtained a better compression ratio, except for one case where the average estimated entropy is 0.0001 bits lower than JPEG-LS bpp. Then, in these cases, the problem lies in the statistical model (which is

better suited for photographic images) and not in the coder.

The best performing configuration introduced in table 7.6 has an increased block size of 16384 symbols. This reduces the KLD due to the need of sending the final ANS state at the end of a code block and aligning each new block to a word, in this case, to bytes (eq. 7.6). For this reason, the configuration achieves a 1.5%,5% and 25.7% mean bpp improvement for $NEAR$ set to 0,1 and 10, respectively, when compressing the complete dataset. These improvements increase to 1.6%,6% and 37.6% when only taking into account photographic images.

Table 7.6: Number of images of the dataset that JPEG-LS achieves a lower bpp. Dataset size: 14 images.

Error	0 [1]	1	2	3	4	5
Nt4_Stcg5_ANS4	8(6)	3(1)	3(1)	3(1)	3(1)	3(1)
Nt4_Stcg5_ANS4 [2]	8(6)	3(1)	3(1)	3(1)	3(1)	3(1)
Nt5_Stcg6_ANS5	4(3)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt5_Stcg6_ANS5 [2]	5(3)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt5_Stfg6_ANS5	1(0)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt5_Stfg6_ANS5 [2]	2(0)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stcg7_ANS6	1(0)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stcg7_ANS6 [2]	0(0)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stcg8_ANS7	1(0)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stcg8_ANS7 [2]	0(0)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stfg8_ANS7	1(0)	1(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stfg8_ANS7 [2]	0(0)	1(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stfg8_ANS7 [2 3]	0(0)	1(1)	1(0)	1(0)	1(0)	1(0)

By default, NI is set to 7 and block size to 2048.

[1] The number of images, considering only photographic ones, is shown between parentheses (total of photographic images is 12).

[2] Configuration with unlimited iterations.

[3] The configuration has a block size of 16384.

7.6.2 Experimental System Efficiency

To evaluate experimentally the sources of inefficiencies, given the chosen model, the KLD resulting from parameter estimation procedures and from the coder were decoupled. To accomplish this, for each image sample, a second average entropy computation was performed, denoted as $H(TSG(\hat{\theta}_q, \hat{p}_q))$, which estimates the bpp assuming an ideal coder. This entropy was computed using the quantized estimations of the distribution parameters (obtained with the procedures described in section 7.4), instead of using the

optimal estimators $\hat{\theta}$ and $\hat{p}$ (computed using eq. 4.5 and 4.6). Then, the KLD due to the distribution parameters estimation procedures was computed as $H(TSG(\hat{\theta}_q), \hat{p}_q)) - H(TSG(\hat{\theta}, \hat{p}))$ and the KLD due to the coder as $bpp - H(TSG(\hat{\theta}_q), \hat{p}_q))$.

The resulting KLD, for all images and for $NEAR \in [0..20]$, is shown in figs. 7.9 and 7.10. These were plotted as a function of $H(TSG(\hat{\theta}, \hat{p}))$. A logarithmic scale is used for the KLD axis, given that values in this axis range over 5 orders of magnitude. The entropy and bpp resulting from the experiments were stored using 4 fractional digits, as it was considered that increasing it would not provide any useful information. Because of this, 10^{-4} is the smallest difference that can be appreciated in the log scale, smaller values (zero or negative) are plotted with a KLD = $6 \cdot 10^{-5}$. In addition, this explains the patterns in the $10^{-4} \leq KLD \leq 10^{-3}$ range.

7.6.2.1 Parameter Estimation Efficiency

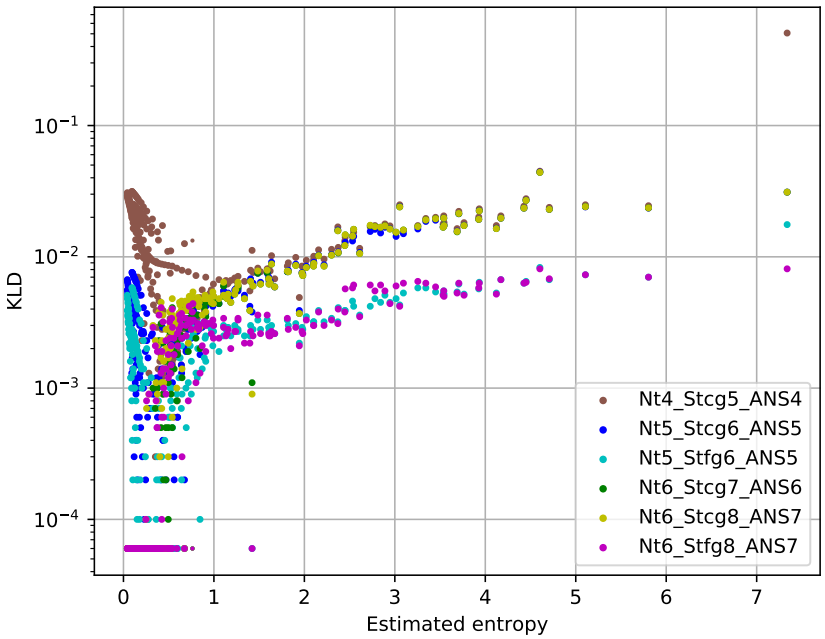


Figure 7.9: KL Divergence due to distribution parameter estimation inefficiencies as a function of the estimated entropy for dataset images and $NEAR \in [0..20]$. The divergence results from using the quantized estimations of the distribution parameters (computed as described in section 7.4) instead of the estimations obtained using eq. 4.5 and 4.6

In fig. 7.9, in general, it can be observed that high efficiencies are achieved by the distribution parameter estimation procedures. For an image average entropy greater than 1, the quantization of $\hat{\theta}$ (indirectly as $\overline{S_t}$ is quantized) dominates observed ineffi-

ciency. Here, a clear separation between prototypes using the coarse grain and the fine grain quantizer for $\overline{S_t}$ can be seen. As a consequence of this effect, the Nt5_Stfg6_ANS5 configuration is capable of matching, and in some cases improving, the compression ratios achieved by the Nt6_Stcg7_ANS6 configuration, whereas the latter requires about twice the memory resources. The finer $\overline{S_t}$ quantization allows the former to be more efficient for medium entropy.

However, when the average entropy diminishes below 1, the effect of having a minimum $\hat{\theta}_q$ and a minimum $\hat{p}_q$ starts to be noticeable (the entropy diminishes as $\theta \to 0$ and p moves away from .5). Here, the main parameter that separates the points of the plot is the ANS state size, which determines these minimums. Additionally, the quantization of $\hat{p}$ contributes to the increase of the KLD, as it is less efficient in this zone (observed in fig. 7.5).

On the other end of the range, for high entropy, the effect of having a maximum $\hat{\theta}_q$ would also increase the KLD. This can only be observed for the prototype using a 4-bit ANS state when losslessly coding the greatest entropy image (zone plate).

7.6.2.2 Coder Efficiency

In the case of the coder KLD, seen in fig. 7.10, the relevant parameters are the ANS state size, the code block size and NI. For an entropy in the $(1,5)$ range, the KLD is basically flat, with a small positive slope, and it would mainly come from ANS intrinsic KLD (eq. 4.1) and the code block size (eq. 7.6). The magnitude of the KLD due to the latter can be appreciated comparing the Nt6_Stfg8_ANS7 prototype with block sizes of 2048 and 16384 symbols (both with unlimited iterations). Note, however, that these differences are not as big as the plot might suggest, given that the KLD is plotted on a logarithmic scale. Additionally, notice that with a block size of 16384 symbols the coder of this prototype achieves a practically null KLD.

The increase in the KLD observed for the lower entropy range is due to the use of the suboptimal tables for the smaller $\hat{\theta}_q$ and $\hat{p}_q$ (see the selection methodology in section 7.5). The lower the average entropy, the more probable it is to use these tables, then the KLD increases. Although the use of these suboptimal tables increases the coder KLD, including these smaller distribution parameters more than compensates, then, the final effect is a reduction in the overall KLD.

As the entropy increases, the KLD can increase for several reasons: the limitation of the geometrical coder iterations (eq. 7.8), the increase of C as $\hat{\theta}_q$ increases (which in turn increases the intrinsic ANS KLD as indicated by eq. 4.1) and the increased average

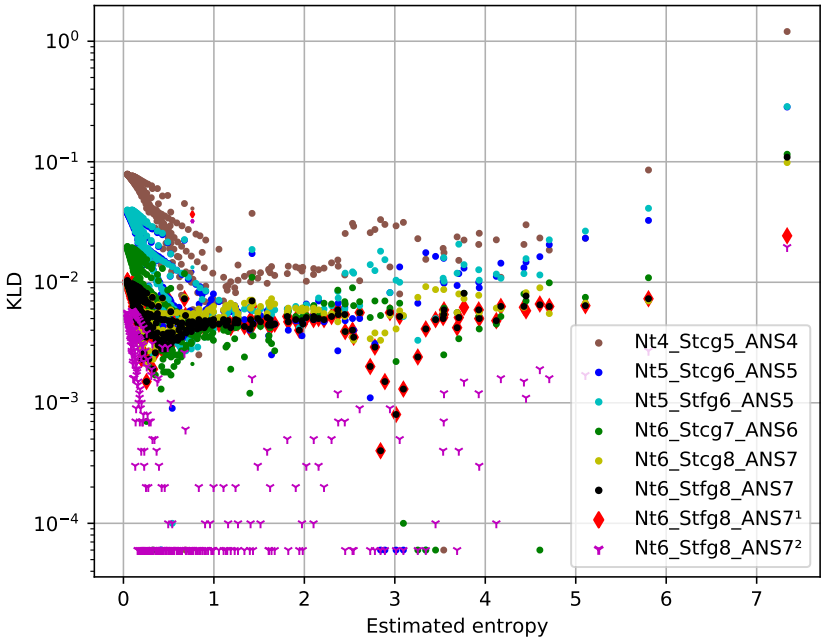


Figure 7.10: KL Divergence due to coder inefficiencies as a function of the estimated entropy for dataset images and $NEAR \in [0..20]$. Computed as the average bits produced by the coder minus the average entropy, assuming the quantized estimations of the distribution parameters are optimal.
Default configuration: NI=7 and code block size = 2048 symbols.
[1] Configuration with unlimited iterations
[2] Configuration with unlimited iterations and code block size = 16384 symbols

iterations (which make the coder incur in the ANS intrinsic KLD several times). The effect of NI can be observed comparing the points corresponding to the Nt6_Stfg8_ANS7 prototype with $NI = 7$ and with unlimited iterations. For low and medium entropy, the two configurations result in approximately the same KLD, while for higher ones the KLD due to the escape mechanism can be appreciated. To understand the magnitude of the effect of increasing C as $\hat{\theta}_q$ increases, the highest entropy cases were compressed with a modified Nt6_Stfg8_ANS7 prototype, setting $\max(C) = 4$. The resulting KLD (not shown in the figure) went back to the (0.001- 0.008) range when the iterations were not limited. For this modified configuration, the average iterations increased (nearly doubled).

7.6.3 Software Performance Comparison

In this section, we compare LOCO-ANS in terms of compression ratio and encoder/decoder speed against well-known and recently developed lossless and near-lossless codecs:

- JPEG-LS (implementation: [128]).
- CALIC (implementation: [129]).
- JPEG2000 Part 1 [130] (implementation: [131]) and JPEG2000 Part 15 High-throughput JPEG2000 (HTJ2K) [132] (implementation: [133]) (none of them provide near-lossless compression).
- WebP [134] (implementation: [135]).
- WebP2, currently under development (implementation: [136]).
- JPEG-XL [137, 138] (although not yet a standard, it is currently under evaluation and the reference software is available [139]).

The tests were carried out in a Raspberry Pi 3 Model B with 1 GB of RAM. This platform was chosen because it better resembles, compared to an x86_64 system, the memory and compute limitations that embedded systems tend to face, which is our target. In addition, it is a widespread platform, facilitating the reproducibility of the results here presented.

All tests were run using a single thread, given that most codec implementations do not have multi-threading capabilities, although they could support it. For example, images could be divided in tiles, like JPEG-XL or JPEG2000 can do. In the case of JPEG-LS, although this is not part of the standard, this could be easily implemented, like in the hardware implementation presented in [40]. This tiling, when performed dividing the image vertically, not only allows for a higher level of parallelism but also tends to benefit JPEG-LS statistical modeling, thus increasing compression (demonstrated by the tests).

Additionally, to show other possible speed-compression trade-offs, a version of LOCO-ANS using four gradients to define the context, as in the original LOCO-I and the standard extension, was also included. Finally, the configurations used for the codecs can be seen in table 7.7.

7.6.3.1 Dataset

Given the large memory requirements of JPEG-XL (even for the lower effort setting "-s0"), it was not possible to process the largest images of the Rawzor dataset using this codec in the chosen platform. For this reason and to obtain more robust results, these tests were run using gray versions of a subset of the Challenge on Learned Image Compression (CLIC) [140] training dataset (this subset was used in the evaluation of

Table 7.7: Codec configurations used in the tests

Codec	configurations
LOCO-ANS (conf. 1)	Nt6_Stcg8_ANS7, 3 grad [1]
LOCO-ANS (conf. 2)	Nt6_Stfg8_ANS7, 4 grad [1]
CALIC	Arithmetic coder
JPEG2000	lossless
HTJ2K	lossless
WebP	effort level(-z) =[1,2] [2]
WebP2	effort level(-effort)=[1] [2]
JPEG-XL	num_threads=0, modular, speed(-s)=[2..5] [2]

[1] NI was set to 7, tables for $p \in [0, 0.5]$ and a block size of 16384

[2] Other effort modes were supported, but they do not compare favorably against the presented configurations of the same codec or encoder times were well over an order of magnitude slower than LOCO-ANS

JPEG-XL lossless compression, and it is available in [141]). It contains 303 2048x1320 photographic images and was not used during the development of LOCO-ANS, so it is also good for validation purposes.

7.6.3.2 Analysis

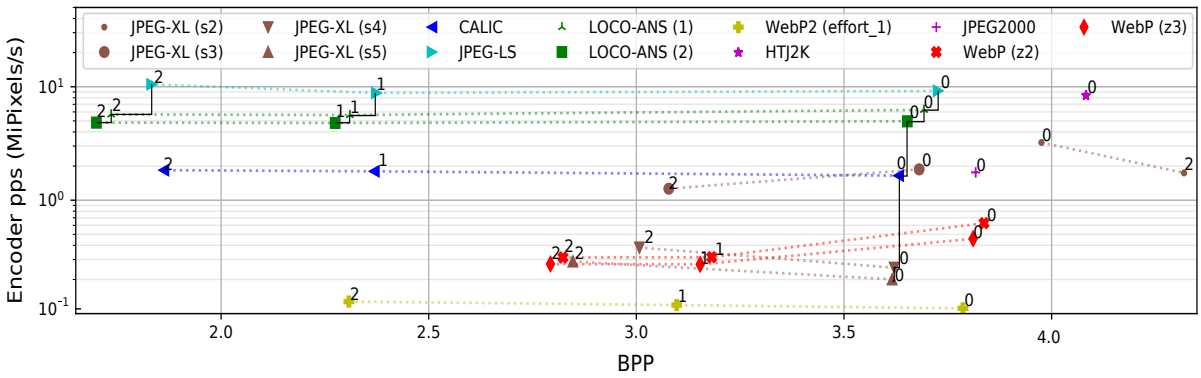


Figure 7.11: Average encoder MiPixels/s versus average bpp for software implementations of different codecs. Numbers next to each point indicate the corresponding peak error. Pareto frontier is drawn with a solid line for error tolerances $\in \{0, 1, 2\}$

The results for encoder and decoder procedures are presented in figs. 7.11 and 7.12, respectively. In addition, table 7.8 summarizes the results for lossless compression, where entries are sorted by bpp. As expected for software implementations, the increased compression obtained by LOCO-ANS comes at the cost of a reduction in the encoder and decoder speeds compared to JPEG-LS. Specifically, the tests show a 32% and 46% encoder

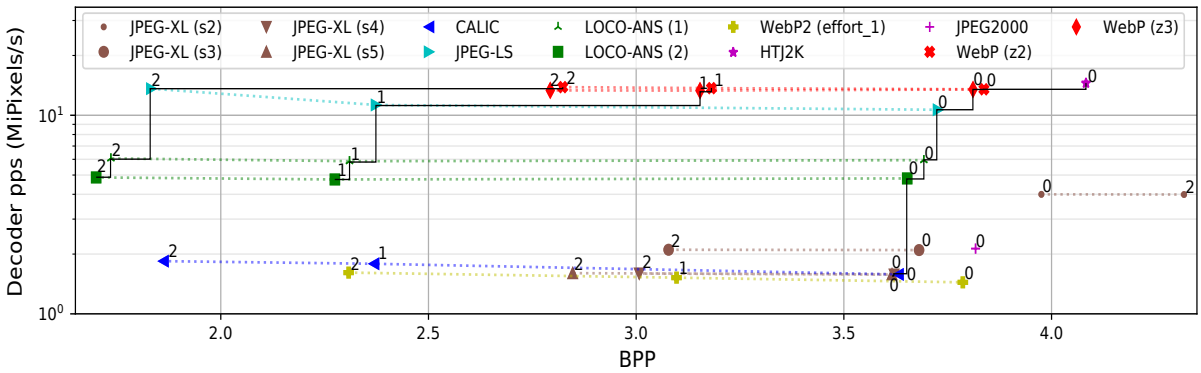


Figure 7.12: Average decoder MiPixels/s versus average bpp for software implementations of different codecs. Numbers next to each point indicate the corresponding peak error. Pareto frontier is drawn with a solid line for error tolerances $\in \{0, 1, 2\}$

speed reduction and a 42% and 54% decoder speed reduction for lossless compression. As the peak error increases, both implementations run-times tend to decrease, although the relative comparison favors JPEG-LS, which can be explained by the incremented use of the run-length coder.

Despite this decrease in performance, given the codecs utilized in this comparison, both of the LOCO-ANS configurations presented are on the Pareto frontier [142] of encoder speed versus bpp and decoder speed versus bpp. When it comes to near-lossless

Table 7.8: Encoder/Decoder speed comparison for lossless compression

Codec	bpp	Encoder BW (MiPx/s)	Decoder BW (MiPx/s)
JPEG-XL (s5)	3.62	0.20 [1]	1.58 [2]
JPEG-XL (s4)	3.62	0.25 [1]	1.58 [2]
CALIC	3.63	1.65 [1]	1.58
LOCO-ANS (2)	3.65	4.96 [1]	4.81 [2]
JPEG-XL (s3)	3.68	1.87	2.10
LOCO-ANS (1)	3.69	6.26 [1]	5.94 [2]
JPEG-LS	3.73	9.21 [1]	10.66 [2]
WebP2 (effort 1)	3.79	0.11	1.44
WebP (z2)	3.81	0.46	13.48 [2]
JPEG2000	3.82	1.76	2.13
WebP (z1)	3.88	0.65	13.59 [2]
JPEG-XL (s2)	3.98	3.22	4.00
HTJ2K	4.08	8.40	14.52 [2]

[1] No other codec simultaneously encodes faster and achieves a lower bpp (in Pareto frontier).

[2] No other codec simultaneously decodes faster and achieves a lower bpp (in Pareto frontier).

compression, most codecs do not perform so well. Particularly, in the case of JPEG-XL, the near-lossless quantization is done as a preprocessing step that reduces the cardinality of the prediction errors, not the range of these errors, and then, it is up to the entropy encoder to detect and exploit the reduced error set cardinality. For this reason, the faster compression modes not only fail to increase compression, but they decrease it. Conversely, LOCO-ANS excels in this type of compression achieving the highest compression ratios for a given peak error, in the presented order of magnitude of encoder speed, and it is only surpassed in encoder speed by JPEG-LS and in decoder speed by JPEG-LS and WebP.

Lastly, to show that tilling does not worsen LOCO-ANS performance and given that the prototype supports it, tests were also run dividing the images in 4 columns (number of cores available in the Raspberry Pi platform). As a result, 3.67 and 3.64 bpp (2.3% improvement compared to JPEG-LS) were obtained for configurations 1 and 2, respectively, compared to 3.69 and 3.65 bpp without tilling.

7.6.3.3 Comparison with Other ANS-based Approaches

Although JPEG-XL entropy encoder is also based on ANS and the implementation used in the tests is highly optimized, it runs slower (in general several times) than LOCO-ANS, even using low-effort modes. Moreover, CALIC compares favorably against it.

JPEG-XL uses a modified version of Range ANS (rANS) to encode symbols given clustered histograms. To perform this operation, all prediction residuals are computed (that is, for the complete image) and then histograms for each context are generated. In general, these contexts are dynamically determined (obtained at run-time). After that, histograms can be clustered (context merging) and the final histograms are signaled to the decoder. Look-up tables to speed up rANS (with a not trivial initialization) are generated for each of these histograms (these tables are called *Alias Tables*, not to be confused with tANS tables). Finally, residuals (or more generally, tokens), after going through other numerical manipulations, are coded with rANS using a 32bit state. ANS code blocks coincide with a tile (256x256 pixels).

To our understanding, JPEG-XL aims to be a general-purpose codec, although oriented to web image delivery [143]. In this scenario, it is reasonable to allow higher complexity (higher computation time and/or computation/memory resources), particularly on the encoder side. This encoder vs decoder speed trade-off is also observed in WebP and WebP2. However, the aim of our work was to improve image compression in sit-

uations with harder constraints (low resources, low-energy budget, high throughput). It is easy to observe that, given their resource requirements, many of the sub-processes that JPEG-XL performs to code the generated tokens are not well suited for high-performance hardware nor embedded software implementation. This is also the case for other procedures that are part of the JPEG-XL codec, for example, those that require full image scanning.

In contrast, LOCO-ANS approach, based on static tANS using parametric distributions (instead of rANS using clustered histograms) leads to higher throughput (illustrated by figs. 7.11 and 7.12). Additionally, it is more suitable for a hardware implementation, given that buffering is limited, simple arithmetic is used and tables are generated at compile time, which allows both software and hardware optimizations, particularly in the latter case.

7.6.4 Discussion

Given the obtained results, it is observed that the proposed TSG ANS coder is particularly well suited for sources with an entropy in the $(.15, 4)$ range, approximately. Even, the 4-bit ANS state configuration achieves a great efficiency with low memory resources, and it is capable of high-throughput operation. Considering the strengths of the GPO2 and the run-length coders, it would be interesting to combine these with the proposed coder. The resulting system may achieve the best complexity-efficiency trade-off for a vast range of applications.

Additionally, it is worth noticing that the TSG coder (or just the geometrical coder) could also be used in other applications, such as audio compression. For example, in the case of MPEG-4 ALS [144] or FLAC [145], the prediction error distribution could be modeled as a two-sided geometric.

7.7 Conclusion

Improved lossless and near-lossless compression was achieved through a series of modifications of the JPEG-LS standard. Particularly, the development of an ANS-based coder for two-sided geometric sources provides highly efficient and low complexity coding. Additionally, this coder enabled the introduction of more precise distribution parameter procedures and to quantize more effectively the gradient defined context space.

The system as a whole admits a wide range of configurations, providing the capability to obtain different trade-offs between coding efficiency, resources and throughput, which allows it to be used in a variety of applications. A prototype available to the community was implemented and a set of experiments were run with different configurations to explore the design space. These configurations range from a very low resource instance that outperforms JPEG-LS in near-lossless compression to an instance using 64 tables with a 7-bit ANS state that closely approaches the estimated entropy.

When compared to JPEG-LS baseline compressing photographic images, LOCO-ANS, using the same context size, can achieve up to a 1.6%, 6% and 37.6% mean bpp improvement for an error tolerance set to 0, 1 and 10, respectively.

Allowing an increase in the context size and image tiling, a 2.3% lower bpp is obtained for lossless compression. Moreover, LOCO-ANS approaches lossless compression rates of more complex encoders, even surpassing them in near-lossless compression, and obtaining a much faster encoder speed.

In appendix B, examples of compressed images of the used datasets can be found.

CHAPTER 8

LOCO-ANS IMAGE CODEC: HARDWARE IMPLEMENTATION

In this chapter, we present and evaluate a High-Level Synthesis implementation for the LOCO-ANS lossless and near-lossless image compressor, which is based on JPEG-LS standard. The design is implemented in two FPGA generations, evaluating its performance for different codec configurations. Compared to the single thread LOCO-ANS software implementation running in a 1.2GHz Raspberry Pi 3B, each hardware lane achieves 6.5 times higher throughput, even when implemented in an older and cost-optimized chip like the Zynq 7020. Results are also presented for a lossless only version, which achieves a lower footprint and approximately 50% higher performance than the version that supports both lossless and near-lossless. Moreover, the implemented system is faster and achieves higher compression than the best previously available near-lossless JPEG-LS hardware implementation. These results show that the algorithm is very suitable for hardware implementation.

This chapter is based on the work published in [21]: Alonso, T., Sutter, G.,& López de Vergara, J. E. (2021). An FPGA-Based LOCO-ANS Implementation for Lossless and Near-Lossless Image Compression Using High-Level Synthesis. Electronics, 10(23), 2934.

8.1 Introduction

The final goal of the algorithmic development, presented in chapter 7, was to obtain an image codec that surpassed the compression ratios of JPEG-LS while keeping its good performance in hardware. For this reason, and given that the transmission side is normally the one that faces more stringent conditions, and thus requiring hardware implementations, a hardware LOCO-ANS encoder was developed. The system has completely designed using High-Level Synthesis, allowing a faster development. The architecture was analyzed and implementation results for two FPGA technologies and for several encoder configurations were obtained to determine the performance and understand the system bottlenecks. The complete set of sources required to reproduce the systems here presented are open to the community through a publicly available repository [126].

8.2 Encoder Architecture

The block diagram in fig. 8.1 shows the main modules composing the system: The Pixel Decorrelator, S_t Quantizer, and TSG coder. Each of these modules is implemented in C/C++ with compiler pragmas and transformed to RTL code using Vitis HLS.

The pixel decorrelator takes pixels as input and outputs a stream of symbols (y,z) with metadata $(\hat{p}_q,t,S_t)$. The last two variables are further processed by the S_t quantizer to generate the $\hat{\theta}_q$ geometric distribution parameter, finally obtaining the TSG symbols (y,z) with their distribution parameters $(\hat{p}_q,\hat{\theta}_q)$. The TSG coder transforms the y and z streams in blocks of bits and, finally, the File Writer output these streams and header information, issuing the appropriate DMA commands.

The TSG coder may need several cycles to code a symbol, but it is capable of much faster operation than the Pixel Decorrelator, so to increase the encoder throughput, the former module runs at a higher clock frequency. FIFO memories are inserted between these modules to move data from one clock domain to the other.

Subsections below explain in more detail each module.

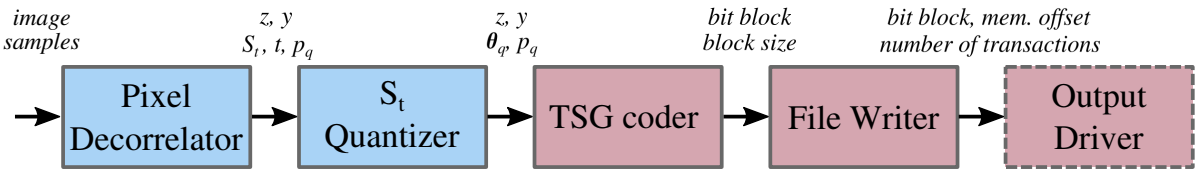


Figure 8.1: LOCO-ANS hardware high-level block diagram. In blue, modules running at the lower frequency, and in red, modules running at the higher frequency.

8.2.1 Pixel Decorrelation

Given the sequential nature of the pixel decorrelation algorithm, it is mainly implemented by a single pipelined module, including a single line row buffer. It consists of an initialization phase and the pixel loop. In the initialization phase, the first pixel is read (which is not coded but included in the bitstream directly), context memories and tables used in the pixel loop are initialized according to the $NEAR$ parameter setting. The operation takes about 512 clock cycles to complete. This could be optimized in many ways, such as computing and storing several memory entries in a single cycle, or avoiding the re-computation of tables when $NEAR$ does not change. Additionally, ping-pong memories could be used to achieve zero-throughput penalty, initializing these memories in a previous pipeline stage, as done in [40]). However, the HLS compiler did not support some constructions required to create that architecture. Although workarounds exist, the potential benefit for HD and higher resolution images is negligible (less than 0.056% performance improvement in the best case and assuming the same clock frequency is achieved). What is more, particularly in high congestion implementations (i.e. FPGAs with high usage ratio), this could even reduce the actual throughput, given that the extra logic and use of additional memory ports can imply frequency penalties. For these reasons, and given that other works have presented optimized architectures for this part of the algorithm (changes to the JPEG-LS algorithm do not have important architectural implications), these initialization time optimizations were not implemented.

Alg. 8.1 describes the pixel loop. This code structure allowed a deep pipeline (shown in fig. 8.2), which reads the row buffer, computes the quantized gradients g_1 and g_2, which do not depend on the previous pixel (after quantization), and starts to compute the context ID before the previous pixel quantization is finished. To obtain the context ID and sign, the value $Q(g_1)\cdot 81 + Q(g_2)\cdot 9 + Q(g_3)$ is computed, where only the g_3 gradient uses the previous pixel. Then, $Q(g_1)\cdot 81 + Q(g_2)\cdot 9$ can be computed in an earlier stage, which is what the pipeline does. Observe that the gradients order in the equation was chosen such that the dependency between loop iterations is eased, as the component

Algorithm 8.1: Pixel loop algorithm structure

```
1:  q_pixel ← first_px
2:  for i ∈ [1, image_size) do
3:      #pragma HLS PIPELINE II=2          // The lossless optimized version uses II=1
                                           //  Data stored in the row buffer does not establish dependencies
4:      #pragma HLS DEPENDENCE variable=row_buffer intra false
5:      #pragma HLS DEPENDENCE variable=row_buffer inter false
6:      Store q_pixel in row buffer
7:      Read new pixel
8:      Compute fixed prediction, context id, and sign
9:      Get context bias and statistics
10:     Correct prediction and compute error
11:     Perform error quantization and modulo reduction
12:     Send symbol with metadata to the output
13:     q_pixel ← Reconstruct the pixel
14:     Update context statistics
15: end for
```

requiring g_3 (which cannot be computed earlier) is not multiplied by any factor.

Additionally, to improve the performance (reducing the II), the updated context data is forwarded to previous stages when two consecutive pixels have the same context. Originally, this optimization was done explicitly in the code and using pragmas (to inform the compiler of the false dependency), but newer versions of the HLS compiler perform this optimization automatically.

Since the HLS compiler handles the scheduling of the operations, the number of pipeline stages may change depending on the target frequency and FPGA. For the tested technologies, aiming at the maximum performance, the pixel loop operations were scheduled in five stages.

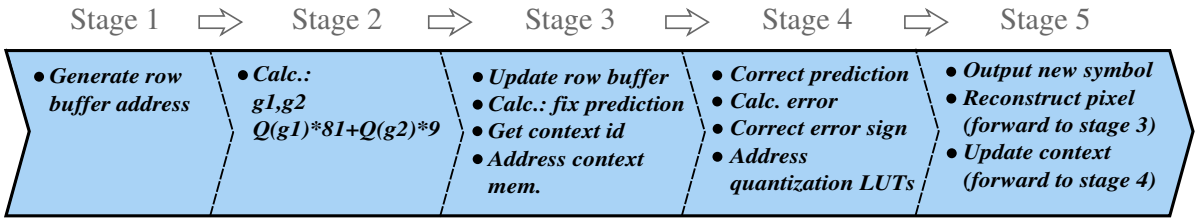


Figure 8.2: Pixel decorrelator pipeline.

8.2.1.1 Obtaining the Distribution Parameter $\hat{\theta}_q$

The decorrelator keeps for each context a register $S_t = \sum_{i=0}^{t} z_i$. The register and the context counter t are then processed by the downstream module $\mathrm{S_t}$ Quantizer (fig. 8.1) to obtain the quantized distribution parameter $\hat{\theta}_q$. The implemented quantization procedure is a generalization of the iterative method used in LOCO-I to obtain the k parameter of the Golomb-power-of-2 coder [52] and it is described in detail in section 7.4.2. Alg. 8.2 shows the coarse-grained configuration of this quantization function.

Algorithm 8.2: Coarse grained θ quantization function (Q_θ)

Require: S_t
Require: t
Ensure: $\hat{\theta}_q$
1: **#pragma** HLS PIPELINE
2: $\hat{\theta}_q \leftarrow 0$
3: **for** $i \in [1, MAX\_THETA\_ID]$ **do**
4: **if** $(St > (t << (i-1)))$ **then**
5: $\hat{\theta}_q \leftarrow i$
6: **end if**
7: **end for**

Although this procedure could have been done within the decorrelator, it was decided to keep it separated, to ease the scheduler job and ensure this operation extended the pipeline without affecting the pixel loop performance. This operation can be compute-intensive, but as there are no dependencies among consecutive symbols, the module can be deeply pipelined, achieving high throughput.

8.2.1.2 Near-lossless Quantization and Error Reduction

To handle the quantization processes, a set of tables[1] was designed to increase the system performance, taking into account that even small FPGA have plenty of memory blocks to implement these tables. The alg. 8.3 describes the error quantization (lines 1-5), modulo reduction (lines 6-10), and re-scale (line 11) processes.

As suggested in [52], the error quantization can be easily implemented using a table. However, here it was decided to store in the table the result after the modulo reduction logic, as the memory resources are reduced, and it helps to speed up the context update, which is one of the logical paths that limits the maximum frequency. A second table

[1]The term look-up table (LUT) is usually used to refer to these tables, but here it is avoided in order not to confuse it with the FPGA resource also denominated LUT

Algorithm 8.3: Error quantization and modulo reduction

Require: ϵ // Input error
Ensure: ϵ_q // Output symbol
Ensure: ϵ_{re} // Re-scaled error, ysed to update context bias
// Uniform quantization

1: **if** $\epsilon > 0$ **then**
2: $\epsilon_q \leftarrow (NEAR + \epsilon)/(2 * NEAR + 1)$
3: **else**
4: $\epsilon_q \leftarrow -(NEAR - \epsilon)/(2 * NEAR + 1)$
5: **end if**
// Reduction modulo α = f(NEAR, pixel depth)
6: **if** $\epsilon_q < MIN\_ERROR$ **then**
7: $\epsilon_q \leftarrow \epsilon_q + \alpha$
8: **else if** $\epsilon_q > MAX\_ERROR$ **then**
9: $\epsilon_q - \leftarrow \epsilon_q - \alpha$
10: **end if**
11: $\epsilon_{re} \leftarrow \epsilon_q * (2 * NEAR + 1)$

contains the re-scaled error (ϵ_{re}), to avoid the general integer multiplication logic and also to ease the sequential context dependency.

Additionally, a third table is used, in this case, to speed up the pixel reconstruction process, which is the other important logical path that could limit the maximum frequency. There are several ways to perform this, as is shown in fig. 8.3. To our knowledge, previous implementations of the LOCO/JPEG-LS encoder reconstruct the pixel starting from the quantized prediction error (as indicated in the ITU recommendation [37]) or from the re-scaled error (e.g. [128]). Instead, we use the value of the exact prediction error (only available on the encoder), to get the reconstructed pixel. Given a $NEAR$ value, each integer will have a quantization error, which can be pre-computed and stored in a table. Then, the exact prediction error (before the sign correction) addresses the table that provides the quantization error, and it is then added to the original value of the pixel. As it can be appreciated in fig. 8.3, using this method greatly simplifies the computation and eases the path. This is one of the key ideas that enabled our high-throughput implementation.

These tables could be implemented as ROMs, supporting a small set of $NEAR$ values, or implemented by RAMs, which are filled depending on the $NEAR$ value currently needed. In the presented design, the latter option was chosen, giving the system the flexibility to use any practical $NEAR$ value, using 3 tables with $2^{pixel\,depth+1}$ entries each. The time required to fill these memories can be masked, as stated before. Although

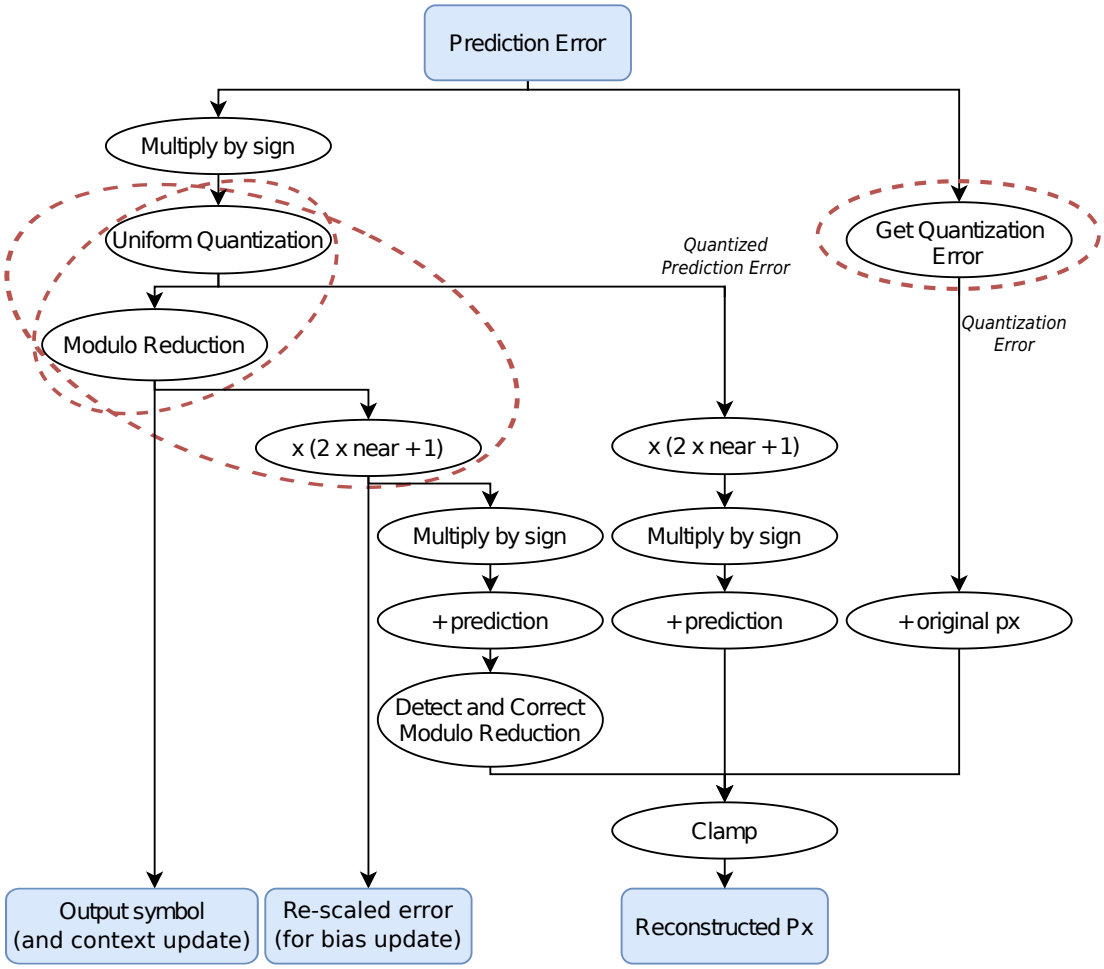


Figure 8.3: Quantization processes. The operations performed by tables are indicated with red ellipses.

the uniform quantization would require general integer division, the tables are filled with simpler logic. It is easy to see that, if sweeping the error range sequentially (either increasing or decreasing by 1) and starting from zero, almost trivial logic is required to keep track of the division and remainder.

If a single clock and one edge of the clock are used, the minimum II for the system will be 2. To compute the prediction, the context memory is read (memory latency >= 1), then the prediction error is obtained, which is needed to address the quantization tables (also implemented with memories with a latency >= 1). The result of the quantization process is used to address the next pixel context, producing a minimum $II = 2$.

Within a module, Vitis HLS does not allow the designs with multiple clocks or using different clock edges. However, in this case, a great improvement is not expected from the implementation of these techniques, they will imply a much greater development

time and the result will tend to be more technology-dependent (given that the FPGA fabric architecture and relative propagation times vary, affecting the pipeline tuning).

8.2.1.3 Decorrelator Optimized for Lossless Compression

A decorrelator optimized just for lossless compression operation was also implemented. The removal of the quantization logic, plus the logic simplification that arises from using only $NEAR = 0$, allows going from an $II = 2$ to $II = 1$ with approximately a 25% frequency penalty in the tested technologies. That is about a 50% throughput increase (see section 8.3). In this case, this pixel loop is implemented with a 4-stage pipeline and the frequency bottleneck is established by the context update.

An interesting fact about this optimization is that going from the general decorrelator to testing on hardware, a first lossless only version took less than one hour. Such fast development was possible given that just a few lines of C++ code needed to be modified. These simple modifications led to significant changes in the scheduling of the pipeline, resulting in the stated performance, which would have been much more time-consuming using HDL languages.

8.2.2 TSG Coder

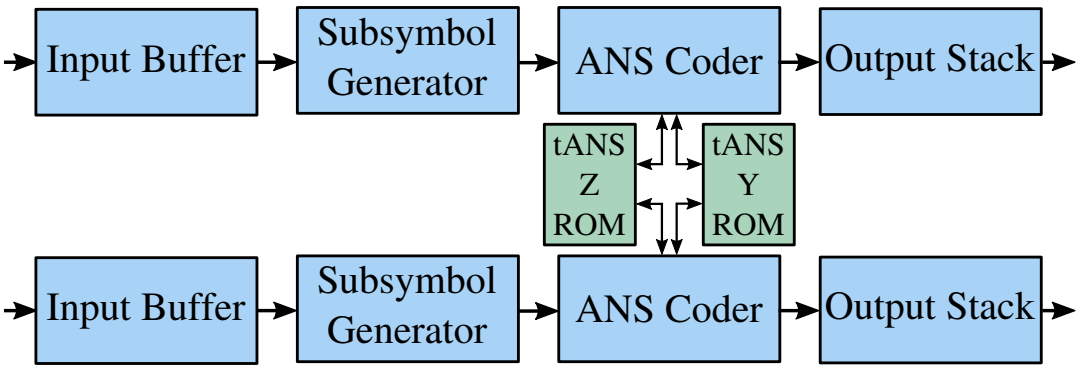


Figure 8.4: High-level block diagram of the double lane TSG coder.

Fig. 8.4 shows the block diagram of the double lane TSG coder, which allows sharing the tANS ROMs without clock cycle penalties, as double port memories are used and each lane requires one port. This module can receive the output of two independent Pixel Decorrelators and process them in parallel. In this way, it allows the compression of images in vertical tiles, which improves compression for HD and higher resolution images (see section 7.6.3).

The system was designed in a 2-level hierarchy because the basic data element changes as we go downstream. The Input Buffer works with blocks of symbols, while the Subsymbol Generator works at the symbol level, the ANS Coder at the sub-symbol level, and the Output Stack with blocks of packed bits. This modularization allows easily choosing the coding technique better suited for each module. The modules shown in fig. 8.4 are instantiated in a dataflow region synchronized only by the input and output interfaces, such that each module can run independently. In Vitis HLS, this is accomplished with the following pragmas:

```
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS DATAFLOW disable_start_propagation
```

8.2.2.1 *Input Buffers*

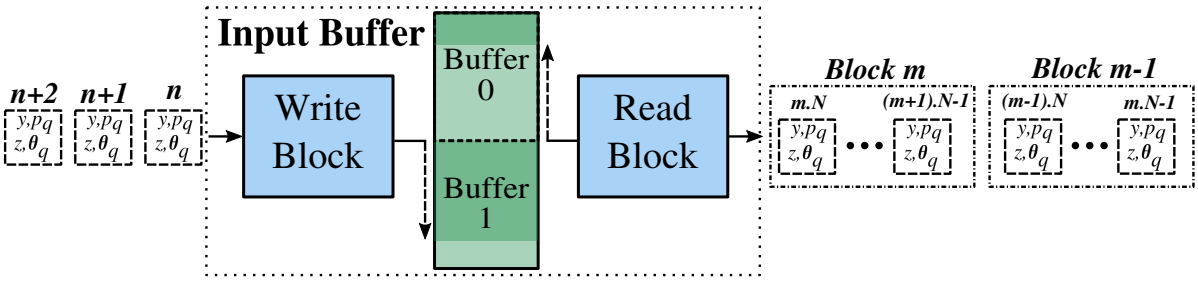


Figure 8.5: Input Buffer block diagram, showing its operation for block size N.

The main function of the Input Buffer is to invert the symbol order to make the adaptive coding with ANS practical (complex methods would be required otherwise). However, to avoid the use of large memories, this module creates blocks of symbols, and the order within each block is inverted (see fig. 8.5). The *write and read* pipelined functions are instantiated in a dataflow region using a ping-pong buffer, given the required non-sequential memory accesses. However, it is noted that there is an alternative with a memory of one block, which comes at the cost of slightly more complex logic.

8.2.2.2 *Subsymbol Generator*

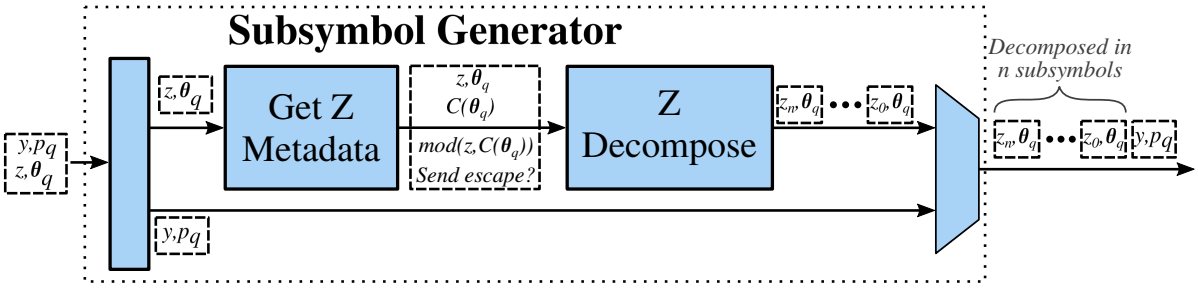


Figure 8.6: Subsymbol Generator block diagram and data transformations within it.

Fig. 8.6 depicts the Subsymbol Generator and how data is transformed as it goes downstream. Its main function is to decompose z in a variable-length sequence of sub-symbols $z_0,..,z_n$, which is one of the main processes of the Geometric coder.

For coding efficiency reasons, the cardinality of the symbol source modeled by the z ANS ROM varies for each distribution parameter θ_q. Then, for a given θ_q tANS will model a distribution of the symbols $[0..C(\theta_q)]$. For this reason, z needs to be represented in terms of these symbols, so it is decomposed as follows: $\sum_{i=0}^{n} z_i = z$, where the first sub-symbol z_0 is equal to $mod(z, C(\theta_q))$ and all the rest are set to $C(\theta_q)$. In this way, to retrieve z, the decoder just needs to sum sub-symbols until it finds one (first encoded, but last decoded) that is different from $C(\theta_q)$. As $C(\theta_q)$ is always an integer power of 2, this process is simple. Finally, if it is detected that the length of this sequence will be greater than a design parameter NI (which determines the maximum number of geometric coder iterations) the sub-symbol sequence represents an escape symbol. Following this sequence, the original z is inserted in the bitstream.

As described in section 7.3.2, this process is used to reduce the cardinality tANS needs to handle, which translates into significantly lower memory requirements and higher coding efficiency while keeping simple operation.

As it decomposes z and serializes the result with y (in the coupled coders version), this module establishes the TSG coder bottleneck in terms of symbols per clock cycle (not the frequency bottleneck, i.e., contains the critical path). Because of this, it was fundamental to optimize this module to be able to output a new sub-symbol every clock cycle. Pipelining the modules was not sufficient to accomplish this goal. As shown in fig. 8.6, the z sub-symbol generation process was split into two modules, one to get the required metadata and another one to decompose the symbol. Also, the Z Decompose module was not described as a loop, as one normally would specify this procedure,

but instead, it was coded as a pipelined state machine, which allowed reaching the desired performance. Finally, all these modules are instantiated in a dataflow region synchronized only by the input and output interfaces.

8.2.2.3 *ANS coder*

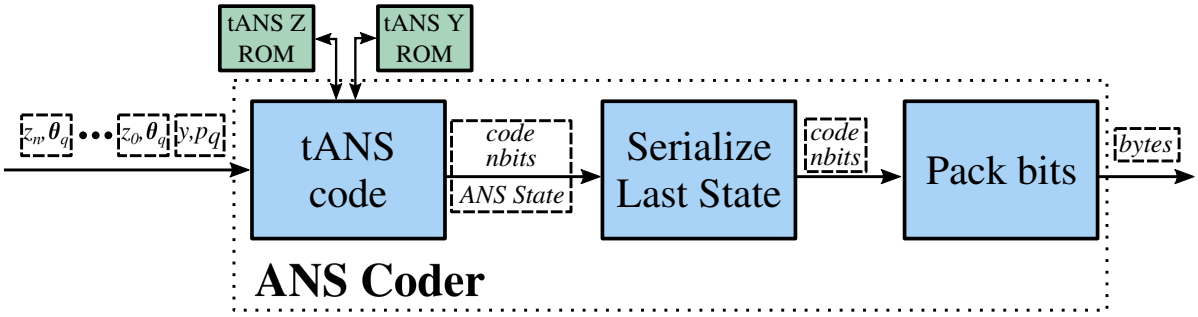


Figure 8.7: tANS coder block diagram showing the transformation of sub-symbols into packed bytes.

As shown in fig. 8.7, the ANS coder is composed of three modules. For each sub-symbol, the first one chooses the tANS table according to the symbol type (z_i or y) and the distribution parameter. This table is then used to obtain the variable-length code for the sub-symbol. Thus, the module implements the Bernoulli Coder and the remaining process of the Geometric Coder. However, they can be easily split, resulting in a simpler module and the ROM memories would have weaker placement and routing constraints. The module also accepts bypass symbols, which are used to insert z after the escape symbol. After the last sub-symbol is coded, the second module inserts the last ANS state as a new code. The last module packs these codes into compact bytes.

The ANS coder can accept a new input in every clock cycle. This was accomplished by instantiating the modules in a dataflow region synchronized only by the input and output interfaces and pipelining each of them with an II=1. This II was achieved by the modularization of the process and by describing all three modules as state machines.

8.2.2.4 *Output Stack*

Finally, the Output Stack is in charge of reversing the order of the byte stream of each block of symbols. For this, it uses a structure similar to the one employed in the Input Buffer.

8.2.2.5 Increasing Coder Performance

Independent component coders As mentioned before, if y and z ANS coders (Bernoulli and Geometric, resp.) are independent, the coder throughput would be increased by a $(\hat{i}+1)/\hat{i}$ factor. As shown in section 7.6.1, $\hat{i}$ tends to be around 1.3 for lossless coding, which is the worst case. Then, applying this value will result in a 1.77 times faster coder. What is more, given that z and y coders will be decoupled and almost no additional logic is required, it is expected that the maximum frequency would be at least the one achieved for the coupled coders. To implement it, the Subsymbol Generator should not serialize z and y, the tANS coder should be split in two (each with one tANS ROM) and the bit packer should merge the two code streams.

Decreasing the maximum iterations limit In addition, the worst-case performance, as well as the maximum code extension, can be controlled using the maximum geometric coder parameter NI. This is particularly important for implementations with limited buffering.

8.3 Results

This section presents how the designs were tested, as well as the achieved frequencies and resource footprints. Finally, throughput and latency analyses are provided.

8.3.1 Test Platform and Encoder Configurations Description

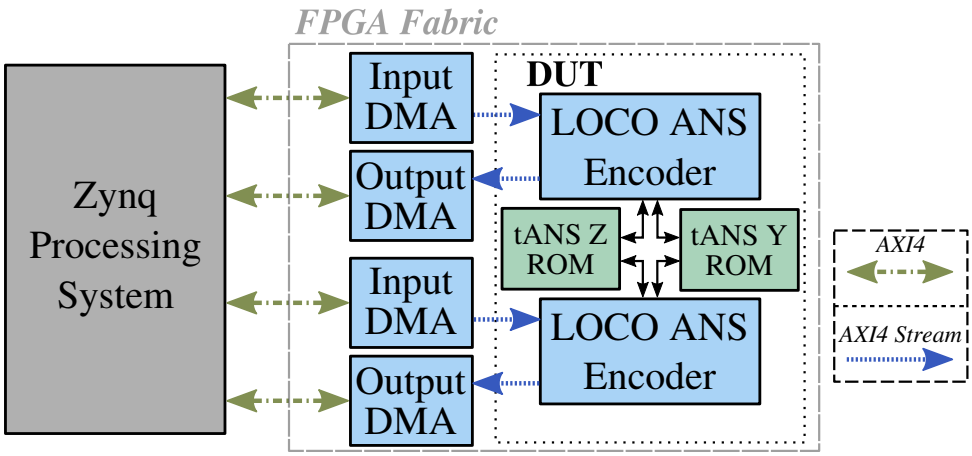


Figure 8.8: Block diagram of the test system.

To conduct the hardware verification, the system depicted in fig. 8.8 was implemented in two different Xilinx FPGA technologies, described in table 8.1: Zynq 7 (cost-optimized, Artix 7 based FPGA fabric) and Zynq UltraScale+ MPSoC. For all implementations, although not optimal in terms of resources, two input and output DMAs were used to simplify the hardware, as the objective was to verify the encoders building a demonstrator, not a fully optimized system. Images were sent from the Zynq μP running a Linux to the FPGA fabric using the input DMAs, which accessed the main memory and fed the encoder using an AXI4 stream interface. As the encoder generates the compressed binary, the Output DMA stores it in the main memory. The evaluation of the coding system was carried out for the configurations in table 8.2.

Table 8.1: Characteristics of target parts used in this work.

Board	FPGA	Speed[1]	Node	LUT	FF	BRAM	DSP	URAM
Pynq Z2	Z-7020 -1	1	28 nm	53K	106K	140	220	-
ZCU104	XCZU7EV -2	2	16 nm	230K	460K	312	1728	96

[1] Speed Grade. For the chosen targets, speed grade ranges from 1 to 3, where 1 is the slowest. In general, we include the speed grade as part of the device name using the format: {version} -{Speed}

Table 8.2: Codec configurations used in the experiments

Configuration	Rel. bpp[1]	State bits	NI[2]	Block size	C range	# of ANS tables for θ	# of ANS tables for p
LOCO-ANS4	-0.5/-5.0	4	7	2048	1-8	11	8
LOCO-ANS6	-1.1/-5.4	6	7	2048	1-8	15	32
LOCO-ANS7	-1.2/-5.6	7	7	2048	1-8	16	32

These configurations correspond to the Nt4_Stcg5_ANS4, Nt6_Stcg7_ANS6, and Nt6_Stcg8_ANS7 prototypes tested in section 7.6.

[1] Bits per pixel relative to JPEG-LS baseline for $NEAR = 0$ and $NEAR = 1$. Data from fig. 8.9.

[2] NI: Number of Geometric coder iterations.

8.3.2 Implementation Results

Table 8.3: LOCO-ANS Encoder implementation metrics for a series of configurations and target parts

Part	Coder config	Clk0/1 (MHz)[1]	LUT	FF	BRAM	DSP
Z-7020	LOCO-ANS4	79.4 / 180.4	4580	4992	19.5	4
Z-7020	LOCO-ANS6	81.1 / 182.2	4832	5160	24.0	4
Z-7020	LOCO-ANS7	79.5 / 167.3	5095	5240	32.0	4
XCZU7EV	LOCO-ANS4	248.3 / 502.2	6580	5954	19.0	4
XCZU7EV	LOCO-ANS6	246.7 / 442.0	6867	6027	23.5	4
XCZU7EV	LOCO-ANS7	234.1 / 395.1	6019	5780	33.5	4
Z-7020	LOCO-ANS4-LS	65.0 / 183.1	3979	4160	16.5	2
Z-7020	LOCO-ANS6-LS	64.3 / 186.0	4248	4298	21.0	2
Z-7020	LOCO-ANS7-LS	62.8 / 166.6	4572	4373	29.0	2
XCZU7EV	LOCO-ANS4-LS	188.4 / 500.5	4706	4949	19.0	2
XCZU7EV	LOCO-ANS6-LS	187.1 / 447.0	4515	4225	21.0	2
XCZU7EV	LOCO-ANS7-LS	185.2 / 387.5	5415	5329	31.5	2

The top half features implementations that support near-lossless compression (including lossless), and the bottom half, lossless-only compression (with -LS suffix).
All the presented implementations have 2 lanes and support up to 8K wide images per lane

[1] Clk0 is the low-frequency clock used for the pixel decorrelation process, while clk1 is the high-frequency clock used for the coder. See fig. 8.1.

For the tested implementations and both technologies, the critical path of the low-frequency clock domain is, in general, in the pixel reconstruction loop for the near-lossless encoders and within the update logic of the adaptive bias correction for the lossless version.

In the case of the high-frequency clock domain, the slowest paths of these implementations tend to be in the TSG coder and the output DMA for the Zynq 7020 implementation. Within the TSG coder, the critical path is, in general, either in the tANS logic (from the tANS ROM new state data output to the tANS ROM address, the new state) or in the Z Decompose module. In the case of the Zynq MPSoC, the slowest paths tend all to be in the tANS logic.

8.3.3 Results Evaluation

Results are analyzed in terms of throughput and latency, which are of paramount importance for real-time image and video applications.

8.3.3.1 Throughput

The near-lossless decorrelator critical path is in the pixel reconstruction loop, which is the same procedure used in the standard. This fact supports that the changes introduced by LOCO-ANS in the decorrelator do not limit the system performance. In the case of the lossless decorrelator, the bias context update logic limits the frequency. This procedure is the same as in the JPEG-LS standard extension, which requires an additional conditional sign inversion compared to the baseline. This tends to worsen the critical path, but it is a minor operation compared to the complete logical path. Although it achieves a slower clock, the lossless decorrelator throughput is about 50% higher than the near-lossless decorrelator, given that it achieves an II=1 instead of II=2.

The presented implementations represent a wide range of trade-offs between performance, compression, and resources (also cost, considering technology dimension). All of them have the Bernoulli and Geometric coders coupled, then their mean throughput will be $clk1/2.3$ MPixel/s for photographic images, where $clk1$ refers to the clock shown in table 8.3. In this way, for a given configuration and target, the TSG coder will have in the mean between 83% and 98% higher throughput than the near-lossless decorrelators for the Zynq 7020 implementations and between 47% and 76% for the Zynq MPSoC. In the case of the lossless optimized decorrelators, this performance gap is reduced to (15%, 26%) and (-10%, 16%), for Zynq 7020 and Zynq MPSoC respectively. From the presented implementations, just one of them shows a lower TSG coder throughput. In this case, the increased compression ratio comes at the cost of not only higher memory utilization but also a throughput penalty.

However, it is observed that many possible optimizations of the TSG coder exist, and particularly of the tANS procedures. The Z ROM memory layout can be enhanced to significantly reduce the memory usage, which could have a positive impact on the maximum frequency, as table 8.3 suggests. Also, alternative hardware tANS implementations exist [57], which may allow a wider range of performance/resources trade-offs.

The obtained results support the hypothesis that the use of the proposed TSG coder, which has a compression efficiency higher than the methods used in JPEG-LS, will not reduce the encoder throughput. This is observed in the hardware tests, where the encoder pixel rate is determined by the decorrelators when photographic images are compressed, except the lower TSG coder throughput case (LOCO-ANS7-LS in the Zynq MPSoC). As expected, this is not the case for randomly generated images, as the coder requires larger code words for them, and then, it is the TSG coder the one that limits throughput, particularly for small images and lossless compression.

8.3.3.2 Latency

The implemented decorrelator latency is determined by the initialization time plus the pixel loop pipeline depth, which results in $512+6 = 518\ cycles$. For the lossless optimized version, this is reduced to $365+4 = 369\ cycles$. In the case of the low-end device implementation (Zynq 7020), this results in 6.3 μs and 5.8 μs latency, respectively. As mentioned before, if required, the initialization time could be reduced or even completely masked, but these optimizations were not implemented due to compiler limitations, and the fact that it was considered that the potential benefits were low.

It is a bit more complicated to obtain the TSG coder latency, as it is data-dependent, and the coder works with blocks of symbols. To determine the marginal latency (delay added by the coder), we consider the time starting when the last symbol of the block is provided to the coder until the moment the coded block is completely out of the module. Then, avoiding the smaller pipeline delay terms, the TSG coder latency can be computed as:

$$(1+\overline{subsym(z)})\cdot BS + \lceil bpp/out\_word\_size \rceil \cdot BS \quad clock\ cycles \tag{8.1}$$

Here, BS is the block size, $\overline{subsym(z)}$ is the mean sub-symbols z is decomposed into, bpp is the mean bits per pixel within the block and out_word_size is the size (in bits) of each element of the output stack. The latency is dominated by two modules: the Subsymbol Generator (first term of the equation) and the Output Stack (second term). This is because, as mentioned before, the former creates a bottleneck given that for each input it consumes it outputs several through a single port and the latter buffers the whole block of output bytes and outputs it in the inverse order.

To obtain a pessimistic mean latency, we assume a low compression rate of 2 ($bpp = 4$). The block size is set to 2048, the output stack word size to 8, and $\overline{subsym(z)} \approx \bar{i} = 1.3$ (as determined in section 7.6.1). Then, for the Zynq 7020 implementation, the mean TSG coder latency is 31.9 μs.

To estimate a practical upper bound to this latency, the following image compression case was analyzed:

- Image pixels equal to $BS = 2048$. In this way, we maximize the block used while keeping the pixel count low, so the decorrelator's capability to learn the statistics of the image is reduced.
- Pixels independently generated using a uniform distribution (worst-case scenario)

and the errors model hurts compression (the prior knowledge is wrong).

- Image shape: 64x32 (cols x rows). This shape allows visiting many contexts, and then, the adaptation of the distribution parameter $\hat{\theta}$ will be slower, thus increasing the resulting bpp.
- $NEAR = 0$ (lossless compression): which maximizes the error range and bpp.

From a set of 100 images generated in this way, we took the lower compression instance, where $bbp = 9.844$ and $\overline{subsym(z)} = 6.31$. This code expansion is because the prior knowledge embedded in the algorithm (coming from the feature analysis of photographic images, such as the correlation between pixels) is wrong in this case and, as the image is small, it does not have enough samples to correct this. Moreover, given that the range of the θ distribution parameter was determined with photographic images, additional θ tables may be needed for these abnormally high entropy sources. Then, using the presented formulas, we obtain 97.2 μs as a practical upper bound on the encoder latency for the Zynq 7020 implementation running at 180 MHz.

Although the presented system establishes a trade-off between latency and compression, the achieved latency is remarkably low and suitable for many real-time systems. Moreover, it is possible to tune this trade-off by modifying the implementation parameters.

8.4 Discussion

In this section, we evaluate the results presented in the previous section as well as analyze them taking prior works into consideration.

8.4.1 Related work

There exists a large set of compression methods that achieve a vast range of compression-resources-throughput trade-offs, but not all have an amenable hardware implementation. The use of dynamic structures tends to make logic slower and require a higher footprint. For example, JPEG-XL [138] can achieve better lossless compression ratios than JPEG-LS, but for that, it needs very flexible contexts and non-trivial logic is used to optimize their histograms and the rANS tables to code for these functions. Also, the use of large memories, like in the case of inter-frame video compression, tends to require external memories, which also contributes significantly to the system power requirements. Given

the fact that this work targets real-time and, in general, highly constrained applications with bounds on the errors generated by the compression system and considering the already mentioned features of the JPEG-LS codec that makes it very suitable for these applications, the discussion is focused on JPEG-LS-like codecs, analyzing the trade-offs within this subregion of the metrics space.

Table 8.4 shows key metrics of the most relevant hardware and, for performance comparison, software codecs implementations. In this section, to provide clearer explanations, we focus on the balanced LOCO-ANS6 configuration.

Table 8.4: Comparison with other codec implementations.

Implementation	Tech	PR [1]	Rel. bpp [2]	Lanes	Mem. bits	Area
LOCO-ANS6 [3]	Zynq 7020 -1	40.6	-1.1/-5.4	2	442.4K	1042 Slices+ 2 DSP
LOCO-ANS6 [3]	Zynq US+ -2	123.4	-1.1/-5.4	2	433.2K	718.5 CLB + 2 DSP
Chen et al. [40] [6]	Virtex 6	25.8	+6.4/+13.0>	2	131.4K	4177 Slices
LOCO-ANS6 [7]	Rasp. 3B [4]	6.3	-1.1/-5.4	1		
Fast JPEG-LS [128][5]	Rasp. 3B [4]	9.2	0/0	1		
LOCO-ANS6 LS [3]	Zynq 7020 -1	64.3	-1.1/–	2	387.0K	639 Slices + 1 DSP
LOCO-ANS6 LS [3]	Zynq US+ -2	187.1	-1.1/–	2	387.0K	548 CLB + 1 DSP
Daryanavard et al. [146]	Stratix 2	155	≈ 0/–	1	9.5K + 1 row	573 ALUT
Murat [41] [5]	Virtex 7 -2	207.8	0/–	1	NR	567 Slices
Kau et al. [42] [6]	Cyclone II	113.0	+1.1>/–	1	12.8K + 1 row	2184 LE

The top half features implementations that support near-lossless compression (including lossless), and the bottom half, lossless-only compression.
Memory bits and area are normalized by the number of lanes.
NR: indicates information not reported
When available, the speed grade is shown to the right of the device name with the "-{speed grade}" format

[1] PR: Pixel Rate in MPixel/s/lane

[2] Bits per pixel percentage decrease (if negative) or increase (if positive) relative to JPEG-LS baseline for $NEAR = 0$ and $NEAR = 1$. Data from fig. 8.9. Lossless-only compression implementations can only provide $NEAR = 0$

[3] Hardware version of the developed codec. Source found in [126].

[4] Software implementations running in Raspberry 3B, with a single thread.

[5] Standard-compliant JPEG-LS implementation

[6] 12-bit image support

[7] Software version of the developed codec. Source found in [147].

8.4.2 Comparison Considerations

Before diving into the analysis of the presented work in light of other works in the area, we examine what we consider the most relevant aspects of the comparison process itself that condition it.

8.4.2.1 Compression Trade-Offs

The fact that most of these implementations use different algorithms complicates performance comparisons, particularly because the compression ratios for a given dataset are not available. Then, it is hard to analyze the trade-offs that each design implies. Although many works claim to be standard-compliant, some present a design that it is not, as they apply several changes to the algorithms, in general, to simplify and/or speed up the implementation. Not supporting the run-mode is a common one.

In [42], for example, we note they introduced the following changes without assessing the implications:

- Not using run-mode.
- Not clamping the corrected prediction (see A.4.2 ITU-T.87). Because of this, the range of the prediction error is increased and, given that JPEG-LS uses limited-length Golomb codes, the binary code after the escape code needs to be increased by 1 bit.
- Error modulo reduction is applied after context bias update (see A.4.5 ITU-T.87).
- Not including the error sign correction required by the bias update (see A.4.3 ITU-T.87). Not applying the error sign correction will have a negative impact on compression, as it is needed to perform the context merge.
- Not limiting the maximum bias correction (see A.6.2 ITU-T.87).

To quantify the impact on the throughput of these changes, we utilize the Vitis HLS implementation feature, which instantiates the resulting HDL module in the target device, performs RTL synthesis followed by place and route (P&R). In this way, it allows obtaining a good estimation of the performance of a module in a non-congested implementation. With these changes, the tool reports that the lossless only decorrelator achieves 100 MHz in the Zynq 7020 (a 55.5% performance increase).

Of course, provided that the trade-offs are understood, changes to the algorithms that improve performance can be useful. For example, in [146] the bias update mechanism was

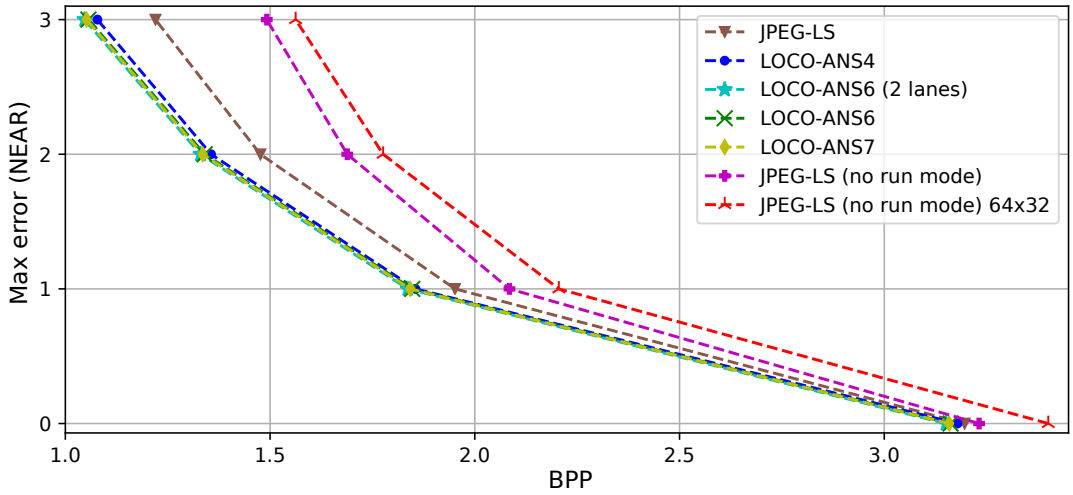


Figure 8.9: Mean bits per pixel (bpp) obtained by JPEG-LS, JPEG-LS without run mode and LOCO-ANS

replaced by a more precise one, which also allowed a much more feed-forward pipeline, resulting in a fasted encoder at the cost of resources. However, in this case, it is not clear whether the presented results are implementation ones or just RTL synthesis.

To better compare the encoders we run compression experiments where, apart from LOCO-ANS and JPEG-LS, we test JPEG-LS without run mode [2] and JPEG-LS without run mode with 32x64 tilling (max tile size supported by [40]). Given the number of changes, and the fact that it probably has issues, we do not attempt to reproduce the algorithm implemented in [42]. In this experiment, we used the photographic (non-artificial) images of the 8-bit gray image dataset maintained by Rawzor [3] for $NEAR \in [0..3]$. The results are presented in fig. 8.9. As it can be appreciated, even when dealing with photographic images, the run-length coder does have a noticeable impact on compression. While LOCO-ANS6 output file size is 1.1%, 5.4%, 9.2%, and 13.4% smaller than JPEG-LS output (for $NEAR \in [0..3]$, respectively), removing the run-length coder increases it by 1.1%, 6.8%, 14.4%, and 22.3%.

Moreover, we can appreciate the effect of different tile sizes. Diving the image in 2 columns (LOCO-ANS6, 2 lanes), which can be compressed in parallel, improves JPEG-LS by 1.4%, 5.9%, 9.9%, and 14.2% for $NEAR \in [0..3]$. We estimate that this improvement comes from the intuition that, for wide images, image statistics vary slower when scanning an image in columns, so the model is more accurate and then, higher compression

[2] This codec was obtained through the modification of the reference libjpeg codec (`https://github.com/thorfdbg/libjpeg`)

[3] `http://imagecompression.info/test_images/`

is achieved. However, using small tiles, and particularly reducing the height, the encoder model does not have enough samples to learn the image statistics, so it does not make good estimations. As a result, JPEG-LS with no run mode with 64x32 tiles worsens compression even further, increasing the output file size by 6.4%, 13.0%, 20.3%, and 28.0%, compared to JPEG-LS.

8.4.2.2 Implementation Technology

Another problem is how to normalize speed, considering the target technology. In the literature, we find implementations in a wide range of devices, using different technologies. Even within the Xilinx FPGAs, it is hard to make performance comparisons as both programmable logic fabric architecture and manufacture node change. Although FPGAs have increased their maximum clock frequency with time, differences between subsequent releases vary and greater variability can exist within a release, considering different architectures and speed grades. Additionally, the clock frequency of feed-forward compute engines (without data dependencies) was able to increase much more with the introduction of more pipeline stages within FPGA hard blocks, like on-chip memories and DSPs. However, codecs with good compression ratios, and particularly JPEG-LS, have feedback loops that cannot be easily sped up.

For a subset of the Xilinx FPGAs used for the hardware codecs works, table 8.5 shows key times involved in the context update logic, which determines the clock frequency of most of these implementations. Observe the relative magnitude of the BRAM clock to output propagation time (without output register) compared to other metrics and that it consumes a significant part of the respective clock periods. Of course, the information in this table is not enough to have an accurate model that would allow fair comparisons between technologies, among other reasons, because routing tends to be a major contributor to the critical paths in FPGA implementations and there is no clear way to compare different fabric architectures. However, this data does seem to explain, at least in part, the frequency jump from Zynq 7020 -1 to Zynq UltraScale+ -2 that we observe in table 8.3.

To overcome this, [41] implemented their architecture, which seems to be standard compliant, in a set of devices used by previous works. As a result, the presented design compared favorably both in terms of speed and resources. For this reason, this work, which achieves 207.8 MPixel/s in a Virtex 7 speed grade 2 with JPEG-LS compression rate, is taken as a reference point to analyze the proposed lossless encoder results. In the near-lossless case, we compare to [40], which is the closest to standard-compliant

Table 8.5: Example of FPGA propagation and set-up times relevant to the critical paths present in most implementations for different technologies

FPGA part		Info			Propagation time			Set up
Device	Speed	Vcc*	Year	Node	LUT	FF	BRAM (no reg)	BRAM
Spartan 3 [1]	-5	1.2	2003	90 nm	530	630	2090	430
Spartan 3 [1]	-4	1.2	2003	90 nm	610	720	2400	490
Virtex 6 [2]	-1	1	2009	40 nm	90	390	2080	620
Virtex 6 [2]	-3	1	2009	40 nm	60	290	1600	470
Artix 7 [3]	-1	1	2010	28 nm	130	530	2460	570
Zynq 7020 [4]	-1	1	2011	28 nm	130	530	2460	570
Virtex 7 [5]	-2	1	2010	28 nm	50	270	1800	420
Zynq US+ [6,7]	-2	0.85	2015	16 nm	35-50	80	979-1020	283

Propagation and setup times values expressed in picoseconds

* Recommended or middle of range internal device voltage in Volts

[1] https://www.xilinx.com/support/documentation/data_sheets/ds099.pdf

[2] https://www.xilinx.com/support/documentation/data_sheets/ds152.pdf

[3] https://www.xilinx.com/support/documentation/data_sheets/ds181_Artix_7_Data_Sheet.pdf

[4] https://www.xilinx.com/support/documentation/data_sheets/ds191-XC7Z030-XC7Z045-data-sheet.pdf

[5] https://www.xilinx.com/support/documentation/data_sheets/ds183_Virtex_7_Data_Sheet.pdf

[6] https://www.xilinx.com/support/documentation/data_sheets/ds925-zynq-ultrascale-plus.pdf

[7] Gathered from a subset of paths from implementations.

and faster design in the literature.

8.4.3 Lossless-only Encoders Comparison

The Vitis HLS implementation feature was used to estimate the clock frequency that LOCO-ANS6 would achieve in a Virtex 7 -2, used by the lossless reference architecture. Although the resulting pipeline of the lossless only decorrelator is very similar, the maximum frequency obtained after P&R is 120 MHz. The performance gap probably comes from the lower level optimizations applied to the context bias update path, as described in [148] and later improved in [41], which is the frequency bottleneck of our and their implementations.

At first glance, for lossless, LOCO-ANS6 achieves a compressed image 1.1% smaller than JPEG-LS (see section 8.4.2.1), at the cost of throughput. However, the TSG coder can achieve 288 MHz in that device for the 6 ANS configuration. That is, 1.39 times faster than the reference design. Thus, if the Bernoulli and Geometric coder are decoupled

(independent ANS states) and an optimized decorrelator is used, the TSG coder would not be the system bottleneck as, on average, it requires running 1.3 times faster.

In practice, we may find symbol sequences that increase the local mean of Geometric coder iterations, particularly with very noisy images, but this can be countered by decreasing the iterations limit (also limiting code expansion) and increasing the cardinality of the tables (decreasing mean iterations). Additionally, increasing the block size (which also improves compression) and using buffering between the decorrelator and the coder can mitigate the eventual performance throttling.

Finally, note that these positive results arise from comparing an HLS coder implementation with the best performing and carefully designed HDL decorrelator.

8.4.4 Near-lossless Encoders Comparison

To analyze our near-lossless implementation, [40] is used as a reference point. Given that this JPEG-LS encoder does not support the run coder and has a maximum tile size of 32x64, the achieved compression ratio is considerably lower than the JPEG-LS standard. The negative effect of not supporting the run-length coder increases with the $NEAR$ parameter, as lower entropy symbols are generated and the Golomb coder becomes less and less efficient, which can be appreciated in fig. 8.9. LOCO-ANS exhibits the opposite behavior, as the TSG coder is very well suited for near-lossless compression. As a result, LOCO-ANS6 (single lane) achieves 7.0%, 16.2%, 24.5%, and 32.4% smaller output size compared to the near-lossless reference implementation. Using the two lanes in parallel to compress an image widens further this compression gap to 7.4%, 16.7%, 25.1%, and 33.0%.

Regarding performance, the reference implementation decorrelator has two lanes with an $II = 2$ running at 51.68 MHz (25.84 MPixel/s/lane) in a Virtex 6-75t. These lanes share a single Golomb encoder with $II = 1$ running at the same frequency. This performance is surpassed by our implementation, also with two decorrelator lanes with $II = 2$ running at 81.1 MHz (40.55 MPixel/s/lane for photographic images of medium and above size) in a Zynq 7020. However, this reference implementation was designed for 12-bit images, which worsens the two feedback paths that can limit the encoder performance. For this reason, to better compare these two designs, we run an implementation with Vitis HLS, configuring our decorrelator to work with 12-bit images. As the newer tool set starting from Vivado (almost 10 years old) does not support devices prior to the 7 series, the low-end Zynq 7020 (with the lowest speed grade) was targeted as opposed to the higher end Virtex 6. Table 8.5 gives a hint supporting that this decision favors the

reference implementation, as all Virtex 6 timings are noticeably smaller than the chosen target. The Virtex 6 speed grade used in that work is not reported, but this consideration is still applicable to the slowest Virtex 6 as it can be appreciated in the table. As a result, the 12-bit HLS decorrelator achieved a clock of 67.3 MHz after P&R, still a 30% higher throughput.

We attribute this performance increase to the alternative method used to reconstruct the quantized pixel (section 8.2.1.2) and the fixed gradient quantizer. The reference implementation uses the multiplication by inverse trick to implement the division and applies a compensation scheme to correct the errors derived from this technique while using 15 bits for the fractional part. For very deep pixels, this might be more efficient, but in the proposed architecture, using a table, we achieve a greater simplification and reduction of the critical path. For deeper pixels, larger tables would indeed be required. But the needed type of memories are abundant (see table 8.1), and for this case, targeting up to 12-bit images, only 8 36K on-chip memories are required (in the case of Xilinx devices). The performance increase comes at the cost of memory resources, but as it can be observed comparing table 8.1 and 8.3, this resource is not the limiting factor.

Again, as mentioned before, these positive results were obtained comparing an HLS implementation with carefully designed HDL ones. Additionally, as noted in section 8.2, further optimizations are possible. However, for the purpose of this work, the presented module was optimal enough to analyze the LOCO-ANS encoder performance.

8.5 Conclusions

A hardware architecture of LOCO-ANS encoder was described, as well as implementation results presented, analyzed, and compared against prior works in the area of near-lossless real-time hardware image compression.

The presented encoder excels in near-lossless compression, achieving the fastest pixel rate so far with up to 40.5 MPixel/s/lane for a low-end Zynq 7020 device and 124.15 MPixel/s/lane for Zynq Ultrascale+ MPSoC. At the same time, a balanced configuration of the presented encoder can achieve 7.4%, 16.7%, 25.1%, and 33.0% better compression than the previous fastest JPEG-LS near-lossless implementation (for an error tolerance in [0..3], respectively).

In this way, the presented encoder can cope with higher image resolutions or FPS than previous near-lossless encoders while achieving higher compression and keeping encoding latency below 100 μs. Thus, it is a great tool for real-time video compression

and, in general, for highly constrained scenarios like many remote sensing applications.

These results are in part possible thanks to a new method to perform the pixel reconstruction in the pixel decorrelator and the high-performance Two-Sided Geometric coder, based on tANS, which increases the coding efficiency. Moreover, as mentioned throughout the article, it is noted that further optimizations of the presented system are possible. Finally, experiment results support that if used with the fastest lossless optimized JPEG-LS decorrelators in the state-of-the-art, this coder will improve compression without limiting the encoder throughput.

CHAPTER 9

100GbE Flow Metering & Dual Read-Update Architectures

In this chapter, we address the problem of designing an architecture to efficiently carry out two read-update processes dealing with significant propagation delays associated with the memory system. The resulting architecture enabled to completely offload a 100 GbE VPN volumetry application. Also, it allowed to implement a TCP flow metering system which reduces the output packet rate and bandwidth to 40.6% and 45%, resp, using a single core. Arrays of these, increase performance further. In this way, we demonstrated the FPGA capabilities for processing 100GbE traffic with state (as opposed to stateless, packet-wise processing) and also proposed an architecture to improve performance of systems with the read-update pattern, specially when numerous on-chip memory blocks need to be cascaded.

This chapter is based on the work published in [64]: Alonso, T., Ruiz, M., Sutter, G., López-Buedo, S., & López de Vergara, J. E. (2019, April). Towards 100 GbE FPGA-Based Flow Monitoring. In 2019 X Southern Conference on Programmable Logic (SPL), Buenos Aires, Argentina (pp. 9-16).

9.1 Introduction

As exposed in chapter 5, implementing the processes that handle the flow table in an FPGA flow metering offload system for 100GbE presents a real design challenge. The system needs to be able to perform up to 148.8 million flow table updates per second, while, at the same time, it has to scan the table looking for expired flows. On top of this, large propagation delays from and to the on-chip memories expected given their required size (in the MB order), and the processing logic can be complex due to, not only the entry update operations, but also the need of handling multi-way caches.

Although small compared to commercial memory chips, FPGA on-chip memories (SRAM based) provide very high bandwidth and very low latency (one to a few cycles). In general, these memories come in tens to hundreds Kbit blocks distributed inside the FPGA chip, which can be combined to create larger memories. However, this combination comes with performance penalties due to additional routing delays. Also, there are limits on how to combine the memories, and in some cases, involves the use of FPGA fabric (slower than dedicated resources, and thus increasing performance penalties). In addition, given that these modules are distributed, when using large memories, larger data routing delays to and from the data operators are experienced. Therefore, careful study of the most suitable architectures is fundamental to maximize the system performance.

9.2 Architectures for the Flow Metering Core

Table 9.1 introduces notation for relevant FPGA logic times, along with their magnitudes for a high-end (target) and a low-end technology. The focus is on Xilinx devices as our target platforms are the VCU118 and Alveo200, but these values are, in general, in the same order of magnitude for equivalent resources of similar process technology.

Notice that flip-flop setup and output propagation times are (in particular the later) significantly smaller than those of memories. In Virtex US+ (target), memories are more than 10 times slower. Also, as previously mentioned, when cascading [1] memories, delays are increased. The additional output delays are approximately $240 \cdot (N-2) + 140$ and $180 \cdot (N-1)$ (in ps), for BRAM and URAM, resp., where N is the number of cascaded memory blocks. URAMs also present additional input delays, which are $\approx 165 \cdot (N-2) + 215\,ps$.

Because of these penalties, although BRAMs have half the output propagation time, cascading them to URAM equivalent size, would end-up with larger delays. So, even for

[1] The term *cascading* is used given that in the target device these memories are not connected in a balanced tree structure but in a lineal chain

Table 9.1: Notation used in the architectures analysis

		Magnitude (ps) [4,5,6]	
Symbol	Meaning	Virtex US+ -2	Artix 7 -1
T_{Mp}	Memory output propagation time [3]	≈ 1000,≈ 2000	2460, -
T_{Ms}	Memory data in setup time	≈ 270, ≈ 300	240, -
T_{FFp}	Flip-flop output propagation time	80	≈ 465
T_{FFs}	Flip-flop setup time	25	≈ 100
T_{comb}	Combinational logic propagation time	Dep	Dep
T_r	Routing delay of the chosen path	Dep	Dep
T_{margin}	Clock skew[1] + clock uncertainty[2]	Dep + (55-90)	Dep + (55-90)

Dep: Dependent on logic and/or P&R
Memory values: left corresponds to BRAM and right to URAM. Artix 7 does not have URAM.
[1] Clock skew: Difference between launch and capture clocks paths delays. Clock skew depends on the implementation, and larger modules favor larger clock skew. Values in the 100 to 400 ps range were common in the critical paths of the flow processor implementations.
[2] Clock uncertainty: Clock jitter and phase error component, dependent on clock sources.
[3] Without output register
[4] `https://docs.xilinx.com/v/u/en-US/ds181_Artix_7_Data_Sheet`
[5] `https://docs.xilinx.com/v/u/en-US/ds923-virtex-ultrascale-plus`
[6] Gathered from a subset of paths from implementations.

the VPN volumetry application, where only 2^{12} flows are to be stored, URAMs are the best memory resource to implement the flow table. Because of this, in the rest of this chapter, URAMs are used to implement the flow table, although most of the analyses are applicable to any memory.

Both BRAMs and URAMs have additional dedicated pipeline registers that allow higher operating frequency, at the cost of increased write/read latency. Due to the presence of RAW dependencies, this additional latency will usually result in lower system throughput. Then, whether to use these registers and how to use them has to be carefully considered.

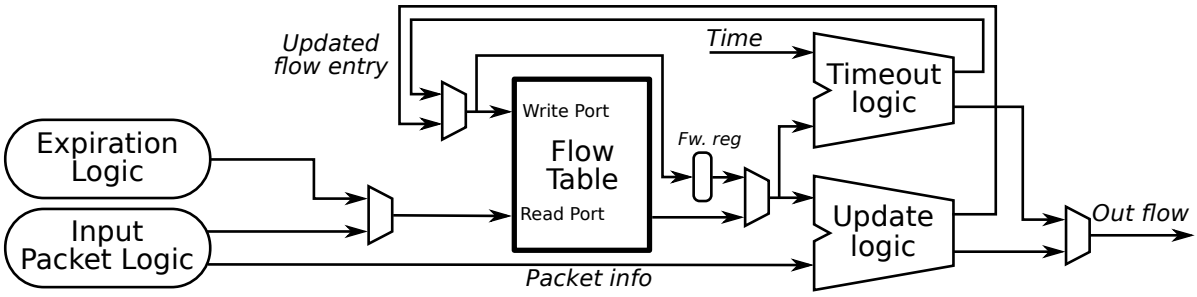


Figure 9.1: Sequential flow metering architecture

9.2.1 Sequential Design

The 10 GbE flow metering hardware in [149] (code available at [150]) runs the update and timeout check processes sequentially. We implemented an optimized version from scratch using Vitis HLS, illustrating in fig. 9.1 the implement architecture, which uses data forwarding logic to achieve an II = 1 [2]. Here, every update operation is followed by an expiration check. Assuming each of these operations takes one cycle (best case), II = 2, and thus, the design needs to run at twice the input packet rate, 297.6 MHz.

$$Clock\,period = T \geq T_{Mp} + T_{comb} + T_r + 2 \cdot T_{mux} + T_{Ms} + T_{margin} \tag{9.1}$$

To understand whether it is possible for it to achieve the target frequency, we analyzed the critical path. Eq. 9.1 shows the generic clock period constraint, where T_{comb} and T_r can belong to either the update or expiration logic and T_{mux} is the additional delay due to the multiplexers. Notice that, just T_{Mp} accounts for 60% of the time budget, and therefore it is not likely that this architecture will achieve the target frequency.

9.2.1.1 Dynamically Scheduled Sequential Design

We may consider trading expiration accuracy in favor of a lower frequency requirement. We can employ non-blocking reads of the input packet stream and use time slots without new packets to run timeout checks. The expiration accuracy degradation is not fixed, but dynamic (function of the load level). This does not imply that the T.O. check process would be starved, quite the opposite. E.g. if the design achieves 223.2 MHz, then it is guarantied that at least 1 in 3 cycles would be used for T.O. checking. This provides a new degree of flexibility to the design. Implementations requiring more complex logic and larger memories would achieve lower frequencies, and thus, a higher tendency of degrading expiration accuracy.

The VPN volumetry problem con be implemented in this way, given that only 4Ki flow (2^{12}) are to be tracked and exported once a second, requiring the system to run at $\approx 148.8MHz$. For this application, to increase the measurements' precision while simplifying the module, an HLS implementation could run an update-only loop until the timeout flag is asserted (every $1s$), and then going to a memory flush loop, after which it starts again[3]. Running at the min. frequency, a packet buffer of at least 4Ki is required.

[2] This architecture would not need the forwarding logic if the tool took advantage of the fact that if using port A of the URAM to write, new data appears in the read port when an address collision occurs

[3] An example HLS implementation using this structure can be found in: `https://github.com/hpcn-uam/hls-conditional-stalling/blob/46ed2bc0d3d7e52ec514449465bb8e05078ed853/HLS/`

Implementing the TCP flow metering core with this architecture is also possible, but only for small memories and limited associativity. Our HLS prototype reached the required frequency up to a memory of 16Ki flow and 4 ways. However, to be able to use larger memories (achieving a greater offloading) and reach higher update rates, a different architecture is required.

9.2.2 Double-Frequency Multi-Cycle Architecture

Fig. 9.2 shows the block diagram of an alternative, higher performance architecture, while fig. 9.3 shows its temporization. It runs using a clock twice as fast as the required packet rate, but most of the logic operates as if the frequency was two or three times lower, as it uses multi-cycle paths. A register enable (EN) signal, switching at half the clock frequency, synchronizes most of the sequential logic, so the core works as if two clocks were used. Also, this signal indicates whether the memory can perform either a read (EN = 0) or write (EN = 1) operation on its ports.

The update process has an II = 2 and is composed of two pipeline stages: memory pre-fetch and execute. The core may accept new packets when EN is asserted, reading the memory in the following cycle. After that, the memory data appears in the output, which is then registered (EN = 1). This registered data is operated by the combinational update logic, executing the write operation after two cycles. As the pre-fetch stage may have accepted a new packet before writing the updated data back to memory, data forwarding logic is implemented (see fig. 9.2).

The exporter logic performs the timeout checks are done using the remaining memory port in parallel. Given that it is a less critical process, it does not pre-fetch the flow table, so the process sequence is completed in 4 clock cycles (II = 4). First, the *exporter control*

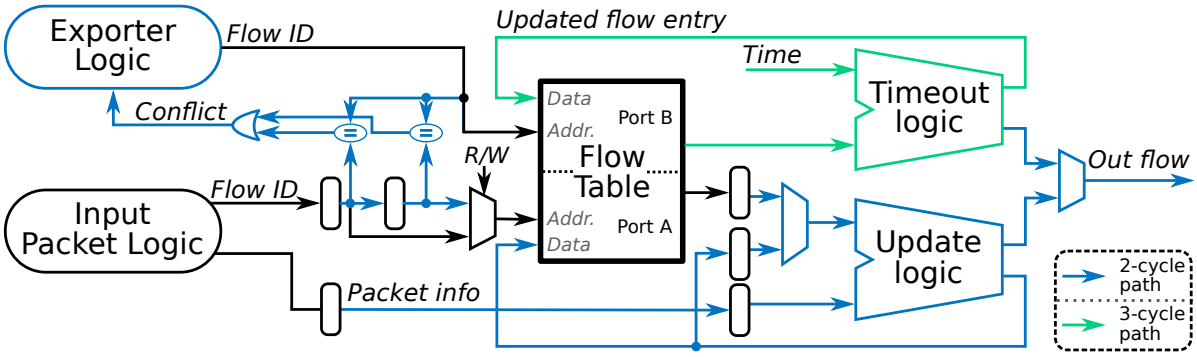


Figure 9.2: Double-frequency multi-cycle architecture block diagram.

Conditional_stalling_stage_implementation/core.cpp#L43

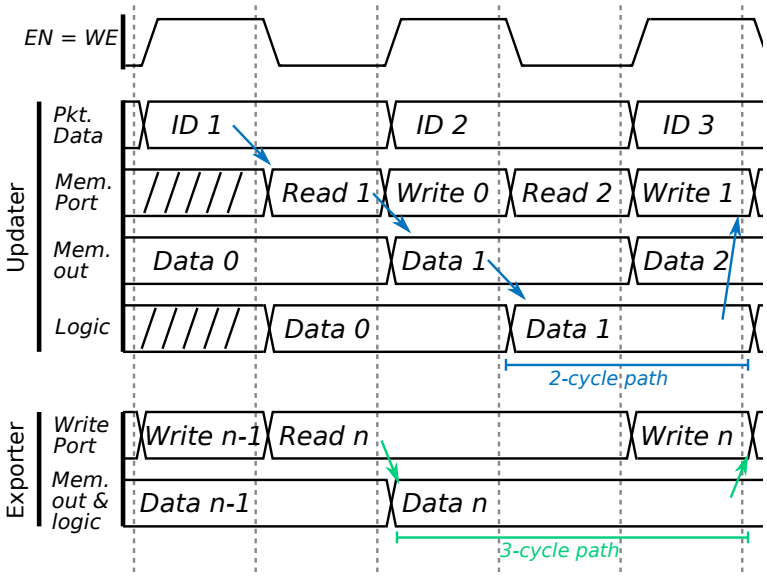


Figure 9.3: Double-frequency multi-cycle architecture timing.

logic addresses the memory (EN = 0), then the output data is directly operated using combinatorial logic during the following 3 cycles. During the last cycle (EN = 1), if the entry is expired, the logic clears the memory entry and outputs the flow record.

As the update and timeout check processes run in parallel, data corruption may occur if both operate upon the same address. For this reason, the addresses in the update process pipeline are available to the exporter control logic, which has two cycles to compare them. If one of the addresses is equal to the exporter's, the exporter sequence is cancelled, and it stays in the reading state until the hazard disappears. Because the processes share these signals, it is important that their read and write cycles are synchronized (see fig. 9.3) so that conflict detection logic can make use of 2-cycle paths.

9.2.2.1 Critical Paths Analysis

The most critical paths are present in the update pipeline, either from the memory data output to its register or from this register to the memory inputs. The clock period constraints corresponding to these paths are shown by eqs. 9.2 and 9.3, resp.

$$Clock\,period = T \geq T_{Mp} + T_r + T_{FFs} + T_{margin} \tag{9.2}$$

$$2 \cdot Clock\,period = 2 \cdot T \geq T_{FFp} + T_{mux} + T_{comb} + T_r + T_{Ms} + T_{margin} \tag{9.3}$$

Although compared to the sequential design this architecture requires a clock twice as fast, it allocates 3 clock periods for the update process. Notice that the memory output

pipeline registers split apart the large memory output propagation time and the update logic delay. Data coming out of the memory is registered, thus masking T_{Mp} delay. There are three positions where this register can be placed: inside the memory block (using the dedicated registers), and using fabric registers before or after the data forwarding multiplexer (fig. 9.2 shows the second option). The optimal position depends on the actual implementation, should try to balance eqs. 9.2 and 9.3, to minimize T [4].

Independently of the memory configuration and logic, the time budget is increased for a given packet rate. Consider the case where the memory data is registered inside the memory block. With this configuration, the memories are capable of achieving very high frequencies, and thus, it is not likely the path constrained by eq. 9.2 will limit the operating frequency (for the target device, URAMs can reach 600 MHz and BRAMs 737 MHz). Then, comparing eq. 9.1 with eq. 9.3, we see that this alternative architecture achieves more than twice the sequential design frequency. All the delays terms in these equations are the same, except that eq. 9.1 contains T_{Mp} ($\approx 2000ps$ for URAMs), while eq. 9.3 has T_{FFp} ($\approx 680ps$ when the block memory register is used). For the TCP flow metering application, having the register outside the memory and before the multiplexer optimized the clock frequency as not only T_{Mp} was masked by the extra 1-cycle phase, but also the memory output cascade delays (if present) and some routing delay towards the update logic. Also, in this way, $T_{FFp} = 80ps$, given that fabric registers are used.

In addition to this, for a given maximum packet rate, the architecture can perform up to 50% more operations, as the timeout check process runs in parallel with the table update one. The former process uses 3-cycle paths to detect expired flows, and decide whether to clean the memory entry and output the records. This schedule allows the more complex and time-critical update process to have preferential access to better nets and resources, and thus, enabling further frequency increases.

9.2.2.2 Pipelined vs Multi-Cycle Logic

Instead of using multi-cycle paths, we could employ pipelining (option chosen by Vitis-HLS when $II > 1$). However, for statically scheduled logic[5], multi-cycle paths are preferred because:

- Pipelines are rarely perfectly balanced, penalizing the maximum allowed frequency.
- Each additional pipeline stage adds register's setup and propagation times.

[4] The *retiming* netlist optimization moves registers to optimize a particular implementation

[5] As opposed to dynamically scheduled logic that may increase mean throughput at the cost of decreasing worst-case throughput (see chapter 10)

- Clock skew harder to minimize as the number of sequential elements increases.
- Multi-cycle paths have lower resource requirements (no additional registers).
- Multi-cycle paths should consume less energy given its lower area and toggle rate.
- Multi-cycle paths may allow logic optimizations that would be otherwise hindered by the pipeline registers, leading to further frequency and area improvements.

9.3 Flow Metering System Implementation

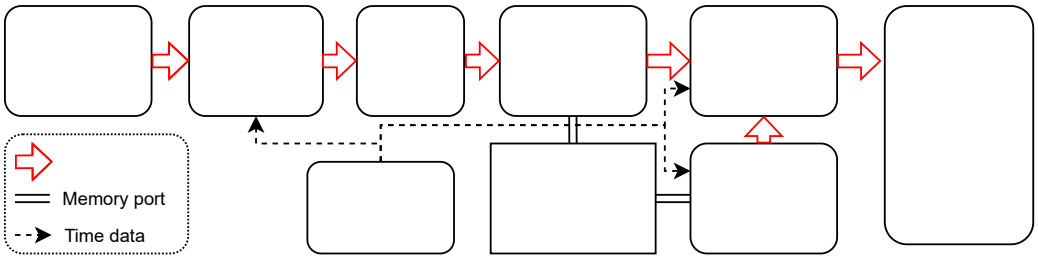


Figure 9.4: High-level diagram of flow metering system

The system has a dataflow architecture to maximize the processing rate and take advantage of FPGA capabilities. The first stage is the *Parser*, which filters packets and extracts information from them. Then, the *Hash* function computes the address of the flow table, where the records are stored. For each new packet, the *Data Updater* creates or updates the corresponding flow entry. The *Exporter* continuously examines the table for expired flows, which will be sent to the output interface.

9.3.1 System Description

9.3.1.1 Parser

The parsers have a 512-bit wide bus input, which provides aligned packets (each bus transaction corresponds to only one packet). This is done to reduce the hardware complexity, but carries the problem of bus underutilization, reducing the effective throughput. To compensate this, the bus requires a higher transaction rate, resulting in a minimum frequency of 294.1 MHz to support 100 GbE under all conditions.

For the TCP flow metering application, a pipelined RTL was designed while, for the VPN volumetry application, the SDNet compiler was used [151] to contrast the two approaches. With SDNet, the description of the parser is done using a specific C++-like

high-level language, which allows extending easily the system capabilities. Fig. 9.5 shows a code snippet, where the top-level module is defined and the parser described. Although the SDNet parser had a higher footprint, it enabled to implement a more sophisticated parser (regarding protocol support) in considerably less time. The extra area was not impactful overall, given that other modules, like the 100 Gbps interfaces and medium access control, have significantly larger resource requirements. Thus, given that it achieved the required operating frequency with the first implementation, it greatly increased our productivity. Conversely, given that the focus of this work was on the flow

```
//Top level
class Parser_v0 :: System {
  PktIn    in_pkt_stream;
  PktOut   out_pck_stream;
  MetadataOut out_metadata;

  Parser_v0_eng parser; //inst. of ParsingEngine

  method connect = {
    parser.packet_in = in_pkt_stream,
    out_pck_stream   = parser.packet_out,
    out_metadata     = parser.tuple_out
  }
} // Parser_v0

//Definition of class InternalData :: Tuple

class Parser_v0_eng::ParsingEngine(48000,3,ETH){
  //48000:Max. pkt size | 3:number of layers
  // ETH: definition of 1st protocol
  // Constants Definitions here
  MetadataOut tuple_out;
  InternalData internal_vars;

  // Ethernet protocol definition
  class ETH :: Section(1) {
    struct {
      mac_dst : 48,
      mac_src : 48,
      tpid_0  : 16, //Tag Protocol Identifier

      // Optional first VLAN
      pcp_1   : 3,
      dei_1   : 1,
      //more VLAN fields here

      // Other fields ....
    }
```

```
    // class ETH continues ...
    // Extract metadata from ETH header
    method update = {
      tuple_out.first_vlan_id =
          if (tpid_0!=VLAN) 0
          else  vlan_id_1
      tuple_out.etherType =
            if      (tpid_0!=VLAN) tpid_0
        else if (tpid_1!=VLAN) tpid_1
        else                   tpid_2
    }

    // Move to following protocol.
    // Up to two VPN tags supported.
    // ETH can be followed by IPV4 or IPV6.
    // Need to check etherType field
    map map_eth_tp {
      (IPV4_TYPE, IPV4),
      (IPV6_TYPE, IPV6),
      Others_L3  //The rest are ignored
    }
    method move_to_section =
          if(tpid_0!=VLAN) map_eth_tp(tpid_0)
      else if(tpid_1!=VLAN) map_eth_tp(tpid_1)
      else                  map_eth_tp(tpid_2);

    // Set position of next protocol (in bits)
    method increment_offset =
        if (tpid_0!=VLAN) 112
    else if (tpid_1!=VLAN) 112 + 32
    else                   112 + 32*2 ;

  } // End of ETH class
   //Definition of other protocols: IPV4, ...
} // Parser_v0_eng
```

Figure 9.5: SDNet parser code snippet showing the top-level module *Parser_v0* and parser engine specialization for the task, where it is defined how to handle the incoming packets. As an example, the snippet show the code for the first layer protocol, Ethernet.

metering core, the finally implemented RTL parser had basic protocol support.

9.3.1.2 Flow Metering Processes

The double-frequency, multi-cycle architecture described in section 9.2 is used to implement the processes that handle the flow table. The architecture structure and timing is agnostic of the stored data and the update operations, thus from application to application only the combinational logic that handles the data needs to be adapted (Update logic and Timeout logic in fig. 9.6). This allowed to implement the VPN application easily once the TCP system was developed.

9.3.1.3 Flow Table

For the VPN volumetry application, up to 2^{12} flows, classified by the VLAN identifier (VID) of the first VLAN tag, are to be tracked concurrently. Thus, the Hash module (fig. 9.4) simply selects the VID as the table address. For the TCP flow metering application, on-chip memory cannot store all the possible concurrent flow, and therefore the Hash module computes a value within the address range based on the 4-tuple that identifies the flow. Contrary to the VPN application where addresses contained the records of a single flow, in this case, each memory address can store more than one flow entry, distinguishing each entry by ways as an N-way associative cache memory. Thus, the effective memory capacity increases, reducing the probability of flow collisions, as it was observed in the simulations in section 5.4.2. Either BRAMs, URAM, LUTRAM or a combination of these can be used to implement this memory, but in both tested applications, URAMs were chosen given the capacity requirements. In both cases, 64 bytes entries were used, leaving room for more records.

9.3.1.4 Data Updater

To ease the implementation of N-way tables, the update logic is divided in two modules: the entry data update and the decision logic.

Entry Updates The entry data update is replicated for each entry, producing an output that assumes that the corresponding way is to be updated. That is, the logic checks if the flow ID matches, and if it does, updates the records, and if not, creates a new flow entry. To use all the 64 bytes of the entry and to stress the implementation further, bidirectional TCP flows [152] are created in the table, comprised of those streams that connect the

same points but have opposite direction (IPs and port numbers inverted). For this, it was necessary to select a hash function that generated the same address independently of the packet direction.

Multi-Way tables decisions The decision logic determines which table way is written (generates each way's write enable signal, WE) and whether flow records are exported. Regarding the VPN application, this module is not necessary, as there is a one-to-one relationship between flows and addresses. For the TCP flows, if there is a flow ID match, the entry is updated, if not, the logic checks whether empty slots are available. When, all entries are full and there is no ID match, a flow collision is produced, so one of the stored entries is removed and sent to the output.

Collision resolution policies The policy that produced the best results in terms of output packet rate and bandwidth, previously described in section 5.4.2, first considered TCP FIN and RST flags, and then, the timestamps of the last packet each entry received. This decision is the function that requires the most logic levels, so the most susceptible to be the slowest path of this stage. Notice that the aleatory export policy is not a good option. Despite being fast, simulation shows a more than a 100% increase in the output packet rate compared to the selected policy, i.e, for 65K and 8 ways, the input to output ratio goes from 0.406 to 0.85. To detect the oldest entry, keeping them sorted could be an option, but this implies a more complex data routing. In general, memory data outputs of a given x way would need to be connected to the data inputs of ways 1, x and $x+1$, where way 1 is where the new updated flow is stored. Although this simplifies both the expiration and the collision resolution processes, each of these data buses (per way) have a width in the order of 500 bits, requiring lots of routing resources to and from RAM blocks and resulting in larger routing delays. Instead, the oldest timestamp is searched using an all versus all logic, aiming at minimizing logic levels, requiring $(N^2-N)/2$ parallel comparisons. Given that increasing the ways above 8, does not significantly reduce the collision probabilities, the number of operations is acceptable.

9.3.1.5 Exporter

This module scans the memory using the second port of the table, comparing the timestamps against the current time, looking for expired flows and exporting them. Since the timestamps will eventually overflow, the comparison for serial numbers defined in the RFC 1982 section 3.2 was used [153]. The module registers the time source data, and

computes the two expiration times (the corresponding timestamps older than these, are expired). The generated expiration times, are only registered in the clock edges where a read operation takes place, therefore is registered when the enable signal is low (as opposed to other registers). This is done so that the expiration detection process uses 3-cycle paths, as previously described when analyzing the architecture.

9.3.1.6 Output Interface

Merging the output streams When a collision is detected, all entries are sent to the output plus the generated WE signal, so that the following stage selects the entry to export based on it also using a 2-cycle path. Then, in the next stage, this stream of flow records is merged with the one coming from the exporter.

Timestamp Extension The final stage allows using shorted timestamps internally while exporting long ones, thus leaving more room for other flow data given an entry size. This task of extending the timestamps can be performed provided that the internal timestamp is not too small to confuse an expired flow (because of the active timeout) with one that is not.

Flows destination In the case of the TCP flow metering application, the exported flows were sent using a second 100 GbE link, while in the VPN one, the flows were sent to the main memory of the host computer using the PCIe bus.

9.3.2 Implementation

9.3.2.1 Implementation of the Double Synchronous Clocks

The intention of the flow metering core design is that all logic, except for some memory signals, run at the packet rate (low clock frequency). This is achieved using an enable signal that toggles at half the frequency of the fast clock and multi-cycle paths, instead of using two synchronous clocks, one at half the frequency of the other and with a 0° phase difference. From the standpoint of the design and HDL description, these alternatives are basically identical. Although the latter also has multi-cycle paths, constraining the former requires more of these timing exceptions, given that synthesis tools do not detect the multi-cycle paths, so it has to be explicitly indicated.

Yet, the former option was chosen for performance reasons. The use of the enable signal could be considered problematic at first glance given that it controls numerous

flip-flips, resulting in a large spanning, high fan-out net, but this signal can be easily replicated, solving these problems. Conversely, although the Ultrascale+ devices have clock divider buffers that allow to efficiently obtain the two required clocks, this alternative results in a higher clock skew and uncertainty. In an implemented proof-of-concept design using the double clock alternative, clock skew of the critical paths was in the $550ps$ order, and clock uncertainty in the $87ps$, compared to the $300ps$ and $57ps$, resp., of the final core implementation. The number of clock-crossing paths used in the proof-of-concept was much lower to what the final system required. Given that clock skew is harder to optimize as this number increases, further frequency penalties are to be expected in a full design implementation. These time differences might seem small, but they are above the 8% of the time budget of the maximum allowed period of the high-frequency clock ($3.333ns$).

Finally, notice that, the enable signal alternative is, in fact, using two clocks, given that, in the target device, flip-flip enable inputs do not act in the data input, but they gate the clock, effectively creating the required half-frequency clock. Having the clock gate at the input of the sequential elements, allow the launch and capture clock to share a larger common path, and then, clock skew is reduced.

9.3.2.2 Deployment and Testing

The TCP flow metering system was implemented on the VCU118 Evaluation Board [154], while the VPN volumetry one in the Alveo U200 acceleration card [155] (both using the Xilinx VU9P FPGA, speed grade 2). To verify the correctness of the systems, simulations and field tests were carried out both using synthetic traffic aiming to check specific parts of the system, and then, a more realistic work regimen was tested using captured traces.

9.3.2.3 Results

Table 9.2 shows the implementation results for different system configurations. As it can be appreciated, the design can support the maximum packet rate up to 64Ki flow using 8 ways. Although, it did not meet the desired timing, the 128Ki flow implementation achieved a significant performance (above 90% of the max. rate), which is very interesting considering that the core is implemented in a single SLR using 80% of its URAMs.

Conversely, the HLS prototype using the sequential architecture reached the required frequency up to 16Ki flow and 4 ways. What is more, for 64Ki and below, implementations achieve frequencies significantly beyond the required 297.6 MHz. This supports the fact

Table 9.2: Flow metering system implementation results for both VPN and TCP metering applications for a series of configurations

Flows	Assoc.	Freq. (Rate[1] %)	LUT (K)	FF (K)	BRAM	URAM
4Ki	1	300 (101) [2]	4 (90.1)	6.8 (154.8)	1 (265)	11 (11)
32Ki	4	352 (118)	**9.5** (42.0)	**10.5** (99.1)	0 (77)	64 (64)
32Ki	8	**353** (119)	18.2 (50.8)	16.7 (105.3)	0 (77)	64 (64)
64Ki	4	313 (105)	**9.6** (42.2)	**10.6** (99.3)	0 (77)	128 (128)
64Ki	8	**325** (108)	17.7 (50.3)	16.6 (105.2)	0 (77)	128 (128)
64Ki	16	285 (96)	43.6 (76.2)	21.1 (109.7)	0 (77)	128 (128)
128Ki	8	271 (91)	18.1 (50.6)	16.6 (105.3)	0 (77.0)	256 (256)
VU9P totals			1182	2364	2160	960

For a given table size, if there are multiple implementations, the best results are in bold. Resource utilization is shown for the subsystem that performs parsing and flow metering, and in parentheses, the implementation totals.

[1] Maximum packet rate supported by the implementations, compared to the maximum 100GbE packet rate (148.8 Mpps).

[2] This implementation corresponds to the VPN application. Higher frequencies were not tested.

that the architecture is suitable for the implementation of a flow metering offloading system using external memories.

Most of the critical paths were between the update logic memory data output and its register (implemented using CLB flip-flips). Nonetheless, the 2-cycle path that generated the WE signal (decision logic) was also problematic. Other positions of the memory data register, that is, using URAM output registers or after the forwarding multiplexer, produced lower performing implementations.

It is an interesting coincidence that, in general, using 8 ways resulted in the higher frequency systems, at the same time that, in simulation, increasing the ways above this number did not enable significantly higher offloading performance. The effect of the number of ways on frequency is mainly explained by the way URAMs cascade to create larger memories and the behavior of the decision logic. For a given table size, fewer ways implies more cascading. As previously explained, this adds output and input delays to wires going to and from the memory, which particularly affects the single cycle paths towards the output data register. This is why, for the 64Ki flow configuration, going from 4 ways to 8, increases frequency. However, as the number of ways increases, the decision logic depth increases. The additional delay more than compensated the decreased memory input delay (less cascading) when going from 8 to 16 ways for a 64Ki flow table, as the critical path of the latter resided in the decision logic.

The table also shows resource utilization of the subsystem that takes packets and

outputs flows, and in parentheses, the implementation totals. URAM usage is directly determined by the table size, but the rest of resources consumed by the subsystem are mainly a function of the number of ways. Apart from the expected heavy URAM usage, the flow metering subsystem requires a small percentage of the available resources in the target platform, also shown in the table. The target resources are evenly divided in 3 SLRs, but even compared to the resources of a single SLR, footprint is low.

9.4 Application of the Architecture for Other Systems with the Read-Update Pattern

Although the proposed architecture was presented in the context of flow metering, it can be used in other streaming-data processing cores, and in general, loop-based processes that have the read-update compute pattern. The presented results show that the proposed architecture is adequate to for high-performance implementations of the read-update pattern using complex logic, even with large memories.

Double-Process Histogram Computation An example of an algorithm that uses the same pattern is histogram computation, which is also the base of other algorithms like the Hough transform (used to detect lines in an image). Fig. 9.6 shows the proposed architecture implementing it, with adapted forwarding logic, allowing to process in parallel two input samples at a time when these differ in value from each other[6]. Given the simplicity of the logic, in the illustration, the memory output data register are used, which reduce the BRAM output latency from $1.02ns$ to $0.29ns$.

LOCO-ANS LS Another example algorithm that can benefit from using the proposed architecture is the presented lossless LOCO-ANS image encoder, whose performance hardware implementation is throttled by the context update RAW dependency. The compiler implements the developed HLS encoder using the analyzed sequential architecture, whose update path establishes the critical path. The utilization of the double frequency multi-cycle architecture would not only enable higher operation frequencies, but also it allows the use of the spare memory port for other processes. In this way, it is possible to encode a second pixel in parallel when the context address differs (like in the previous histogram example), further increasing the system's pixel rate or to prepare the context

[6]Notice that if the samples have equal value, the update logic could be flagged and increment the accumulator by 2 instead of by 1 using one of the processing branches

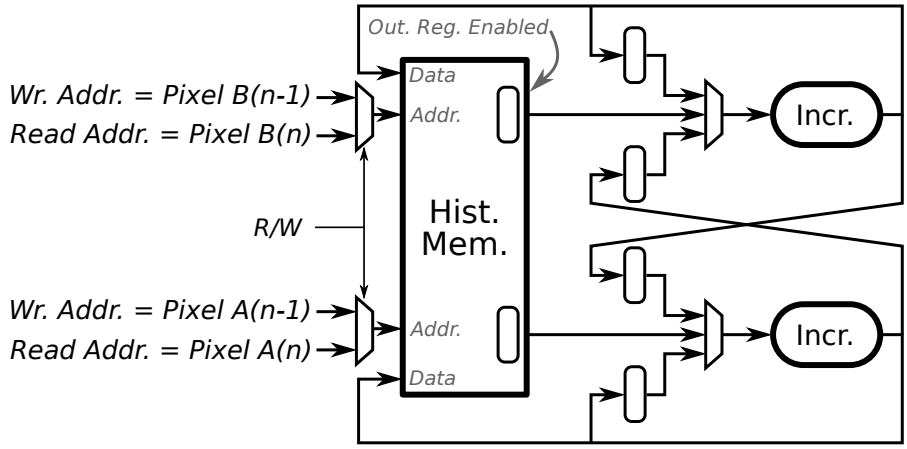


Figure 9.6: Double-process histogram computation using the proposed architecture

memory for the next image block (using a ping-pong scheme within the block memory) enabling continuous processing and reduced encoding latency.

Although, originally, we aimed at designing the flow metering system with HLLs, we reached the conclusion that, due to the presence of the studied RAW dependency, the flow metering core needed to be designed using at RTL. Based on the exposed results, the incorporation of the presented architecture for high-performance implementations of the common read-update pattern would increase the quality of results of C++ HLS designed modules containing this pattern.

9.5 Architecture Improvements

Analyzing the implementation results, we observe a series of possible paths towards achieving even greater performing architectures for FPGA flow metering, in particular, and for the read-update pattern, in general.

9.5.1 Arrays of Flow Metering Cores

As observed in table 9.2, smaller tables allow higher operating frequencies and apart from URAMs, the system has a low footprint. Then, for the TCP application instead of having a single metering core, we may consider having several handling smaller tables (URAM usage would be the same). Parsing would take place, then based on the hash value, packet are distributed among the cores, and finally, their output flows are merged into a single stream. This scheme simplifies routing and reduces the URAM cascading, therefore enabling the use of larger memories at a given frequency. In the

128K implementation, routing latencies from memories were in the $\approx 2.3ns$, then $\approx 70\%$ of the available time was spent in those routes, while these delays were in the $600ps$ order for 64K implementations. In addition, it allows to more easily scale the system to process higher link rates or multiple links.

This also interesting for implementing a flow metering solution using the HBM memories recently available in FPGA chips, which have several ports to access it. Then, a memory bank could be assigned to these smaller and faster metering cores, which could access them independently. If one core stalls, waiting for memory data, the rest may continue running, therefore increasing the average packet rate supported.

To test the idea, a couple of proof-of-concept were implemented using a 2x64Ki and

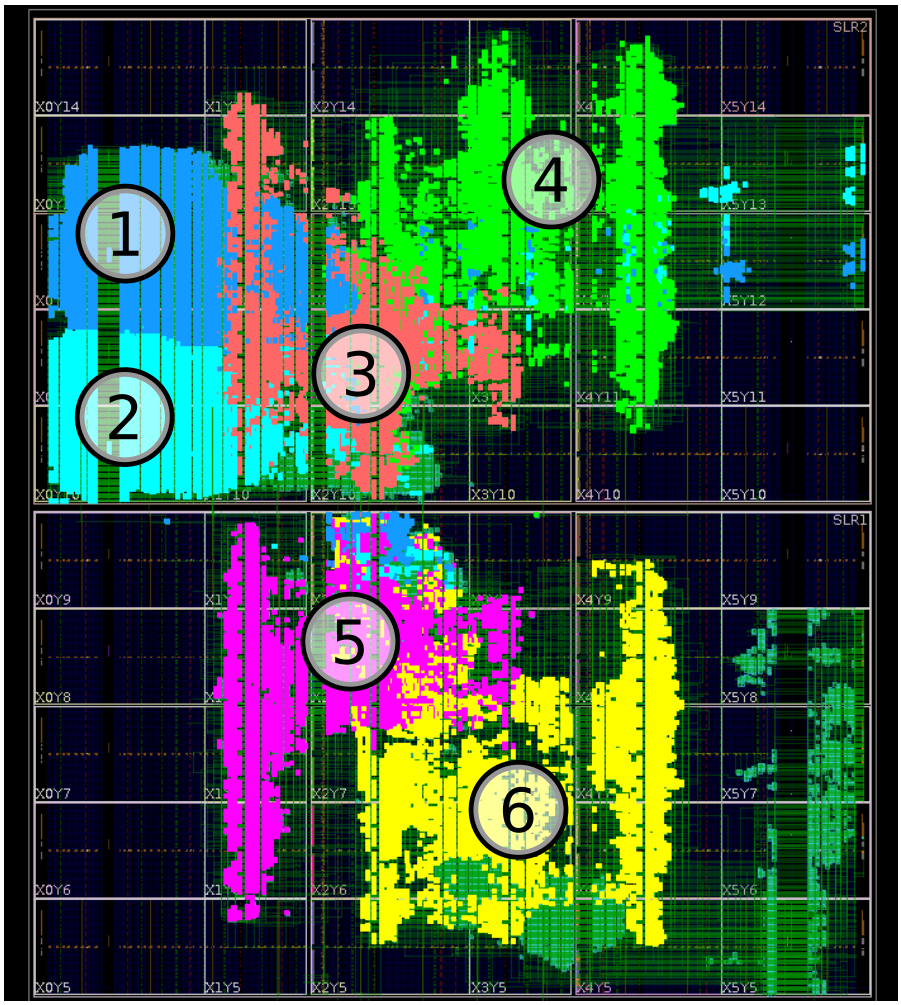


Figure 9.7: Implementation of four 64K flow probs in two SLRs running at 300MHz. The highlighted resource areas 1 and 2 implement the 100GbE interfaces, while areas 3 to 6, the metering cores.

4x64Ki configuration, both with 8 ways. As a result, the 2x64Ki implementation achieved 318 MHz, while the 4x64Ki, 300 MHz. Fig. 9.7 shows the implementation layout of the latter. In both cases, a pair of these cores were assigned to a single SLR, then the former used 1 SLR and the latter 2 of them, as observed in the figure. Compared to the single 128Ki implementation, the 2x64Ki improves the processing rate by more than 17%. These results can be further improved given that neither the packet distribution and flow merging structure, nor the SLR crossings, were fully optimized.

Finally, in the VPN application case, multiple cores allow supporting both several parallel links and higher bandwidth ones. In the latter case, packets would be distributed using round-robin scheduling and each metering core would have partial records of the flow. These partial flows could be aggregated by a downstream module, which does not need a table of its own, given that the application exports records every second, therefore all the metering cores could synchronously emit the partial records of a given flow ID.

9.5.2 Double Frequency Architecture with Increased Pipelining

Analyzing the implemented systems, it is observed that the single cycle path from the memory data output to its registers in the FPGA fabric establishes the frequency bottleneck. Along with this, the WE signal also limited frequency, which was not only due to the complexity of the decision logic, but also the cascade additional input delays and the fact that the WE input has the double setup time that the memory data inputs. Then, taking into account that LUT and flip-flop utilization is low given the available resources of these devices, we consider pipelining the architecture further.

The design presented in fig. 9.8 addresses both the detected bottlenecks of the flow metering core, showing the logic of only one port, and its timing diagram is shown in fig. 9.9. As observed, the system can be described as working with two synchronous in-phase clocks, one (clk signal) twice the frequency of the other (s signal), with only the circuitry in orange (memory and logic controlling its inputs) working at the high-frequency one. Compared to the previous architecture, the memory data output has twice the time budget to be registered (if using BRAMs, these need to be configured in no-change mode), and the update logic input/output signals come from/go to registers, thus avoiding the larger output propagation/set-up times associated with memory blocks. This comes at the cost of increased logic, particularly, a more complex data forwarding unit (which bypasses updated values that have not yet been written to memory), but the additional time budget should be more than enough to accommodate this additional logic.

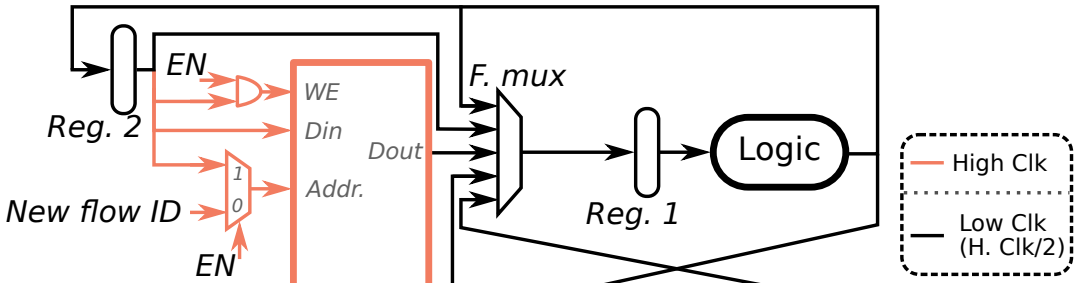


Figure 9.8: Double frequency architecture with increased pipelining block diagram.

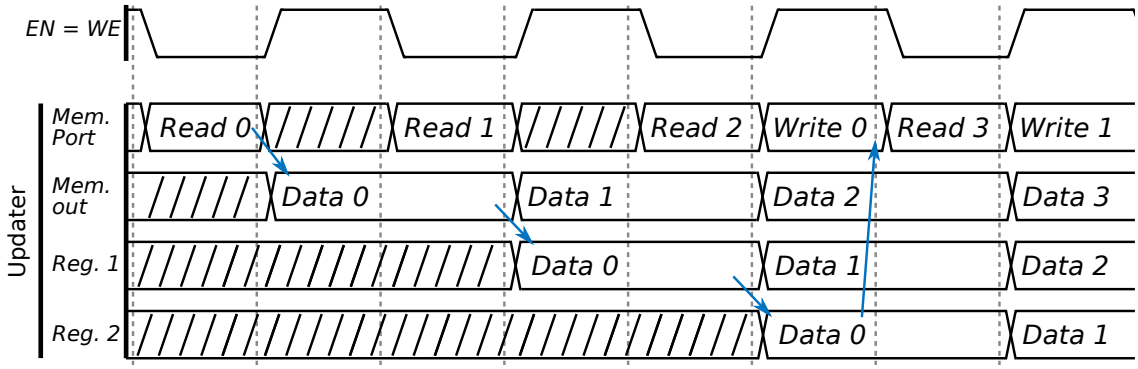


Figure 9.9: Double frequency architecture with increased pipelining timing.

Finally, taking inspiration from the optimized sequential architecture, which dynamically allocated timeout checks, this higher performing architecture could choose whether to process 2 packets, or process 1 packet and do a timeout check. This decision would depend on the availability of input packets and whether the packets need to access to the same memory address or not.

9.6 Conclusion

This chapter presented a hardware implementation of a 100GbE flow metering system for two applications with different memory capacity requirements and logic complexity. For the simpler VPN volumetry application, the designed system can easily meet timing in the target VUP9 FPGA, and it completely offloads the metering task. For the more complex TCP flows application, the system is capable of supporting the maximum 100GbE packet rate with a single core using up to 64Ki flow tables and 8-ways. According to our experiments, this translates into a significant offload of the task, reducing the output packet rate and bandwidth to 40.6% and 45%, resp., with respect to the input packet rate and bandwidth after parsing (already reduced compared to the

link bandwidth).

We also presented preliminary results to pave the way towards higher link rate support and systems using external memories. The implementations of an array of up to 4 of the developed cores in the target devices (2 per VU9P SLR) at 300 MHz show it is possible to use even larger flow tables, increasing the offloading capabilities, while still supporting the maximum packet rate.

These results were made possible by the proposed architecture for the flow metering core, which addresses the problem of implementing a high-throughput and complex read-update process that needs to handle a large memory. For slower link speeds, an HLS implementation could handle maximum packet rates, but given that network links bandwidth grow faster than silicon speed, we needed to implement this critical module using an optimized architecture and RTL code. The presented architectures can be used as general templates and incorporated into an HLS compiler, not only for flow metering systems but to improve performance of systems with the read-update pattern, particularly when many on-chip memories need to be cascaded and more than 2 memory ports are needed.

CHAPTER 10

STUDY OF THE CONDITIONAL STALLING TECHNIQUE

Ambiguous read-after-Write (RAW) dependencies are omnipresent in multiple streaming applications, establishing hard to optimize bottlenecks. Considering actual input data, these may rarely be true dependencies. However, the increasingly used High-Level Synthesis (HLS) compilers must assume the worst-case scenario, as they rely on static optimizations. Conditional stalling is a simple yet impactful technique, useful even when conflicts are common. At the cost of a small area penalty, it allows improving (in some cases, by several times) the mean throughput of these systems. In this chapter, we describe a high-frequency HLS implementation of the technique and examine its behavior as a function of input and architecture characteristics, with the goal of understanding when to use it and how to optimize throughput.

10.1 Introduction

Data dependencies are omnipresent in very diverse applications. As these can be major obstacles towards obtaining a high-throughput implementation, their detection and optimization has been a subject of study for decades [156, 157]. Addresses may be computed at runtime, so it might not be clear whether a memory operation is dependent on another. Static analysis (at compile time) may confirm the presence or not of a *conflict* —when the addresses of these operations are equal, so there is a dependency, limiting

parallelism—, and then, appropriate optimizations can be applied. Additionally, the analysis may reveal conflicts in specific loop iterations. Here, several optimizations have been proposed [158–161], e.g., varying the processing rate depending on induction-variable-based runtime conflict checks.

However, some dependencies are ambiguous at compile-time, so if only relying on static analysis, operations must be scheduled assuming the worst-case scenario. Thus, if the *dependency distance* (DD) is maximum number of cycles separating the pair of dependent operations that still violates the dependency in case of a conflict, then the *initiation interval* (II) —number of clock cycles the logic needs to be ready to process a new input or iteration— will be $DD+1$. We refer to this II as $\mathrm{II}_{\mathrm{base}}$. In many cases, these ambiguous dependencies do not occur very often, thus the hardware will have a considerable amount of unnecessary idle cycles.

For half a century, different compile-time, runtime, and hybrid optimizations have been proposed [157, 162–164]. Yet, most of these techniques have not been incorporated in current High-Level Synthesis (HLS) compilers [165–167], so in recent years, many works have focused on applying them to HLS design and tools. Bypasses from write to read operations (data-forwarding) were proposed in [168] to improve scheduling when Read-after-Write (RAW) dependencies were present. As a result, II can be reduced down to the processing logic latency, eliminating the memory latencies from the equation. Although useful for simple logic, it is not very effective for deeper pipelines.

In [169], a *conditional stalling* (CS) scheme, also incorporated in [170], was implemented in a source-to-source compiler to improve loop pipelining. This technique, inspired by μP architecture, consists in running a pipeline at full rate when no conflicts are detected, while stalling the appropriate stages until those that appear are solved. In this way, $\overline{\mathrm{II}}_{\mathrm{sys}} \leq \mathrm{II}_{\mathrm{base}}$, where $\overline{\mathrm{II}}_{\mathrm{sys}}$ is the average II of the optimized system.

Squash and replay on top of data-forwarding was proposed in [171] to deal with data dependencies. This technique, used in super-scalar μP, consists on speculatively executing an operation and if a conflict is detected later, the dependent operations are suppressed and the pipeline is restored to the stage it was when the violation occurred to replay all operations. When there is a conflict, this technique incurs in penalty cycles, which is not the case for CS, and also, it increases the complexity of the logic, which favors frequency penalties.

We also notice that CS, though simple, is very suitable for stream processing applications, like network packet processing, data compression or data analytics, where high throughput is generally sought, and RAW dependencies are common. Yet, in the work

in [169], previously mentioned, only a modest throughput improvement (approx. 7.5% on average) was observed for the chosen applications over long input sequences, even showing a performance decrease in some cases. Although these results are input dependent, they are in part explained by a 23% (on average) increase in the clock period caused by, as noticed in that work, the stalling control logic. Also, processing logic is pipelined ignoring the input characteristics, resulting (as we will show) in lower throughput.

In this chapter, focusing on stream processing applications, we describe how to implement CS with no or negligible frequency penalties and low area overhead. In addition, with the aim to generate results that can be extrapolated to other applications and implementations, we analyze the technique as a function of the data and logic characteristics, rather than for particular cases. Finally, we provide models that could be employed by compilers to take design decisions. Example systems, as well as the developed simulation and mathematical models, are available through a public repository [172].

```
template <typename T, int DD> struct WaitList {
  T list[DD]; // addresses in the pipeline
  WaitList(T init_val){ //loop should be unrolled
    for (int i= 0; i<DD; i++) list[i]= init val;
  }
  void update(T val){//loop should be unrolled
    for (int i= DD-1; i>0; i--) list[i]=list[i-1];
    list[0] = val;
  }
  bool is_in_list(T val){//loop should be unrolled
    bool found=false;
    for (int i= 0; i<DD; i++) found |= list[i]==val;
    return found;
  }
};
void stall_stage(FIFO<pkt_t> &in, FIFO<pkt_t> &out){
  #pragma HLS PIPELINE II=1
  static WaitList<wait_id_t,DD> waitlist(EMPTY);
  static pkt_t packet;
  static bool conflict = false;
  if (!conflict) packet = in.read();
  conflict = waitlist.is_in_list(packet.read_addr);
  packet.valid = conflict? 0:1;
  out.write(packet);
  waitlist.update(conflict?EMPTY:packet.write_addr);
}
```

Figure 10.1: Stalling stage HLS code.

10.2 Implementation of Conditional Stalling

For stream processing pipelines, the dependency control logic can be implemented as a preceding stage. Fig. 10.1 shows a C++ HLS description of such a stage. In the following, we assume to be optimizing a RAW dependency, as they are more common in these applications, but is also applicable to Write-after-Read ones. `WaitList` stores in `list` the write addresses of the data units to process—namely, *packets*—sent to the output in the last DD cycles. For each packet, to determine whether there is a conflict, the stage checks if the read address matches any of the addresses in `list`. If there is conflict, instead of sending it to the processing stage, the packet is kept until no conflict is detected. During these cycles, *bubble* packets (flagging they must not be processed) are sent to the output, ensuring proper synchronization with the processing stage even if there is intermediate store between them. Alternatively, if there is a tight coupling between these stages, the first stage may not produce any output (the second one must use non-blocking reads).

Given that the stalling stage creates a dependency-free input pattern, the processing module (logic to optimize) can be pipelined as if no dependencies exist, achieving a better II, which we call $\mathrm{II_p}$, ideally equal to 1 to maximize throughput. For HLS implementations, this only involves adding a compiler directive (or `pragma`), indicating that there are no dependencies associated with the memory accesses.

To verify the performance of the stall stage, a float64 accumulation example was developed using Vitis-HLS 2021.1 targeting Xilinx Z7020-1 and ZU7EV-2 chips for different DD and address bit widths (AW). This example was chosen because it allows us to test the technique for a deep, high-performance pipeline. Two versions of the system were implemented, one with the conflict detection logic in a separated stage and the other with the logic merged within the processing logic, as in [169]. Some implementation

Table 10.1: Stall stage HLS implementation performance comparison

Part	Max. freq BRAM	DD = 8 AW=8		DD = 16 AW=16	
		Stage	Merged	Stage	Merged
Z7020-1	400	200	150[1]	165[1]	125[1]
ZU7EV-2	637	635	450[1]	575[1]	395[1]

Frequencies are in MHz and rounded to the closest multiple of 5.
BRAM were configured in read-first mode.
[1] Conflict detection logic in critical path.

results are presented in table 10.1, where it can be seen that implementing the logic as in Fig. 10.1 the system achieves a more than 30% and 40% higher frequency for Z7020 and ZU7EV resp. As a reference, the maximum clock for the on-chip memories (BRAMs) is also shown in the table. In the case of the Z7020, we attribute the larger frequency gap to the slower FPGA fabric of this low-end device. Considering the achieved frequencies, we think that most systems would experience a low or negligible frequency impact when adding the stage in the pipeline. In addition, the stalling stage has a low area impact. For $DD = 8$ and AW = 8, approx. 300 LUTs and 600 flip-flops were consumed, which represents 0.55% of the available resources in the low-end Z7020 device. This value increases to 0.75% (400 LUTs and 800 flip-flops), for $DD = 16$ and AW = 16.

As done in super-scalar μP, to mitigate or eliminate the frequency penalties observed for wide addresses, they may be hashed and then compared to detect conflicts. High-performance hardware hashes exist, so their utilization should not have frequency penalties. Of course, a lower number of operation identifiers (cardinality, C) decreases performance, but it might not be noticeable for high C (see section 10.4).

Although these results are compiler-dependent, they do provide useful information about how to maximize throughput when implementing this technique within either a source-to-source or HLS compiler. Additionally, regarding designing with current HLS compilers, it shows that providing hints in the code about the architecture we aim for is still useful.

10.3 Modeling Conditional Stalling

The $\overline{\Pi}_{sys}$ of a system using CS is a function of the address distribution (data dependent) and DD (architecture dependent), rather than the algorithm itself. For example, consider image-processing applications using pixels as addresses. Scanning 8-bit artificial images has (in general) a much higher probability of obtaining single-pixel-value bursts, producing numerous conflicts, compared to 16-bit raw photographic images.

The analysis is focused in two addresses distributions: the uniform and the Zipfian. The former emerges naturally in many situations while in others by design, e.g., using hashes. In addition, in context-based data compression, contexts are sought to be equally probable to improve compression [173]. Zipf-like distributions also have been observed to characterize classes in different applications [174], e.g., web requests [175], and serves as a skewed probabilities example. To limit the extension of the analysis, we restrict it to *in situ* updates (read address = write address), more common in stream processing.

Results also apply to i.i.d. stateless address distributions.

10.3.1 $\mathbf{II_{sys}}$ Distribution for $DD = 1$

Given a block of W packets with uniformly distributed addresses, we want to get $P(\mathrm{II}_{\mathrm{sys}})$. When $DD = 1$ and the pipeline is full, there are two packet acceptance sequences: the new packet is accepted in the next cycle (S_0) or it waits one cycle and it is accepted in the following one (S_1). It is easy to see that the number of S_1 (N_1) $\sim B(n = W, p = \frac{1}{C})$. Given that the block takes $W - N_1 + N_1 \cdot 2$ cycles to be processed, $P(\mathrm{II}_{\mathrm{sys}} = cycles/W) = P(N_1 = cycles - W)$. From this, it follows that $\overline{\mathrm{II}}_{\mathrm{sys}} = 1 + p = 1 + \frac{1}{C}$.

For the Zipf and other stateless distributions, an approximated model can be obtained setting the binomial parameter $p = \overline{P_c} = \sum^{a \in A} P(a)^2$, where A is the address set and $\overline{P_c}$ is the mean collision probability between two addresses.

10.3.2 Hidden Markov Model for $DD \geq 1$

A Hidden Markov model (HMM) can capture the behavior for general DD. This is only presented for the uniform case, given that it allows a simplification, which makes the size of the model manageable. In general, without this simplification, we consider it is simpler to rely on simulation data.

10.3.2.1 Model

As observed in fig. 10.2, each state captures the occupation pattern of the pipeline —in a binary manner, bubble (0) o packet (1)— after having accepted a packet. Then, states are named ignoring the first stage occupation (always full) and the size of the state set is 2^{DD-1}. For each new packet, there is a state transition, that depends on whether there is a conflict or not, and if there is one, with which stage. Additionally, associated with each transition, there is an observed property, which is the number of cycles required to accept the new packet (instantaneous $\mathrm{II}_{\mathrm{sys}}$). The model is characterized by the transition ($\boldsymbol{TM}$) and emission ($\boldsymbol{EM}$) matrices, which contain the probabilities of a state transition and of emitting an II, given the current state.

10.3.2.2 Automatic Model Generation

Each state has as many conflict transitions as it has packets in the pipeline, plus one non-conflict transition. If the new address is in conflict with the one in the stage

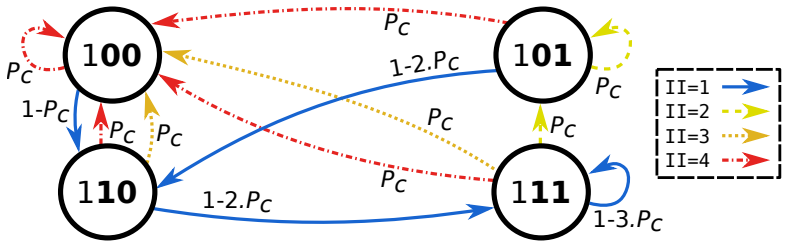


Figure 10.2: Hidden Markov model example for $DD = 3$ and $C \geq 3$.

$x \in [0..DD-1]$, then the emission will be $\text{II} = DD + 1 - x$ and $(state + 2^{DD-1}) >> \text{II}$ will be the new state. The probability of that conflict to occur is $P_c = \frac{1}{C}$. If there are multiple transitions between a pair of stages (with the same direction), the transition probability is the sum of all the individual probabilities. If there are no conflicts, then $\text{II} = 1$ and the new state is computed as before. Finally, notice that depending on C, there are forbidden states and transitions given that there might not be enough different addresses to fill the pipeline. A complete implementation can be found in the public repository. As an example, equation 10.1, shows $\boldsymbol{TM}$ and $\boldsymbol{EM}$ matrices for $DD = 3$ and $C \geq 3$.

(10.1)

$$\boldsymbol{TM} = \begin{array}{c} \begin{matrix} 00 & 01 & 10 & 11 \end{matrix} \\ \begin{pmatrix} P_c & P_c & 2.P_c & 2.P_c \\ 0 & P_c & 0 & P_c \\ 1-Pc & 1-2.P_c & 0 & 0 \\ 0 & 0 & 1-2.P_c & 1-3.P_c \end{pmatrix} \end{array} \begin{array}{c} \overleftarrow{from} / \downarrow to \\ \begin{matrix} 00 \\ 01 \\ 10 \\ 11 \end{matrix} \end{array} \qquad \boldsymbol{EM} = \begin{array}{c} \begin{matrix} 00 & 01 & 10 & 11 \end{matrix} \\ \begin{pmatrix} 1-Pc & 1-2.Pc & 1-2.Pc & 1-3.Pc \\ 0 & P_c & 0 & P_c \\ 0 & 0 & P_c & P_c \\ P_c & P_c & P_c & P_c \end{pmatrix} \end{array} \begin{array}{c} II \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array}$$

Using the HMM matrices, we can compute, for example, $\overline{\text{II}}_{\text{sys}} = [1 \ldots (DD+1)] \cdot \boldsymbol{EM} \cdot \boldsymbol{\pi}$, where the first row vector contains the value of the II emissions and $\boldsymbol{\pi}$ is the steady state distribution column vector (obtained from $\boldsymbol{TM}$). Notice that $\boldsymbol{EM} \cdot \boldsymbol{\pi}$ is the steady distribution of II_{sys}.

10.3.2.3 Approximation of the II_{sys} Distribution for General Block Size

The distribution of II_{sys} for any block size W and DD, is not trivial. However, we can obtain a good approximation modeling the system as a stateless one with only two possible packet acceptance sequences: S_0 (No conflict) and S_1 (the average conflict sequence). It is not hard to see that $P(S_0) = \sum_{i=2^{L-2}}^{2^{L-1}-1} \boldsymbol{\pi}_i$ and S_0 emits $\text{II}_0 = 1$. S_1 emits the mean conflict cycles, $\text{II}_1 = (\overline{\text{II}}_{\text{sys}} - P(S_0))/P(S_1)$, where $P(S_1) = 1 - P(S_0)$ and $\overline{\text{II}}_{\text{sys}}$ is given by the HMM. In this way, the number of S_1 in the W block (N_1) follows a

binomial distribution and $P(\mathrm{II}_{\mathrm{sys}} = cycles/W) = P(N_1 = \frac{cycles-W}{\mathrm{II}_1-1})$ (notice that $cycles = W - N_1 + N_1 \cdot \mathrm{II}_1 \in \mathbb{R}$).

10.3.3 A Simple $\overline{\mathrm{II}}_{\mathrm{sys}}$ Approximation

Although exact for the uniform distribution, the HMM requires somewhat compute-intensive operations. There are occasions when faster methods, though no exact, are more useful, and we also would like to have estimations for other distributions. We obtain a simple formula (exact for $DD = 1$) by assuming that the probability of having a full pipeline (no bubbles) is approximately one. As a result, we get: $\overline{\mathrm{II}}_{\mathrm{sys}} \leq F_2(DD, \overline{P_c}) = 1 + (DD^2 + DD) \cdot \overline{P_c}/2$. It is an upper bound because the full state has the highest conflict probability. The bound will be tighter as $\overline{P_c} \cdot DD$ grows smaller, given that the probability of this state gets closer to 1.

As $\overline{\mathrm{II}}_{\mathrm{sys}}$ increases, a linear approximation, $F_1(DD, \overline{P_c})$, fits very well the data (see fig. 10.4). Then, we may set a $\overline{\mathrm{II}}_{\mathrm{sys}}$ above which $F_1(\cdot)$ is used instead of $F_2(\cdot)$. Finally, using $F_2(\cdot)$ to estimate $F_1(\cdot)$ coefficients, we obtain:

$$
\begin{aligned}
DD_{lim} &= DD|_{F_2(\cdot)=\overline{\mathrm{II}}_{\mathrm{lim}}} = (\sqrt{(8 \cdot (\overline{\mathrm{II}}_{\mathrm{lim}} - 1)/\overline{P_c} + 1)} - 1)/2 \\
b &= \partial F_2/\partial_{DD}(DD_{lim}, \overline{P_c}) = (2 \cdot DD_{lim} + 1) \cdot \overline{P_c}/2 \\
\overline{\mathrm{II}}_{\mathrm{sys}} &\approx \begin{cases} DD < DD_{lim}, & 1 + (DD^2 + DD) \cdot \overline{P_c}/2 \\ DD \geq DD_{lim}, & \overline{\mathrm{II}}_{\mathrm{lim}} + b \cdot (DD - DD_{lim}) \end{cases}
\end{aligned}
\tag{10.2}
$$

An $\overline{\mathrm{II}}_{\mathrm{lim}} = 1.35$ results in good approximations (see fig. 10.3). For a more conservative approach (higher, pessimistic $\overline{\mathrm{II}}_{\mathrm{sys}}$ estimations), higher $\overline{\mathrm{II}}_{\mathrm{lim}}$ may be used.

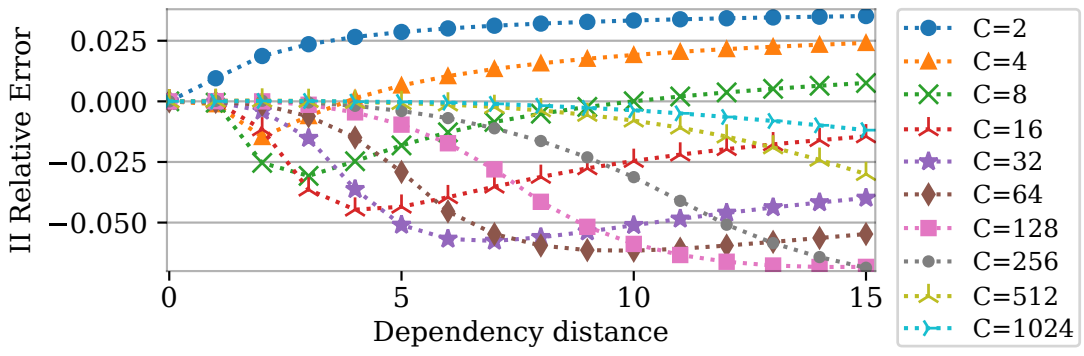


Figure 10.3: Relative error of Eq. 10.2 with $\overline{\mathrm{II}}_{\mathrm{lim}} = 1.35$ for random distributions.

10.4 Performance Analysis

10.4.1 II_{sys} Improvement for a given Processing Latency

Fig. 10.4 compares II_{base} with $\overline{II}_{sys}$ when CS is applied and the processing module is pipelined to achieve $II_p = 1$ without changing the frequency or DD (the processing latency remains constant). This is shown for both distributions and different cardinalities (C). The Zipf parameter s is set to 1.8 to evaluate a very skewed distribution ($P(1) = 0.6$ for a 8-symbol source), in contrast to the uniform. Notice that, even for low C and very skewed distributions, $\overline{II}_{sys}$ improves significantly. As C grows and the Zipf's s decreases, the throughput improvement is greater, given that conflict probability decreases.

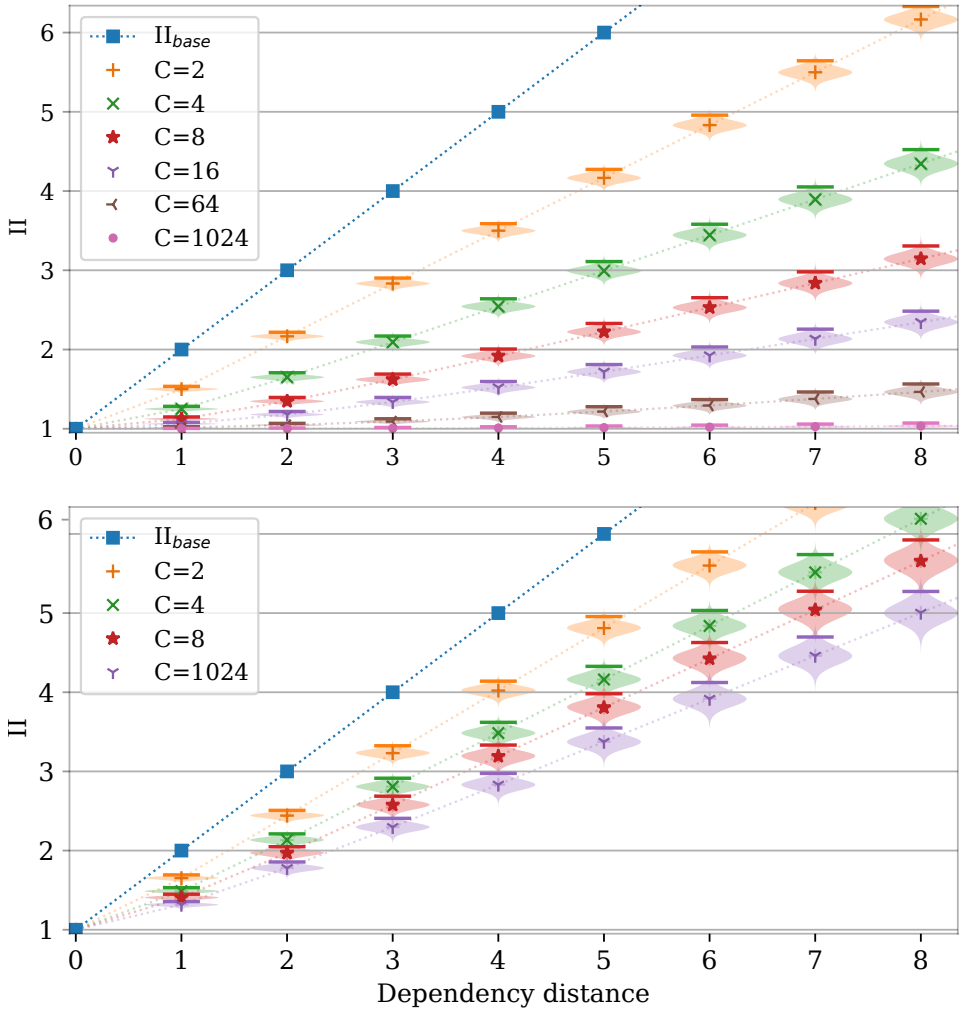


Figure 10.4: $\overline{II}_{sys}$ violin plots (99^{th} percentile delimited) for blocks of 1000 packets for uniformly (top) and Zipf with $s = 1.8$ (bottom) distributed addresses.

In general, e.g., due to the nature of the problem or the available buffer size, we need to understand the $\overline{\text{II}}_{\text{sys}}$ behavior for packet blocks of a given size. The worst-case performance for non-deterministic address sequences is II_{base}, which occurs for single-address bursts. Of course, as the block size increases, this sequence becomes rarer. To illustrate this, fig. 10.4 shows, using violin plots, the PDF of $\overline{\text{II}}_{\text{sys}}$ for blocks of 1000 operations, where the 99^{th} percentile (delimited) is noticeably better than II_{base}.

10.4.2 Increasing Throughout Optimizing Pipeline Depth

For feed-forward circuits (data only flow downstream), we can increase throughput using a deeper pipeline to reduce the clock period. However, there are many technology-

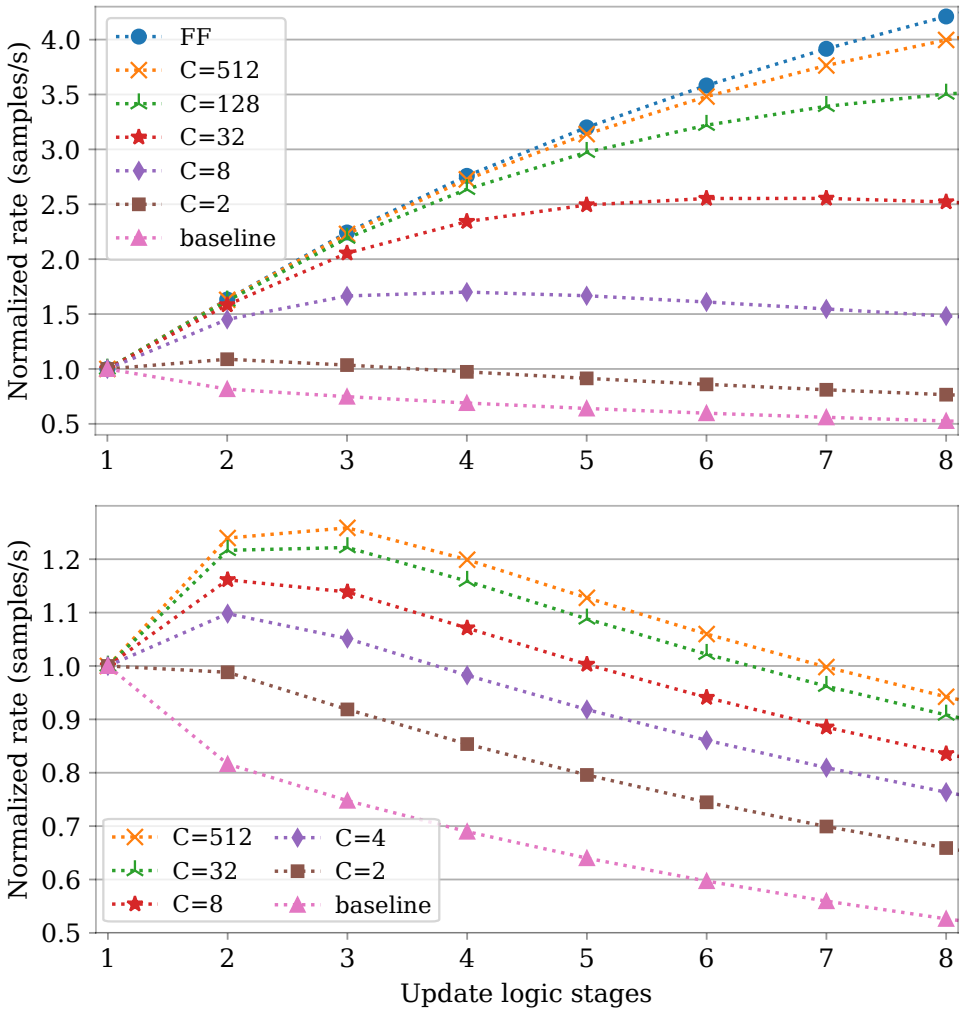


Figure 10.5: Throughput estimation as the number of pipeline stages of the processing module increases for uniformly (top) and Zipf (bottom) distributed addresses. For comparison, FF curve shows the feed-forward circuit behavior.

dependent inefficiencies associated with this process (work imbalance, increased clock skew, additional routing delays, etc) [176] [177, Chapter 2]. A simple model to estimate the resulting clock period for $S > 1$ stages would be: $P = T_{comb}/S + T_{pp}$, where T_{comb} is the period for the single-stage logic and T_{pp} the sum of pipelining penalizations, assumed approximately constant. In fig. 10.5, the FF curve shows the normalized throughput estimation for a feed-forward circuit with $T_{comb} = 8ns$ and $T_{pp} = 0.9ns$ as a function of logic stages. These constants fit the behavior of the example system in section 10.2.

A dependency creates a feedback loop in a module. If the dependency loop logic is deepened to increase frequency (using only static optimizations), the II increase will more than compensate the period reduction, worsening throughput. The baseline curve shows this effect, where, to ease the comparison with FF, the same T_{comb} and T_{pp} are used and the write and read latencies are set to 0 and 1, resp, then $DD = stages - 1$.

Conversely, when CS is applied, increasing the dependency loop pipelining has the potential to trade decreased worse-case throughput in favor of higher mean throughput because $\overline{\mathrm{II}}_{\mathrm{sys}}$ increases slower. The $C = x$ curves are confined between FF and baseline curves, drawing near to the former as the conflict probability decreases. Although increasing pipelining eventually decreases performance (conflict penalty increases faster than frequency), the curves show that most systems can be improved optimizing the pipeline depth.

10.4.3 Trade-off Between $\mathrm{II}_{\mathrm{sys}}$ and Area

We have only considered using a processing module with $\mathrm{II}_{\mathrm{p}} = 1$, but it might not be achievable, e.g., due to resource contention. For the simpler case where the processing logic consumes both packets and bubbles at a II_{p} rate, the system behaves as if $DD' = \lfloor DD/\mathrm{II}_{\mathrm{p}} \rfloor$ scaled by II_{p}. Then, the logic may only track DD' addresses and $\overline{\mathrm{II}}_{\mathrm{sys}} = \overline{\mathrm{II}}^{1}_{\mathrm{sys}}(DD') \cdot \mathrm{II}_{\mathrm{p}}$, where $\overline{\mathrm{II}}^{1}_{\mathrm{sys}}(\cdot)$ gives $\overline{\mathrm{II}}_{\mathrm{sys}}$ for a given DD and $\mathrm{II}_{\mathrm{p}} = 1$.

Moreover, controlling II_{p} enables different throughput-area trade-offs. Particularly for deeply pipelined modules and skewed distributions, increasing II_{p} can have a small impact on $\mathrm{II}_{\mathrm{sys}}$, while the area reduction may be significant as it increases the possibility of sharing resources and simplifies the control logic. Additionally, it may reduce pipelining penalties.

10.5 Application to 100 GbE Flow Metering

To further illustrate the capacity of the CS technique, we apply it to the HLS design of the TCP flow metering core for 100 GbE (see section 9.2.1). The core was compiled targeting different pipeline stages, and then implementing the whole metering system. The flow table was configured to hold 32Ki flows and use 8 ways (cache-like table), for which it was not previously able to support the maximum packet rate of a 100 GbE link. The implementations only apply the CS optimization, and thus the minimum DD is 1. To assess the behavior of the II, we employ the trace used in the memory requirement evaluation presented in section 5.4.2.1. Fig. 10.6 shows the results, where we compare the performance of this HLS design against the RTL solution presented in chapter 9.

As observed in the fig., for this design and application, CS can duplicate the average packets per second ("HLS+CS avg pps" curve), also significantly increasing the 99th percentile performance ("HLS+CS P99 pps" curve). This enables the HLS solution to surpass, most of the time, the optimized RTL design[1], even with shallow pipelines. As a reference, the fig. includes the pps of the HLS design as if CS were not applied ("HLS no CS pps" curve). This also indicates the worst-case performance of the HLS design with CS, which generally decreases with additional pipelining[2]. Thus, we trade off worst-case pps in favor of greater mean pss. Finally, confirming the results obtained in section 10.2, the CS stage does not contain the critical path in any of the HLS implementations.

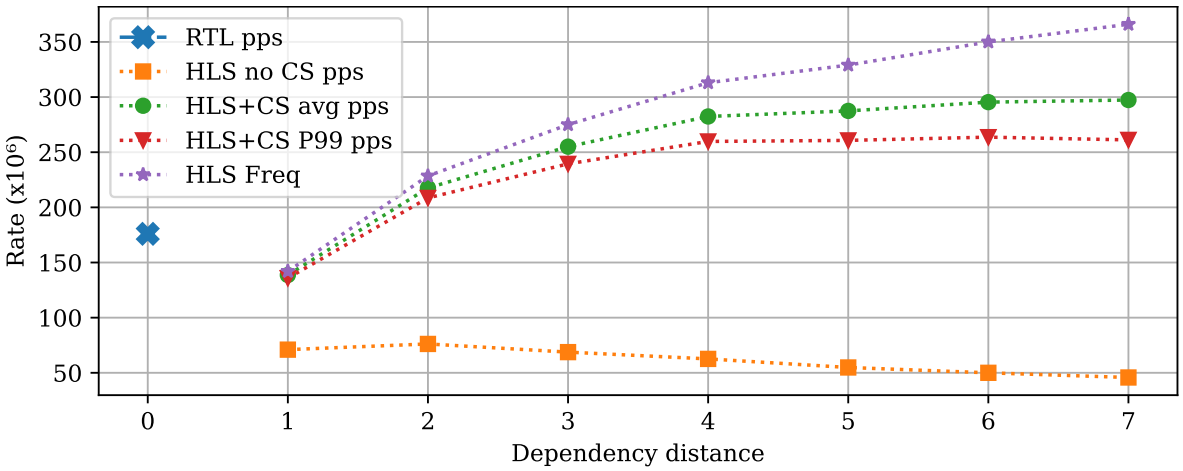


Figure 10.6: Performance comparison of RTL and HLS design with and without CS.

[1]CS and the RTL architecture optimization could be applied simultaneously.

[2]The performance of HLS without CS increases from $DD = 1$ to $DD = 2$ because there is a dead cycle in the circuit operation due to the memory latency and lack of data forwarding. Thus, moving to $DD = 2$ approx. duplicates the available time for the logic, while II increases only by 50%.

10.6 Discussion

For systems with 1-cycle-latency memories and single-stage processing, data-forwarding will generally be a more suitable as it assures an II = 1, although it may have some frequency penalties. However, as the dependency distance DD (function of the logic and memory latencies) increases, it gets less and less effective. It is precisely here where conditional stalling, CS, has a clear application. As illustrated by fig. 10.4, the larger DD, the greater the potential performance increase CS can offer. This improvement is also a function of the address distribution, but II will never be worse than the baseline. In addition, these techniques can be used together, using data-forwarding to mask the write latency (reducing DD) and CS to partially mask the remaining latencies[3]. For example, for Intel eSRAM (12 or 13 cycles latency), this would allow a much higher performance.

Additionally, CS enables throughput improvements by tuning the processing pipeline depth. However, notice that CS can be a double edge sword (see fig. 10.5), since this technique magnifies the diminishing returns of pipelining. Thus, we may end up with a higher area and a slower system. To actually increase throughput, knowledge of the application and of how frequency varies with the number of stages is necessary. Nowadays, obtaining the latter is easier than it was in the past, given that an HLS compiler can automatically iterate over increasingly deeper pipelines and gather timing data (pre- or post-RTL-synthesis estimations, or post-RTL-implementation).

Then, when the address distribution, the mean collision probability ($\overline{P_c}$), or a representative input vector is available, mathematical and/or simulation models can be employed to optimize the logic depth and compute the required buffers for a given confidence level. Of course, the distribution might not be stable or very little information about it might be available. In these cases, taking a pessimistic approach, assuming a very skewed distribution might be a viable option. A naïve attempt to limit pipelining would be that if $DD >= C$, the logic depth should not be increased. However, this is not very useful as, even with zero pipelining penalties, the throughput increase after this point is almost null. As future work, we would like to study the implementation of an adaptive system with multiple processing units of varying depth (and clock frequency), choosing at runtime the higher throughput alternative according to the collected conflict statistics.

Finally, CS enhances portability and functional robustness. A design may ignore a dependency because it is not true given known input data properties, but if these

[3]For CS to effective, slow memories have to queue enough requests without significantly increasing the latency (which also has to be bounded to use CS).

properties change or the design is reused in another system, it might fail. CS ensures that designs will always be functionally correct.

10.7 Conclusion

In this chapter, we have studied the conditional stalling technique, a throughput optimization for logic with data dependencies. This dynamic scheduling method consists of running a pipeline at full rate when no conflicts are detected, using scheduling logic to insert bubbles to ensure that no dependency is violated. We implemented two HLS designs and analyzed the footprint and frequency penalties associated with this optimization. As a result, we observed that the alternative with decoupled scheduling logic architecture will not generally present frequency penalties and has a low area footprint.

We also examined the performance when using this method as a function of input data and architecture characteristics, with the goal of understanding when to use it and how to optimize throughput. We showed that, even in adverse cases, it can significantly enhance performance, particularly when unavoidable latencies are present in the dependency path. However, we demonstrated that to correctly optimize throughput, we must take into account both the address statistics and the evolution of frequency as the pipeline is deepened. Previous works have not considered this, and as a result, applying the technique led to poor improvements or even to a slower and higher area system. Finally, this optimization could be integrated within HLS compilers, which can use the models here provided to make design decisions, resulting in better quality of results and increased designers' productivity.

Example systems, as well as the developed simulation and mathematical models, are available through a public repository.

CHAPTER 11

Automatic Partitioning and Resource Balancing

This chapter presents and evaluates our novel tool for system partitioning and resource balancing. It addresses a set of problems that emerge when scaling up FPGA accelerators in multi-SLR and multi-FPGA platforms. The utilization of this optimization increased the frequency and computational units per area of the tested CNN dataflow accelerators, demonstrating up to a 103% performance improvement over an already optimized monolithic implementation. Targeting multi-node platforms allowed to increase compute density further, and when combined with model-parallelism, enabled lower latency and more power-efficient implementations.

This chapter is based on the work published in [83]: Alonso, T., Petrica, L., Ruiz, M., Petri-Koenig J., Umuroglu Y., Stamelos I., Koromilas E., Blott, M. & Vissers K. (2021). Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning. ACM Trans. Reconfigurable Technol. Syst. 15, 2, Article 15 (June 2022), 34 pages. https://doi.org/10.1145/3470567. © 2021 Association for Computing Machinery . This project was done in collaboration with the Xilinx Research Labs in Ireland, working as a research intern and later as a visiting scholar. Also, the implementations' evaluation was done with the help of InAccel (`https://inaccel.com/`). My main contributions in this project were formulating the ILP optimization, developing its implementation, building the accelerator prototypes here presented (including debugging and performing the initial performance assessment), collaborating on the effort to provide support in the FINN compiler for the tested CNN models (including additional optimizations) and co-designing the tool evaluation testing methodology).

11.1 Introduction

As our data processing needs grow, larger systems will be deployed using FPGAs. However, as described in chapter 6 complex optimizations need to take place to achieve efficient implementations. This is particularly true for designs that do not fit in a single monolithic FPGA region (SLR). For these, it is more important to consider logic placement to avoid large frequency penalties due to very long paths. Even with today's large FPGAs, which have in the order of millions of logic cells, there are systems that do not fit within a single chip and also multi-chip implementations can allow higher density deployments, and thus, optimizing costs. What is more, given that a global point of view is required, FPGA implementations often leave unused large proportions of FPGA resources, missing the opportunity of better compute density (performance/area).

We propose a novel tool to address these optimizations in a unified fashion. This tool, based on integer linear programming, allocates dataflow (chain of modules running in parallel) hardware implementations to FPGA tiles considering resource and connection constraints. Additionally, the tool can accept a set of versions for each module (with equal functionality, but different resource utilization profile, e.g., using LUTs instead of DSPs for an arithmetic operation) and it will choose the most appropriate one given the optimization goals and constraints (different module instantiations may use a different version). The tool supports multi-FPGA deployments, instantiating, when necessary, communication infrastructure between chips for intermediate results transmission without host CPU intervention. The tool implementation and code exemplifying its application to FINN CNNs for frequency optimization are available through public repositories [120, 178].

11.2 Partitioning and Resource Balancing Tool

In this section, we explain what the tool does, how it works, and how it is implemented.

11.2.1 Implementation Flow

Given that to optimize some wire connections, like reset and memory interfaces, it is necessary to know, at least approx., the final placement of the module, logic placement is done in two steps. Resource balancing also needs to take into account placement before version instantiation, and thus, also requires a two-step process.

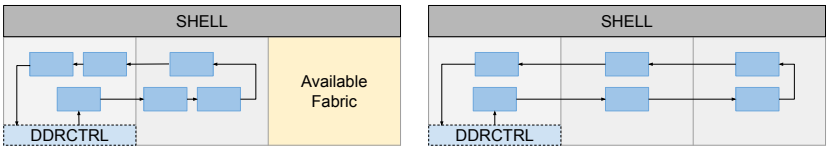
The first step is the floorplaning and version instantiation one, which consists of a coarse placement, where module versions are assigned to tiles. In Xilinx Vivado, tiles are pBlocks, which are defined by the user and can be from a whole FPGA to an SLR or a fraction of it, like a clock region. After this, logic connections are performed considering the floorplan, then the system goes through logic synthesis, and finally, the second placement step, P&R. In the floorplaning step, soft pBlocks are used, which allow module logic to spill over the pBlock boundaries, giving P&R more freedom, and thus, in principle, implementations should be at least as optimal as those of using hard pBlocks (no spill over allowed).

The floorplaning step is done by our tool using resource estimates for each of the modules. There are several ways for the user to get these estimations. In general, for RTL designs, post OOC synthesis estimates may be used, while HLS IPs may also use the ones provided by the HLS compiler. If the module resources have been previously modeled, estimates might be obtained directly from its parameters, like FINN can do [179]. The FINN model and HLS estimates are not, however, as precise as those after OOC synthesis, which are very similar to the final results after P&R, but they require more time. The most suitable estimation method depends on the context (e.g. prototype vs production phases).

11.2.2 Tool Capabilities

Given the accelerators and target platform description, provided by the user, the tool will search viable allocations. The accelerator description specifies the modules that compose it and their compute and communication (bandwidth/pins) resource requirements. The platforms are described as a set of tiles, each of which provide compute resources and are connected though communication resources. As in the case of the platform tiles, the granularity of the accelerator modules is defined by the user. For CNNs accelerators, for example, we may choose CNN layers. To control congestion, the user provides limits to the utilization rate for each of compute resource and/or to the average utilization rate of a set of them, which are applied to each tile. Therefore, a viable allocation consist in an assignment of each module to a tile, such that the resource constraints are respected.

Anchoring Modules For several reasons, the user may need to restrict the location of a module to a subset of tiles, for example, due to the location of certain interfaces. Additionally, the user may want to force two modules to be in the same tile, for example, those that access off-chip may be forced to be placed together to avoid the implementation



((a)) SLR minimization ((b)) Congestion minimization

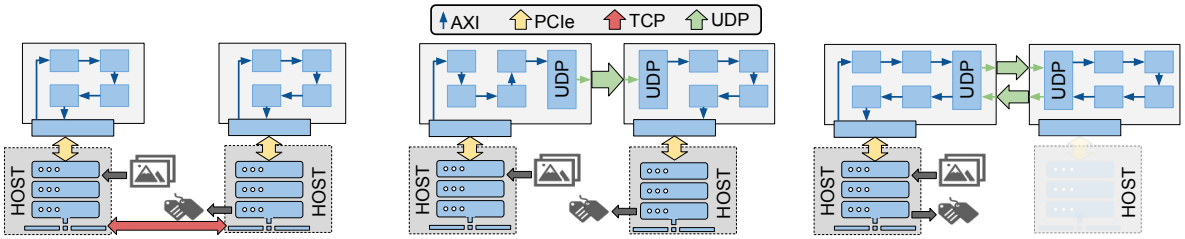
Figure 11.1: Alternative system partitioning strategies. Source [83]

of multiple interfaces. Both of these anchoring types are supported by the tool, using the name *absolute anchor* for the former and *relative anchor* for the latter.

Allocation Optimization For any given instance, there might be multiple viable allocations. From this set, the tool will choose those with the minimum communication cost, which is a function of the hardware platform. For example, module connections across FPGAs are less desirable compared to those within a single chip, as the former require more logic and energy. In general, we may use these communication costs to implement higher level partitioning goals. For example, an SLR minimization strategy would reduce renting costs in a datacenter multi-tenancy scenario, which can be obtained penalizing SLR crossings, resulting in an implementation as fig.11.1(a) illustrates. Alternatively, we may want to minimize congestion for the given platform, reducing resource utilization limits until the desired congestion equalization is achieved or there are no viable solutions (going from the system in fig.11.1(a) to that of fig.11.1(b)). In this latter case, SLR crossing penalization are still useful as a way to minimize them, given that these crossings limit place and route within a tile. Thus, a large amount of SLR crossings, albeit below the number of super-long lines (SLL, paths connecting SLRs), may lead to frequency penalties.

Multi-Dataflow Systems As commented indicated in section 6.3.2.3, there are situations in which we may want to implement multiple accelerators, which is supported by the tool, enabling data-parallelism, modules communicating through external memory, or, in general, to obtain a globally optimized implementation of multiple systems in a single platform.

Choosing Model-Parallelism Alternatives In chapter 6, we introduced model-parallelism, indicating that it basically consists of multiple nodes (chips) implementing a fraction of a single large DFA accelerator, but there are several partitioning alternatives from the accelerator interface point of view. Each of the three approaches, illustrated



((a)) SWMP ((b)) HWMP ((c)) TMP

Figure 11.2: Model parallelism alternatives for dual-FPGA implementations. Source [83]

by fig. 11.2, has advantages and disadvantages. Software model-parallelism (SWMP) requires the hosts to share intermediate results, but it does not dedicate logic to implement FPGA-to-FPGA communication. In hardware model-parallelism (HWMP), the intermediate result transmission is done without host intervention, favoring lower latency. Finally, transparent model-parallelism (TMP) interfaces to the host as if the implementation was single-node and only needs a single host after the initial set-up, but this requires more connection resources. A priori, neither of these alternatives will consistently result in the most efficient implementation, for example, in terms of congestion equalization. The set of possible allocations for each of them will generally differ (e.g. TMP will have the first and last nodes in the same FPGA while HWMP may not), and thus the efficiency and the viability of the implementation depends on the specific resource profile of the accelerator. Setting the platform connection options and the anchors appropriately determines the set of MP alternatives available. For example, TMP can be forced enabling direct FPGA communication and anchoring the last module of the accelerator to the first one.

Generic Nature Finally, notice that although developed for FPGAs, the tool is not technology-specific. In principle, the tool can be used to partition for heterogeneous platforms. In such cases, different versions of a module may correspond to implementations in different technologies, such as FPGA fabric, μP, GPU or AI engines [180], each of which will have their own type of resources.

11.2.3 ILP Formulation

In this section, we describe the ILP formulation of the optimization problem.

11.2.3.1 Input Definition

Dataflow Graph The task graph $T = (T^n, T^e)$ carries the information regarding the computations to be implemented, where T^n represents the set of DF nodes, e.g. CNN layers, and T^e represents the set of connections between nodes.

Each node $p \in T^n$ is defined for the purposes of partitioning by its potential implementations and their respective resource utilization profiles. We denote the set of node resource types as U^n and $d = |U^n|$ is the total number of node resource types. For FPGA implementations, typical resource types are LUTs, FFs, DSPs, BRAMs and URAMs so $d = 5$. Each possible node implementation $v \in p$ is therefore a vector in $\mathbb{Q}^d$ or $\mathbb{N}_0^d$ specifying the utilization of each resource type for that implementation. For example, we can have a version using DSPs for a certain computation and an alternative version using LUTs. We denote t^n_{p,v,u^n} the requirement of resource type $u^n \in U^n$ of version $v \in p$ of task $p \in T^n$.

Each task edge is an ordered pair $k = \{p, q\} \in T^e$ which establishes a connection from task node p to task node q, i.e. the output of task node p is the input of task node q. We denote the set of connection resource types as U^e. For multi-FPGA accelerators, connections between task nodes can be established through dedicated wires if task nodes reside on the same FPGA or through shared chip-to-chip connectivity if nodes reside on different FPGAs. The number of wires and off-chip throughput define the set of resource types in this example. We denote t^e_{k,u^e} the edge requirement of resource type $u^e \in U^e$ associated with task edge $k \in T^e$.

Compute Platform The graph $C = (C^n, C^e)$ describes the target platform. Each node in C^n represents a compute unit we can assign tasks to, for example, for Multi SLR FPGA devices it can be a SLR, or otherwise a pBlock or even a whole FPGA device. Each compute node $i \in C_n$ provides a certain amount c^n_{i,u^n} of each resource type $u^n \in U^n$. Each edge in C^e establishes a connection between compute nodes i and j and provides up to c^e_{i,j,u^e} of each connection resources type $u^e \in U^e$. For example, a connection between SLRs residing on the same FPGA may provide a large number of wires (SLL) but no off-chip throughput, whereas a connection between SLRs on different FPGAs will provide no wires and some off-chip throughput.

11.2.3.2 Constraining the Partition

To be able to set utilization ratios for the compute resources, a limit $l_{u^n} \in [0, 1]$ is provided for each resource type $u^n \in U^n$. Average utilization constraints of the form:

$Avg(u_0^n, u_1^n, u_2^n, ..) < l$ are defined by a list A^u of tuples $\{l, M\}$, where $l \in [0,1]$ is the average utilization limit and $M \subseteq U^n$ is the subset of resource types included in the average.

Anchoring constraints are specified through two lists of tuples: A^a and A^r corresponding to absolute and relative anchors respectively. Each tuple $\{p, N\} \in A^a$ restrict a task $p \in T^n$ to be placed in a subset of compute nodes $N \subseteq C^n$. Each tuple $\{p, q\} \in A^r$ forces task $p \in T^n$ to be in the same compute node as task $q \in T^n$.

11.2.3.3 Optimizing the Partition

The partitioning algorithm described herein maps the task graph to the compute graph, within the constraints specified. In addition, we want to optimize placement and/or connections of the task nodes. For this, we use an additional matrix, the connection cost matrix C^c. The element $c^c_{i,j} \in C^c$ is the cost associated to the connection between compute node i and j.

The partitioning is expressed as an Integer Linear Program (ILP). For the formulation of the problem, we use two sets of binary variables, the edge map M^e and the node map M^n . The edge map variables $m^e_{i,j,k} \in M^e$ state whether the task dependency k connects a task node mapped to compute node i with a task node mapped to compute node j. The other set of auxiliary variables are the node map variables $m^n_{i,p,v} \in M^n$, that state whether the version v of task node p is assigned to the compute node i.

Finally, the ILP formulation is:

$$\text{minimize} \sum_{i \in C^n} \sum_{j \in C^n} \sum_{k \in T^e} m^e_{i,j,k} * c^c_{i,j} \tag{11.1}$$

Subject to:

$$\forall p \in T^n : \sum_{i \in C^n} \sum_{v \in p} m^n_{i,p,v} = 1 \tag{11.2}$$

$$\forall k = \{p, q\} \in T^e, \forall i \in C^n : \sum_{j \in C^n} m^e_{i,j,k} = \sum_{v \in p} m^n_{i,p,v} \tag{11.3}$$

$$\forall k = \{p, q\} \in T^e, \forall j \in C^n : \sum_{i \in C^n} m^e_{i,j,k} = \sum_{v \in q} m^n_{j,q,v} \tag{11.4}$$

$$\forall i \in C^n, \forall u^n \in U^n : \sum_{p \in T^n} \sum_{v \in p} m^n_{i,p,v} * t^n_{p,v,u^n} \leq c^n_{i,u^n} * l_{u^n} \tag{11.5}$$

$$\forall \{i, j\} \in C^e, \forall u^e \in U^e : \sum_{k \in T^e} m^e_{i,j,k} * t^e_{k,u^e} \leq c^e_{i,j,u^e} \tag{11.6}$$

Here, eq. 11.2 forces each task node to be placed once and only once. Constrains 11.3 and 11.4 create the link between M^e and M^n, that is, these variables are not independent. For each task edge $k = \{p, q\} \in T^e$, there are compute nodes i and j such that $m^e_{i,j,k} = 1$. Then, by definition there must be $v_0 \in p$ and $v_1 \in q$ such that $m^n_{i,p,v_0} = 1$ and $m^n_{j,q,v_1} = 1$, respectively. Resource limitations are set by eq. 11.5 and 11.6 for node resources and connection resources, respectively.

Applying anchors The formulation for these constraints is:

$$\forall \{p, N\} \in A^a : \sum_{i \in N} \sum_{v \in p} m^n_{i,p,v} = 1 \tag{11.7}$$

$$\forall \{p, q\} \in A^r, \forall i \in C^n : \sum_{v \in p} m^n_{i,p,v} = \sum_{v \in q} m^n_{i,q,v} \tag{11.8}$$

Applying average utilization constraints Defining C^n_M as the subset of nodes in C^n for which $u^n > 0$ for $u^n \in M$, the formulation for these constraints is:

$$\forall \{l, M\} \in A^u, \forall i \in C^n_M : \frac{1}{|M|} \sum_{u^n \in M} \frac{1}{c^n_{i,u^n}} \sum_{p \in T^n} \sum_{v \in p} m^n_{i,p,v} * t^n_{p,v,u^n} \leq l \tag{11.9}$$

11.2.4 Partitioner Implementation

The developed ILP formulation is generic, i.e. can partition any DFA for any platform as long as these are specified appropriately. For this work, the partitioner was implemented in Python using the ILP solver provided by the mip [181] module. The resulting partitioner is relatively fast. For a graph consisting of 100 nodes targeting a platform of 10 SLRs with resource utilization approaching the target resource limits, the runtime is only a few seconds. The partitioner can be easily integrated into any framework which provides per-layer resource estimation and infrastructure to pass placement constraints into the design at build time.

11.2.5 FINN Integration

Figure 11.3 illustrates how our partitioner integrates with the FINN compiler. A custom FINN analysis pass translates between the FINN intermediate representation (IR) and the partitioner task graph. The compute graph is generated by custom code which describes platforms, which were described at the SLR level. Anchors are optionally provided by the user as arguments to the analysis pass. Based on the partitioner solution, the location and configuration of each node is stored in the IR. Later in the DFA build

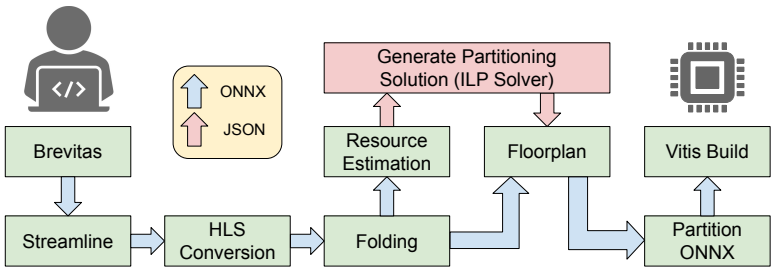


Figure 11.3: Integration of the ILP partitioner into FINN. Source [83]

process, the location attribute instructs FINN to separate the model into partitions, i.e. contiguous layers allocated to the same region. Subsequently, each partition is converted into a Vitis kernel and linked by the FINN backend to produce a bitstream.

11.3 Tool Evaluation

11.3.1 Experimental and Implementation Setup

Communication Infrastructure For the FPGA-to-FPGA connectivity, we use the Vitis network layer [182] (VNx), which implements the UDP/IP protocol over a 100GbE link in the FPGA fabric. This kernel requires 35K LUTs, 86K FF and 183 BRAMs, achieves 100% of the UDP bandwidth (function of packet size), dissipates 6 Watts at full throughput and has a client to user latency of about 0.5 μs for point-to-point connections. These specifications are sufficient for our application. Although other communication systems might be more efficient in terms of bandwidth, latency, and resources, VNx provides great flexibility as Ethernet connections are generally available in acceleration cards, the connections between FPGA can be easily reconfigured without rewiring (e.g. connecting FPGA through a packet switch or router) and there are no co-location restrictions between chips.

ETH Zurich XACC The implemented systems were deployed and tested in the Xilinx Adaptive Compute Clusters (XACC [183]) in the ETH Zurich. This evaluation cluster consists of four servers, two of them with 2xU250 Alveo cards and the other two with a 1xU250 + 2xU280 configuration. To provide connectivity, one of the two 100 GbE interfaces of the Alveo cards and the server's NICs are connected to a switch, while the other acceleration card interfaces are used for direct connections among them. The XACC was chosen given that it provides the necessary acceleration and connectivity

infrastructure expected to be found in datacenter environments, plus it is open to the research community, and thus, facilitates results replication.

Software for runtime execution For the FPGA accelerator instantiation and configuration XRT [184] and PYNQ for Alveo [185] are used, while the multi-node orchestration is done with Dask [186] and InAccel Coral [187]. To support multi-FPGA accelerators, InAccel developed a custom runtime, enabling initialization-time configuration of the VNx and runtime synchronization between accelerator segments where applicable (HwMP, SwMP). This runtime abstracted the multi-node aspects, enabling the use of the existing InAccel MLPerf test harnesses with no changes to application code.

Platform and Accelerator description Platform tiles correspond to FPGA SLRs, and their available resources were obtained from Vivado-generated reports instead from the cards' documentation [188], as we found the former to be more accurate. For multi-node system evaluation, we used dual-FPGA platforms with 100 GbE connection between them, either using VNx or the host NIC. We applied absolute constraints to ensure DMAs were placed in SLRs which have access to off-chip memory interfaces. Although U280 cards have both HBM and DDR, the HBM was used by default. As for resource estimates for DFA layers, OOC synthesis estimations are employed. In most cases, pipeline registers or FIFOs were inserted to cross SLR boundaries.

Build configuration Regarding the implementation flow, Vitis 2020.1 was used to perform partitioned kernels connections, SLR allocation and implementation, and we utilized U250_XDMA_201830_2 and U280_XDMA_201920_3 platforms for U250 and U280, respectively. A high effort process was used to implement the accelerators, selecting the highest Vitis optimization level (-O3). In addition, PHYS_OPT_DESIGN phase and post-route TNS cleanup were activated, plus ExploreWithRemap directive was used for OPT_DESIGN phase, while Explore directive for PLACE_DESIGN, ROUTE_DESIGN, and PHYS_OPT_DESIGN. Finally, to eliminate the typically high variability in the resulting F_{max} when changing the Vitis target frequency, each build was run multiple times with target frequencies ranging from 180 to 240 MHz in increments of 10 MHz, and we kept only the build with the highest F_{max}. Although a more complex build process could be used to better evaluate the impact of our optimizations on maximum frequency (for example, using implementation strategy sweeping), we regard it as a good balance between data quality and the number of systems we can build for evaluation (the process it is very time-consuming).

11.3.2 Effect of Partitioning on Operating Frequency

Table 11.1: $F_{\max}$ of baseline and partitioned single-node DNN accelerators. Source [83]

Accelerator	Card	#DFAs	$F_{\max}$ Monolithic (SLR0/SLR1)[1]	$F_{\max}$ Partitioned	Speedup
RN-50	U250	1	109 / 154	193	1.74 / 1.25
RN-50 (Fold 2x)	U280	1	123 / 208	217	1.76 / 1.04
MN	U250	2	110 / 142	210	1.90 / 1.48
MN	U280	1	152 / 215	241	1.58 / 1.12

[1] Control signals and DDR interface connected to the specified SLR ports.

To evaluate the impact of partitioning (as performed by our tool) on clock frequency, we implemented single-FPGA MN and RN-50 accelerators targeting U250 and U280. Folding factors and number of DFAs were configured to achieve large resource utilization rates. For each accelerator, three connection configurations were used. Two of them employed monolithic DFA kernels, connecting its control signals and DDR interfaces either to SLR0 ports (Vitis default configuration) or to the more central (SLR1). The third configuration used kernels partitioned by our tool (aiming at equalized congestion) with its ports connected to the corresponding SLR where it was placed. A single version of each DFA module was provided so that the total resource utilization rates are approx. the same [1]. As previously commented, frequency sweeps were performed for each configuration from 180 to 220 MHz (5 builds each).

Table 11.1 presents the maximum frequencies achieved for each configuration, along with the relative speedup. These results show that when the port location for the monolithic kernel is optimized (using SLR1 ports instead of SLR0 ones), frequency increases significantly, particularly for the U280 implementations. The FPGA in U250 has 4 SLRs while U280 has 3 of them, then the reset logic on SLR4 has to still traverse 2 SLR crossings and/or conditions logic placement more strongly, which explains the larger frequency penalty U250 implementations exhibit. However, the partitioning created be our tool further improved the operating frequency resulting, on average, in an 75% and 22% increase, compared to the Vitis default and the port location optimized configurations, resp. Again, for the same reasons explained above, the U250 builds show a more pronounced improvement.

It is also worth noting that we observed an increased routing success rate when the design was partitioned. For example, for RN-50 (baseline), 3 out of 5 builds (with

[1]Optimizations, like constant propagation and logic replication, may create some discrepancies

different target frequencies) failed using the default configuration, 2 of 5 failed in the optimized case, but all builds succeeded for the partitioned DFA. We also observe a reduction in total build runtime by approximately 10% when using partitioning.

11.3.3 Effect on Compute Density

Table 11.2: Comparison of compute density achieved by the ILP tool for single- and multi-node implementations, and with and without multi-version resource balancing against the optimized monolithic implementation for MN targeting U280 Alveo cards

	#DFA	F_{max}	#Cards	FPS/Card	Speedup
Monolithic SLR 1	1	215	1	1920	1.00
Simple-node	1	241	1	2152	1.12
Dual-node	3	227/230	2	3040	1.58
Dual-version	2	218	1	3893	2.03

For this application, we measure compute density as FPS/area (computed from accelerator specifications), or given that we limited the analysis to U280 cards, we may use FPS/Card. To evaluate the capacity of our tool to improve this efficiency measure, we compare against the results achieved by the optimized monolithic implementation for the MN CNN targeting U280 Alveo, presented in section 11.3.2. As previously seen, in this case, partitioning the single-version MN DFA the shows a modest 12% improvement given that U280 has only 3 SLRs. In this scenario, where partitioning has a lower impact, we tested the impact of the partitions on compute density when the number of nodes increased, and when there were more versions of each module. Table 11.2 presents the results of this evaluation.

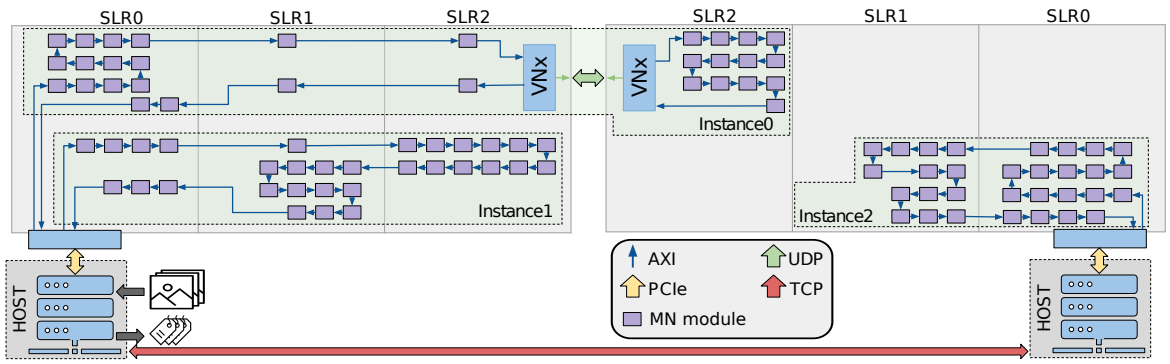


Figure 11.4: Partition of 3xMN in 2xU280 (DP+TMP, default resource profile). Source [83]

11.3.3.1 Dual-Node implementation

The partitioning result using single-version MN modules is illustrated in fig. 11.4, where it can be appreciated that the tool was able to squeeze an additional DFA, using the spare SLR from the FPGA on the right, while balancing congestion in the FPGA on the left using a TMP configuration. This is achieved with only a 5.8% frequency penalty, resulting in a 41% compute density improvement overall (58% if compared to the baseline). Notice the good frequency balance achieved as one FPGA implementation reaches 227 MHz while the other 230 MHz. Another sign of the quality of the partition is the reduced number of SLR and FPGA crossings that it presents.

Although fig. 11.4 shows how a single user would exploit it, this implementation provides a good example of the increased efficiency that could be achieved using the tool in an FPGA multi-tenancy scenario, given that the additional DFA uses a TMP configuration, and thus, from the host point of view, it is implemented in the FPGA on the left. The host on the left would be renting 4 SLRs to implement 2 MN, while the host on the left just 2 SLR for a single MN, and thus, each would minimize its costs, and the overall energy consumption is lowered as 2 cards are used instead of 3. In this example, the datacenter operator is providing with 66% of resources almost the same services (there is some frequency penalty).

11.3.3.2 Improvement Through Resource Balancing

To evaluate the most advanced resource balancing capabilities of our tool, an additional version of each module using DSP and URAMs was provided to the ILP tool. As a result, it was able to squeeze another DFA in a single U280, with only a 9.5%

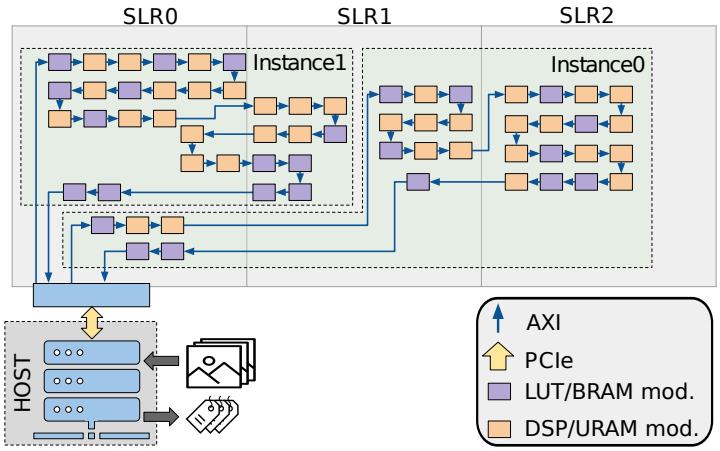


Figure 11.5: Partition of dual-version 2xMN in 1xU280. Source [83]

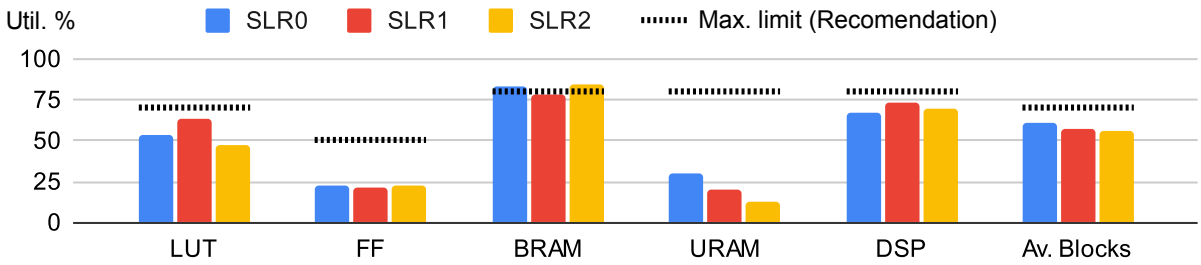


Figure 11.6: Resources per SLR for partitioning of dual-version 2xMN in 1xU280

frequency penalty, resulting in an 81% compute density improvement overall (103% compared to the baseline). Fig. 11.5 show the partition solution, where the two types of modules can be distinguished, and fig. 11.6 shows the resource rates per SLR of this partition. To achieve this implementation, the tool chose a great mix of versions, and we note that the tool does not always choose the same version for different instances of a given module. Although fig. 11.6 shows high utilization rates, the implementation presents a low frequency penalty, supporting the quality of the partitioning achieved by the tool. complexity of balancing all these simultaneously. Despite the complexity of this optimization task, where multiple resources need to be balanced, connection pins between SLRs are limited, and it is desired to minimized SLR crossings, the tool requires less than a second to solve this instance.

Notice that, in this case, BRAM would not allow increasing the number of DFAs/Card (at least, not without great frequency penalties). Additional versions (for example, hybrid ones) and lower grained DFA modules, would also open the opportunity for frequency improvement. Finally, to enhance performance further, the MN DFA folding could be reduced, but FINN did not support additional parallelism for the MN (see section 6.3.2.3).

11.3.4 Effect of Scale-Out Alternatives on Latency and Power

Four dual-node systems were implemented, one using data-parallelism (DP) and three using model-parallelism (MP), one per each type (see 11.2.2). Given that FINN cannot further increase the parallelism of the MN DFA, we use RN-50 targeting a dual U280 platform. The MP systems use the two cards to implement a single RN-50 DFA with the baseline folding (II = 56 kcycles), while the DP system uses two folded down DFAs (II = 112 kcycles), one per Alveo, given that the baseline-fold DFA does not fit in a single U280. We measure the DFA inference latency, but also we use two test scenarios of the MLPerf benchmark [189]: Single-stream (SS) test, which measures the 90th-percentile latency experimented by the application, and Multi-stream (MS), which

Table 11.3: Implemented ResNet-50 inference on two Alveo U280s at XACC. Source [83]

Parallelism	DFAs (#)	F_{max}	Peak FPS	DFA Latency (ms)	SS[1] Latency (ms)	MS[2] 50ms, P_{99} (Streams)	Cards Power [3] (W)
DP (Fold 2x)	2	215	3513	2.72	5.71	58	134
SW-MP	1	215/230	3559	2.01	6.25	82	116
TMP	1	215/215 [4]	3374	1.52	4.08	109	121
HW-MP	1	220/235	3618	1.55	3.97	120	125

[1] Single-Stream scenario measures the 90th-percentile application latency.
[2] Multi-Stream scenario measures the number of streams that the system can serve in 50ms (P_{99}).
[3] Average power dissipated by Alveo cards (CPU and NIC not included) for batch size is set to 400.
[4] Test run for old version with lower frequency (205/220 MHz).

focuses on latency-constrained throughput, measuring the 99th-percentile of the number of streams (parallel queries) that the system can serve within 50ms.

For the DP instance, the default Inaccel Coral runtime is utilized to distribute the data to the FPGAs and gather results, while the adapted runtime handles the data movements for HwMP and SwMP. For the MP solutions, ≈ 40 Gbps (at peak performance) are required for the communication between the FPGAs, while the DP requires ≈ 0.84 Gbps. For robust operation, in TMP and HwMP dual-FPGA systems, the frequency of both cards is set to the minimum of the two FPGAs F_{max}, given that UDP does not provide flow control. Table 11.3 presents the results of this evaluation. As noticed in section 11.3.3.1, see that the achieved frequency of each node of a given implementation is similar, which further supports the capacity of the partitioner to equalize congestion.

Latency Analysis Although MP provides a slight FPS improvement due to the higher achieved frequency (probably caused by the reduced OCM congestion), the greater impact is on the DFA latency, which is almost halved by the TMP and HW-MP alternatives. These two implementations, which use the direct FPGA-to-FPGA transmission of the intermediate results (TMP in both directions), also achieve the best SS and MS measures. TMP provides a 28.5% lower SS latency and 88% more streams in the MS test compared to DP, while HW-MP improves them further, accomplishing 30.4% lower SS latency and 106% more streams (vs. DP). The other MP instance, SW-MP, has a noticeably higher latency, which is explained by the fact that the host moves the intermediate results (large amount of data).

Power Analysis Ideally, to evaluate the energy efficiency of the different implementations, given that they differ on their infrastructure usage, one would measure the power consumed by the whole system. However, XACC does not provide meant to do this, so only the Alveo card measures were obtained (see table 11.3). Then, TMP measures more accurately the power required for the inference application (no host NIC data movement), followed by HW-MP (peak NIC BW≈ 35 kbps), then DP-MP (peak NIC BW≈ 0.84 Gbps) and, finally, (peak NIC BW≈ 40 Gbps). Each VNx IP consumes 6 Watts at full throughput, which is half (or less) of what commercial 100 GbE NICs consumed [190, 191]. Note that when using direct FPGA-to-FPGA transmission the data is already in the FPGA, while using the host involves first moving the data to the NIC (either from the host memory, or worst, from the FPGA, which requires one additional PCIe movement). Thus, although SW-MP measure is the lowest, the whole inference task consumption is probably in the 120-135W range. Interestingly, TMP and HW-MP cards, which have no or little host data movements, require less power than the DP cards. This can be explained by the fact that folded down DFAs are less efficient in terms of FPS/area, as seen in section 6.3.2.2. As a result, although the DP implementation does not have VNx instantiations, it requires over 70% more LUTs, over 45% more BRAMs and a few more FFs and DSPs than any of the MP implementations.

11.4 Discussion

11.4.1 Designer's Productivity

The evaluation shows that our partitioner and resource balancer ILP tool can improve several design quality measures of the tested large FINN CNN accelerators. Compared to using monolithic kernels, it achieved increased frequency through adequate floorplaning and reset logic, and also it enabled greater computational densities, in particular, targeting multi-node platforms and performing resource balancing with multi-version modules. In addition, when combined with model-parallelism, latency was further reduced, and lower power accelerators were obtained. Although only dataflow CNN implementations were used in our analysis, the principles that our tool uses to perform the optimizations are not application-dependent, but based aimed at solving the issues that emerge when scaling-up FPGA systems, presented in section 6.3. Therefore, we expect that its use will have a similar impact on other large FPGA designs that can expose a comparable module granularity and structure. We have tested pipelines that can be represented as somewhat-

linear directed graphs (as exposed by FINN, not internally), which are common in many FPGA applications like video, network packet, or signal processing pipelines. However, and particularly for pipelines with more and more complex interconnections, additional evaluation is required to confirm these results.

The designs presented by Petrica *et al.* [192] were partitioned manually. To perform these simpler partitions (single-node, single-version, ≈ 15 modules), it would take $\approx$ 30min each, which may need to be re-done due to changes in the design. This task is not only time-consuming for humans, but also we can hardly demonstrate that we have achieved an optimal solution except in elementary cases. In addition, this optimization is based on hardware concepts, like resource balancing, critical paths and high fan-out nets, limiting the capacity of developers without this lower level knowledge to obtain efficient FPGA implementations. The automatization of this task, not only made us more productive, obtaining the results in $\approx 1s$ for the presented systems, but also enabled the creation of larger systems and to perform more complex optimizations. At the same time, the tool achieved the best results given the optimization objective, providing high-quality results like those illustrated by fig. 11.6 for the complex multi-version implementation (see fig. 11.5).

11.4.2 Tool Scalability

Although the tool proved to be fast for designs many times larger than the presented implementations, it has to be noted that given our use of an exact ILP solver, the tool may not be sufficiently quick for designs with significantly more modules (for example, because of exposing a very fine granularity) and more FPGA nodes. This problem can be affronted by an initial approximate solution, for example, grouping modules, and thus, simplifying the optimization. This initial solution can be then iteratively improved by exposing finer granularity in the node boundaries, as implemented by the graph-partioning algorithm presented by Henzinger *et al.* [193]. Notice that the use of this latter step has its greatest impact for the partitions at the node level, but might not have a significant effect within a given node. This is because of our use of soft pBlocks (see section 11.2.1), and thus the P&R implementation step is allowed to move logic across pBlocks tiles if it considers it necessarily to meet the target frequency.

11.4.3 Design Methodology

To be able to apply the presented optimizations, we required to implement the accelerators as explicit interconnected modules (IP, kernels) as opposed to a monolithic accelerator (single module accelerator with an internal dataflow that only exposes input and output ports). Principally, this allowed to distribute the computational load among multiple nodes and regions, and having a distributed control and reset logic for modules implemented without the knowledge of the final layout. In addition, enabled to optimize, based on global restrictions, the balance among FPGA resources. Therefore, this modular design methodology permitted to circumvent the limitations of the available tools, and to decouple floorplan and resource balancing optimizations from the design of the accelerator logic. Alternatively, when only a monolithic design is available and using Vitis to implement the system, it is advisable to use a centered SLR reset to reduce the possible frequency penalties (see table 11.1).

11.5 Related work

11.5.1 Multi-SLR FPGA Partitioning and Optimization

FPGA design partitioning has been studied for multi-FPGA systems and different integrating technologies aiming at diverse objectives, including timing closure, temperature and routability. Here, we focus on multi-die, interposer-based FPGA optimizations targeting similar metrics.

Hahn *et al.* [194] proposed modified P&R cost functions, which improved the routability of circuits (reducing the number of wires required to successfully implement it) and slightly increasing speed when targeting multi-SLR FPGAs. That work was further improved by Nasiri *et al.* [195], incorporating graph partitioning for logic assignment to dies before placement, obtaining better routability. Furthermore, Lui *et al.* [196] optimized implementation total wire length by separating the problem into multi-die floorplanning and signal assignment steps, using an enumeration- and a network-flow-based algorithm, respectively. Mao *et al.* [197] floorplans a set of fixed geometry rectangular modules (more adequate for ASIC than FPGA design) to multi-die FPGAs using a two-step placement, using a force-directed method followed by simulated annealing to refine the solution, aiming for total delay optimization.

Kou *et al.* [198] designed an ILP-based algorithm to optimize the pin assignments, minimizing the number of SLR crossings for multi-FPGA designs with the goal of im-

proving operating frequency. Voss *et al.* [199] presented a greedy-based algorithm to facilitate timing closure of high-congestion implementations using multi-SLR devices. Their algorithm aims at limiting inter-die communication achieving improved implementation feasibility through balancing the physical memory resources (LUTRAM, BRAM, and URAM) of each separate module (circuit with high interconnection) of the design.

The main difference with our solution is that, while these tools need to maintain a cycle-accurate circuit, we may modify the underlying logic to improve timing closure. We connect modules to the appropriate control and reset controllers as a function of the resulting floorplan and insert pipeline registers or FIFO to cross SLRs. What is more, we perform resource balancing through partitioning and choosing module versions based on global objectives and constraints. In addition to this, we natively support multi-node platforms, minimizing the communication cost and limiting congestion globally.

11.5.2 Multi-node FPGA DNN Partitioning

In the state-of-the-art we can find many multi-node DNN FPGA implementations, for which, as in our case, tools were developed to distribute the load. We focus on those that distribute the hardware logic, as opposed to those that distribute the inference load to an array of identical processing engines.

Tarafdar *et al.*[200] uses a greedy algorithm to distribute DFA layers to FPGA nodes. Note that, although fast to implement and to run, a greedy partitioner will not generally provide an optimal solution, neither guaranties good ones. What is more, to obtain the useful TMP configuration, additional tiling of the nodes is required (to be able to come back to the original tile), which in-turn worsens the quality of the partition achievable by a greedy algorithm. Zhang *et al.* [201] presents a mapping approach for asymmetric multi-FPGA architectures using a dynamic programming-based partitioning that considers both resource and bandwidth constraints. They only support linear networks (any non-linear one is transformed to a linear), while we support any graph structure, including disjoint ones. This allowed us to optimize globally the implementation of multiple accelerators, combining data- and model-parallelism. In addition, they only perform monotonous partitions (not allowing to go back and forward between tiles), and thus, it has the same problems noted for the greedy partitioner. Zhang *et al.* [202] also proposed a dynamic programming algorithm. They create only linear maps (monotone) of CNN layers to a homogenous FPGA cluster connected in a ring network, either to minimize latency or maximize throughput.

Geng et al. [125] proposes FPDeep, which provides automatic partitioning and resource balancing of the training task graph to MPEs through a multistep approximation algorithm. Their algorithm assumes N equal FPGAs, for which N connected sub-graphs are generated, aiming at maximizing throughput using a non-linear optimization objective without apparent limitations to inter-FPGA communication requirements. Like in the previous cases, they only support monotonous partitions. Because of this, they resort to store parameters in a different node to the one where these parameters are used, which requires additional communication resources and power, plus it is limited to parameter memories balancing. FPDeep also performs compute resource balancing as a fine-tuning step, after the partitioning is completed, but only allows trade-off between two classes of resources.

Unlike those works, we do not impose restrictions on neither the compute platform, nor on the accelerator network structure. One consequence of this, is that our tool may perform intra-layer partitioning if the accelerator is exposed at that finer-grain level. The tool minimizes the communication cost (generally providing the best solution[2]), but any partition that fulfills the constraints is allowed (see figs. 11.4 and 11.5). When appropriately setting the cost matrix, the minimization of this cost results in increased computational density (see section 11.3.3). Also, we support heterogeneous nodes, resources and connections. What is more, our tool handles, simultaneously, the intra-FPGA (SLR) partitioning, and can accept multiple versions of a node allowing to further improve the computational density of the solution.

Compared to the implementations presented in the reviewed works plus other relevant ImageNet classification FPGA CNN accelerators found in the literature [203–205], our dual-node accelerators obtain the highest FPS (one of them , 3xMN in 2xU280, by a large margin). Note that a good quantitative comparison against other multi-node accelerator implementations is hindered by the fact that these designs differ in FPGA technology, CNN topology, language abstraction level, variable quantization, tensor sparsity, and implementation, software driver design and training effort. However, this result does support the quality of the partitions and the efficiency of the implemented circuits. In addition, our designs can be further improved, for example, incorporating more flexible scale-up through folding, optimized RTL components, or using techniques to employ resources more efficiently (e.g., the optimization by Petrica *et al.* [192] to reduce the OCM requirements of DFAs).

[2]Depending on the runtime allowed, the ILP solver might not prove that the solution is optimal.

11.6 Conclusion

We presented a partitioning and resource balancing tool, which solves a set of related FPGA implementation problems in a unified manner in a single-stage global optimization. In particular, it addresses the control set connection issue of large kernels in multi-SLR FPGAs, and balances multiple resources across FPGA regions and/or chips, partitioning the pipeline (allowing disjointed partitions in a region) and choosing between equivalent versions of pipeline modules with different resource profiles. This is done while minimizing the communication cost.

The application of this optimization increased the frequency and computational units per area of the tested MobileNetV1 and ResNet-50-v1.5 dataflow accelerators, demonstrating up to a 103% performance improvement over an already optimized monolithic implementation. At the same time, the tool natively maps systems to a multi-node implementation if it does not fit in a single FPGA, allowing to increase compute density further and when combined with model-parallelism, enables lower latency and more power-efficient implementations. Given the capabilities of the tool, we were able to implement transparent model-parallelism, which simplifies multi-node accelerator interfacing and also paves the way towards FPGA multi-tenancy.

To benefit from this optimization, it is necessary to move from a monolithic kernel design methodology to one where accelerators are constructed by explicit smaller multi-version kernels. As a result of its utilization, our design productivity was greatly increased by simultaneously improving the implementation quality of results and reducing design time.

Although it is expected that these results will extrapolate to other dataflow architectures, given that the developed optimization algorithm is generic, i.e. not based on application characteristics, further evaluation using a more diverse set of designs needs to be carried out to confirm this (informed) speculation. Given the generality of the algorithm, a very interesting venue of future work is to apply this optimization to heterogeneous platforms, that is, create hybrid (FPGA, CPU, GPU,...) mappings of accelerators.

Part IV

Putting All Together

CHAPTER 12

LESSONS LEARNED DESIGNING WITH HLS

In this chapter, surveying the literature in the matter and combining it with the results and experience obtained in the realization of this thesis, we discuss the benefits of using HLS and how to incorporate it in a hardware design methodology. As a result, a series of methodological aspects were identified as the most impactful. In particular, we observed that HLS can achieve great quality of results, leveraging wider algorithmic exploration and function specialization. Also, we identify the benefits of modular partitioning and refinement, and using a hardware oriented development mentality.

12.1 Introduction

In the different explorations done throughout this thesis, many HLS prototypes were built, although with different degrees of success. We reexamined what lead to success or failure, and even in some cases, reimplemented systems. In this process, we observed that, apart from the limitations HLS compilers still have, coding styles and design methodologies also explained the quality of results (QoR). Depending on the coding style, for example, the compiler might infer architectures with idle cycles (not processing data) or be unable to increase the amount of parallelism.

The impact on performance can be great, for example, in [206] tuning and refactoring

code of a JPEG encoder (initially based on a software version) achieved a 647X throughput improvement and a 221X latency reduction with only a 4X area increase. Singh *et al.* [207] demonstrated for a maximum string match procedure that a CPU efficient code translated into slow hardware. An alternative code, more suitable for static parallelization analysis, allowed to implement a higher performing, parallel architecture. This was because the compiler as not able to determine statically in the CPU code how much it could unroll loops with data dependent bounds. Given its impact, many source to source compiler optimizations have been proposed, among others, to improve scheduling by solving dependencies issues or by considering data access patterns [159–161, 168–171, 208].

What is more, coding styles affected our capacity to identify optimizations. This is because they sometimes they lead to implementations whose scheduling and architecture were harder to analyze, and also, it was harder to understand the issues that prevented achieving the desired performance from an algorithmic point of view. Conversely, an appropriate code structure allowed to detect the system inefficiencies more easily and even facilitated the circumvention of some limitations of the HLS compiler. For example, the use of non-monolithic accelerators allowed us to obtain global resource balancing, a reset structure aware of the physical layout (chapter 11), and have multiple clock domains for C++ modules (chapter 8).

In this chapter, surveying the literature in the matter and combining it with the results and experience obtained in the realization of this thesis, we discuss the benefits of using HLS and how to incorporate it in a hardware design methodology.

12.2 Incorporating HLS for Better Productivity

12.2.1 How HLS Improves Productivity

Given the limitations HLS compilers have, why using it? A recent literature review analyzed 46 studies that compared RTL versus HLS implementations of a given system in different application areas. They reach the conclusion that although RTL designs provided higher QoR, the productivity (performance/dev. time) using HLS was 4.4 times better than with RTL [14]. In typical designs, only a subset of procedures limits performance below requirements. Given that, from our experience and the mentioned literature review, the performance of HLS modules is often not far apart from RTL ones, we expect that HLS would provide many good enough modules for deployment, and thus, reducing development time and costs.

12.2.1.1 Performance Through Wider Algorithm Exploration

For a given design, the QoR obtained by HLS is an lower bound to what RTL can achieve. However, for a fixed development time, HLS not only allows a more comprehensive DSE, but in particular, a wider algorithm exploration. For example, the LOCO-ANS HLS implementation achieves the highest throughput JPEG-LS pixel near-lossless decorrelation. The discovery of an alternative (mathematically equivalent) algorithm to reconstruct the quantized pixel was one of the main reasons the previous RTL designs were superseded. This algorithm was not implemented in the initial design, but developed after confirming the location of the bottleneck. Once identified the optimization, a functional system was obtained and tested in the FPGA in a few hours. Table 12.1 shows the QoR impact of this optimization. As it can be seen, not only frequency increased by a 33%, but also most resource requirements decreased.

Conversely, in RTL design, more time is devoted to microarchitecture design, writing code, and performing verification, and therefore, leaving less time for algorithm design. In addition, algorithmic changes might not be properly considered after the RTL implementation, as they can require a great design and re-code effort. Alternatively, a thorough algorithmic optimization can be done before RTL development, but often this will imply spending time in parts of the system that have little impact on QoR. HLS allows avoiding these premature optimizations, enabling easier incorporation of later algorithmic changes.

Note that, although mathematical equivalencies provide a vast source of optimizations, there other sources. For example, using alternative data access patterns (e.g. loop tilling or reordering nested loops.), approximations (e.g. CORDIC [209]), using alternative numerical formats (e.g. using block floating point of the FFT [210]), or bit manipulations (e.g. the fast inverse square root [211]). De Fine Licht *et al.* [212] provides a set of transformations to HLS code to improve performance, particularly useful for high-performance computing applications[1].

Table 12.1: Frequency and resource variation due to the using of the alternative algorithm to reconstruct the quantized pixel

Part	Freq(%)	LUT (%)	FF(%)	BRAM (diff.)	DSP (diff.)
Z7020	33.2%	-11.3%	-4.0%	+1	-1
ZUS+	33.6%	-9.7%	-4.2%	+ 1	-1

[1]A small note on the work by de Fine Licht *et al.* [212]. Among others, they propose *loop fusion* as a pipeline enabling transformation. Considering the key aspects detailed in section 12.3.1, we encourage

12.2.1.2 Performance Through Module Specialization

Another important source of productivity when using HLS is module specialization. That is, having a module optimized for a given set of parameters and/or fixed inputs. Although RTL designs benefit from constant inputs through netlist optimizations like constant propagation, HLS compiler can make more thorough optimizations of the architecture. A good example of this is the LOCO-ANS decorrelator optimized just for lossless compression operation. The removal of the quantization logic, plus the logic simplification that arises from using only $NEAR = 0$, led to significant changes in the scheduling of the pipeline. This allowed going from an $II = 2$ to $II = 1$ with approximately a 25% frequency penalty in the tested technologies, that is about a 50% throughput increase. Given that just a few lines of C++ code needed to be modified, in less than one hour we could test the system in hardware.

In particular, leveraging C++ templates to ease these module specializations, improved QoR, rivaling or even superseded RTL, can be achieved [213, 214]. The FINN hlslib [117], employed and expanded during the development of the CNN accelerators, provides a good example of the use of templates in HLS designs. Here, templates are not only used for parameter customization but also as a way of decoupling the control path from the datapath, and thus, favoring code re-use. This is achieved using template class arguments to define datapath operations.

12.2.1.3 Boosting Productivity in Early Design Stages

HLS has also proved to be a very useful tool, even when it has not considered for deployment. First, being able to translate higher-level descriptions to hardware allows faster hardware-algorithm co-design (which), as it facilitates the obtaining real feedback. Also, it enables earlier system integration, and thus, among other benefits, enhancing parallel driver software debug and optimization [214]. In addition, it eases this system integration as HLS design becomes functionally correct faster than RTL ones [215]. Having an integrated functionally correct system available allows accurate determination of system bottlenecks. Also, it enables gradual integration of component refinements, and thus, it accelerates system optimization as the identification of the source of an issue is faster.

the use of dataflows instead of loop fusion for increasing throughput and routability. They recognize that routability may be negatively affected by this transformation, and thus, can carry frequency penalties. An example of how loop splitting enabled higher frequencies was presented in section 10.2. Loop fusion will normally have a slight latency and possibly an area advantage.

12.2.2 Software vs Hardware Development

Although the HLS uses languages that were originally designed for software or inspired by them, the same code styles and methodologies do not generally apply [212]. This is because, among other reasons, HLS does not target an ISA but a circuit, and thus, the implications of code might be fundamentally different. For this reason, vendors provide recommendations of how to best design for HLS [216, 217].

Sun *et al.* [206] also recognized that software and hardware developments achieve high performance through different means, and as a consequence, in some cases it was better to start from scratch than from an available SW code. Our development of a hardware accelerator of the LHE image encoder [218, 219] starting from a software implementation, was not nearly as successful as the development of the LOCO-ANS hardware encoder starting from scratch, supporting this observation. In this case, of course, not only the starting point affected the results, but also other methodological aspects.

12.2.3 Desired Methodology Characteristics

This led to the question of how to best incorporate HLS in the hardware design methodology. We previously mentioned some desired features of a methodology that uses HLS. To improve productivity, we need code that facilitates the detection of bottlenecks and that allows iterative refinement of the system [220]. As a part of this, it is useful to be able to effectively decouple problems, for example, through the use of latency-insensitive protocols [221].

Note that the best coding styles for a given problem is probably going to depend on the particular HLS tool [215]. For example, in [206] the technique that they found the less effective for the HLS compiler to share resources, was precisely the one that enabled us to share the ANS tables in the dual TSG coder. Therefore, part of the iterative refinement is dedicated to find the code styles better suited for the tool we employ. In addition, a good methodology should also enable to take advantage from using HLS, while being able to circumvent its limitations. Finally, given the mentioned productivity gains derived from using HLS, the methodology should minimize, when possible, the RTL code implemented.

12.3 Key Methodological Aspects

We already described how HLS boosts productivity in early design stages and how it can improve performance through wider algorithm exploration and module specialization. Although there are many other aspects that influence the success of a development that employs HLS, we focus on those that we identified, both in our experience and in the literature, as the most relevant and generally applicable. Of course, a methodology depends, at least in part, on the tools currently available. As HLS compilers are improved (for example incorporating optimizations proposed in this thesis) and the systems we aim to build change, how to best design with HLS will vary.

12.3.1 Modular Design and Refinement

Appropriate design modularization, both within an HLS generated IP and of a larger system, was one of the main enablers of productivity. Sun *et al.* [206] indicated that partitioning allowed to better express the data flow and reduce the complexity of optimizations. Among other ways, this is achieved by alleviating the data dependencies between functions, particularly when combined with FIFO channels between them. Also, they note that by splitting functions, they can create more efficient memory mappings customized for a given process, instead of a compromise among many of them. We extended this last observation to coding styles and basic compute element in the design of the TSG coder, as detailed in chapter 8.

Both Mantovani *et al.* [222] and Rokicki *et al.* [223], in their respective designs of a RISK V processor, described the benefits of partitioning the different processor stages to better express parallelism. In addition, the former work notes that when combined with latency-insensitive channels, it allows a richer DSE by independent optimization of each pipeline stage. These findings are inline with a previously presented design flow based on *modular refinement* by Dave *et al.* [220]. Here the authors also highlight that partitioning facilities code reuse, parallel system development and functional/performance debugging, which are also supported by de Fine Licht *et al.* [212]. Interestingly, the modular refinement methodology closely resembles many aspects of the implementation process of the TSG coder, as the example at the end of this section shows.

Sarkar *et al.* [215] also demonstrated that they were able to achieve better QoR synthesizing modules separately (non-monolithic design), given that this allowed tuning compiler's configurations for each specific module. As previously mentioned, we also found non-monolithic implementations useful, given that they allowed us to obtain global

resource balancing, implement multi-FPGA accelerators, a reset structure aware of the physical layout (chapter 11), have multiple clock domains for C++ modules (chapter 8), and easily mixing RTL and HLL (chapter 9).

Non-monolithic kernels have also advantages for debugging and optimization. First, consider that HLS compilation and RTL simulation have considerably longer run-times than software equivalent processes. Using non-monolithic kernels, or at least, setting the particular module we are interested in as the HLS top-level, can considerably reduce the compilation and RTL verification times, thus enabling faster development. In addition, these partitioned IPs allow us to analyze intermediate data using probes for debugging the implemented hardware.

In general, partitioning the design is a great divide-and-conquer technique as it allows faster debugging and decoupling dependencies, DSE, frequency, implementation strategies, placement, design teams, code styles, memory mappings and development language. Note, however, that it is convenient to keep dependency loops within a single module. This is because, partitioning tends to increase loop latency (decreasing throughput). For this reason, the HLS flow metering core and the pixel decorrelator were implemented using a single module.

12.3.1.1 Partitioning Example

The HLS implementation of the TSG coder (section 8.2.2) is a good example of how modularization enables performance. First, we isolated the non-sequential data use patterns required by the algorithm within modules, which allowed the rest of the system to access sequentially to data using FIFO interfaces. This was achieved by encapsulating

Algorithm 12.1: Codification procedure of a geometrically distributed symbol using tANS.

Require: $z, param$
1: $c \leftarrow get\_cardinality(param)$
2: $remaining\_sym \leftarrow z$
3: $subsym \leftarrow z \mod c$
4: **repeat**
5: $remaining\_sym \leftarrow remaining\_sym - subsym$
6: $obits \leftarrow ANS\_table[param][state][subsym].bits$
7: store_in_binary_stack($state, obits$)
8: $state \leftarrow ANS\_table[param][state][subsym].nx\_st$
9: $subsym \leftarrow c$
10: **until** $remaining\_sym = 0$

the symbol inversion and binary inversion processes in separate dataflow functions, the Block Buffer and the Binary Stack, resp. These modules were further partitioned in two concurrent processes, one to write streaming data to a memory and another to read the data from it, following the required pattern to generate the output data stream. As the internal memory is implemented as a ping-pong buffer, and these modules are capable of consuming and generating data concurrently and without major stalls.

Partitioning was also used to simplify control and debug performance, for example, in the implementation of the geometric coder, which alg. 12.1 illustrates (previously presented in chapter 7, alg. 7.1). We had the goal of obtaining an architecture capable of coding a new sub-symbol every cycle, which we knew it was possible after analyzing the algorithm. The straightforward code for this algorithm, however, resulted in low-performance hardware due to both frequency and idle cycle issues. To obtain an architecture capable of the desired performance, we iteratively refined the partitions

Algorithm 12.2: Algorithm 12.1 after loop splitting.

Require: $z, param$

// **Module 1**

1: $c \leftarrow get\_cardinality(param)$
2: $remaining\_sym \leftarrow z$
3: $subsym \leftarrow z \mod c$

// **Module 2**

4: **repeat**
5: $remaining\_sym \leftarrow remaining\_sym - subsym$
6: $end\_of\_loop \leftarrow remaining\_sym = 0$
7: $fifo1$.push($subsym, end\_of\_loop$) // Replaces following function calls
8: $subsym \leftarrow c$
9: **until** end_of_loop

// **Module 3**

10: **repeat**
11: $(subsym, end\_of\_loop) \leftarrow fifo1$.pop() // Replaces previous function calls
12: $obits \leftarrow ANS\_table[param][state][subsym].bits$
13: $fifo2$.push($state, obits, end\_of\_loop$) // Replaces following function calls
14: $state \leftarrow ANS\_table[param][state][subsym].nx\_st$
15: **until** end_of_loop

// **Module 4**

16: **repeat**
17: $state, obits, end\_of\_loop \leftarrow fifo2$.pop() // Replaces previous function calls
18: store_in_binary_stack($state, obits$)
19: **until** end_of_loop

decoupling procedures, and finally arriving at the design presented in section 8.2.2.

Loop splitting was one of the main tools used to decouple processes. That is, we distributed loop operations among new loops with identical bounds, using FIFOs to communicate all the data necessary to execute these new loops, as alg. 12.2 illustrates. This division is allowed because there are no loop-carried dependencies between them. Each of these loops were then encapsulated in different functions, and we indicated to the compiler that these run in parallel. This code structure better resembles the hardware we needed, and it allowed to address the issues of each of the split module separately. This partition also enabled to share efficiently the ANS state between the Geometric and the Bernoulli coders. In addition, using simpler modules tends to simplify control, which favors high operating frequency, as demonstrated in section 10.2, when evaluating the conditional stalling implementations.

Can this process be automated by the compiler? In some cases, it can, but in general, this process can be too complex to automatize, then manual transformation might still be required, as noted by Cardoso *et al.* [224]. There are other reasons to do this. First, if we know the architecture we want, this is a mechanism through which we can force it when directives are not enough. Second, it allows us to better identify the source and solutions to performance issues, and using the code styles and directives better suited for each of them. In particular, this enabled us to use more explicit FSM to replace the loop structures, which were the source of great inefficiencies given that they required one cycle to start and one cycle to end the variable-length loops. This is specially problematic for those procedures whose average loop count is small, like for the z decomposition process (Module 2 in alg. 12.2). The average loop count for it is between 1 and 1.3, and thus resulted in a $3 < \overline{\mathrm{II}} < 3.3$. The alternative implementation shown by alg. 12.3, was capable of emitting a new sub-symbol each clock cycle, which is what we needed. From a loop perspective, it resulted in $1 < \overline{\mathrm{II}} < 1.3$, that is, 2.5 to 3 times more throughput without significant area changes.

12.3.2 Architecture Design

12.3.2.1 Hardware Design Knowledge

To optimize software, it is important to understand the underlying architecture. For example, we need to know how caches work to better exploit the concepts of temporal

Algorithm 12.3: Efficient HLS implementation of Module 2 of algorithm 12.2 .

```
1: remaining_sym ← 0                                   // Initial configuration after reset
2: while true do                                                          // PIPELINE with II=1
3:   if remaining_sym = 0 then
4:     (remaining_sym, subsym, c, last_symb, ...) ← fifo_in.pop()
5:   end if
6:   remaining_sym ← remaining_sym − subsym
7:   last_subsymb ← last_symb and remaining_sym = 0
8:   fifo_out.push(subsym, last_subsymb)
9:   subsym ← c
10: end while
```

and spatial locality, and avoiding issues like *false sharing*[2]. The more we tune code for a particular architecture, the more performance we can extract. This will tend to come at the expense of code portability to other architectures. For example, super-scalar processors with deep pipelines are more affected by branch miss predictions, than shallow in-order processors. Because of this, in the former architecture, compiled code with more instructions, but fewer branches (or easier to predict) may run faster than a shorter code with more branches.

In the same way, we found that hardware design knowledge was necessary for identifying issues, coming up with solutions and taking design decisions. For example, to understand techniques like conditional stalling requires considering the variable time. Also, as hardware often uses pipeline implementations, the throughput of the system is determined the minimum throughput of all stages. Conversely, software run-time optimization tends to be better guided by Amdahl's law. What is more, frequency considerations are also important when stages of the dataflow have different II. In these cases, if a single clock is used, then we may find that frequency is limited by a stage with low II, thus, decreasing the performance of the stage with the highest II, and so, slowing down the whole system.

This last consideration describes the situation we faced with the LOCO-ANS HLS encoder, as the TSG coder has an $2 < \overline{\text{II}} < 2.3$, while the lossless decorrelator has an II = 1. However, TSG coder can operate at a frequency more than 2.3 times higher, and thus, using different clocks for each of them increased the system throughput by a factor of 2-2.3. An alternative would have been to set II = 2 for the lossless decorrelator to increase its clock frequency, but performance would have increased by a factor lower

[2]False sharing occurs when different cores of a processor utilize different addresses that are stored in the same cache line, invalidating each other's cache, although their data is not outdated.

than 2 due to pipelining penalties (see section 10.4.2). In any way, it is evident that this design decisions require hardware concepts.

Lahti *et al.* [14] also found in their RTL vs HLS survey that hardware expertise is still a requirement to achieve high QoR using current HLS tools. Nane *et al.* [225] reached a similar conclusion in their evaluation of four HLS compilers, even when the developers of three of them were authors of the work. Inline with this, Venkatakrishnan *et al.* [226] in their evaluation of two HLS tools, concluded that knowledge about FPGA and board architecture was a requirement to implement performance optimizations, although they used a simple convolution kernel as a benchmark. Silva *et al.* [227] evaluated FPGAs using HLS kernels for graph algorithms acceleration. They considered that good hardware knowledge is needed to obtain good performance, particularly for what they classified as *FPGA-unfriendly* domains (given the algorithm's structure).

Finally, note that if hardware knowledge is important for successful HLS developments, it is more so for hardware-algorithm co-design. Even if compilers continue to improve, understanding of the underlying technology is probably going to remain useful to take decisions about the algorithms.

12.3.2.2 Using RTL Analysis to Guide the HLS Implementation

When optimization modules, we found it useful to perform a coarse RTL design, that is, identifying main hardware resources and data movements. This was specially important when dependencies were present. For example, we used this technique to optimize the operating frequency of the LOCO-ANS decorrelator. A basic analysis at the RTL revealed that many operations in the critical path (context address computation) could be precomputed. After refactoring the code to have a more regular structure and to hint the compiler the architecture we wanted, we reached the alternative scheduling. As a result, frequency improved by 15% and area got slightly reduced. In this case, we only needed to rearrange the image row buffer updates to have it in a single point of the code (instead of two, one before the pixel loop and one at the end of it). Note that in this analysis, we did not design a detailed RTL, but only studied the processes involving the critical path, identifying feed-forward logic and dependencies, and considering the latency of BRAMs (which were part of this path). Therefore, it was much less time-consuming than a full RTL design and implementation.

Other works have found that using an RTL design to guide the HLS implementation resulted in better QoR (similar to that of the RTL). Canis *et al.* [228] compared hardware generated from a software code of a Solve filter against one guided by a hand-crafted RTL

design. As a result, the HLS code optimized for hardware was one order of magnitude faster than the software version. In [214], a first HLS design had a significantly greater footprint than a reference RTL. After an optimization phase, which involved using specific directives to achieve the same DSP mappings that the reference RTL design had, the HLS implementation achieved the same amount of DSPs and BRAM and lower LUT and FF. It has to be noted that these reported improvements came from matrix operations, which are already more amenable to automatic optimization given the regularity of their computations.

Sun *et al.* [206] also considered that having an RTL design in mind helps the processes of guiding the HLS tool towards better performing architectures. The previously mentioned RISC-V implementations, by Mantovani *et al.* [222] and Rokicki *et al.* [223], also used this middle-level design analysis to achieve QoR comparable to hand-crafted RTL. Finally, Özkan *et al.* [229] studied the methodological aspects of designing FPGA kernels with the Altera SDK for OpenCL, reaching the conclusion that better results were obtained using a hardware development mentality.

12.3.3 Interface Design

Being able to select an interface protocol just by using directives is one of the most clear benefits of HLS. Yet, it is important to keep in mind that it is hardware what we are describing. In software, performance is not generally a primary concern when creating function interfaces. They do have some impact, for example, they affect data locality, the need of doing data copies and the amount of memory required. In HLS, the implications are different and generally have a greater impact, particularly on throughput.

For both external and between modules, designing interfaces in HLS implies mainly two decisions, what protocol and what transaction unit to use. In general, we have to choose the protocol that better adjusts to the access pattern. For example, a memory interface can perform sequential reads, but a FIFO interface will better signal the procedure's access pattern to both the compiler and other designers. However, we have found useful to transform algorithms (e.g. reordering operations and/or splitting loops) to use streaming interfaces (FIFO-like). This is because streaming interfaces simplify dependency analysis and favors a code style better suited for hardware. This interfaces also allow to better decouple modules [206, 220, 223].

The second aspect of interface design is the transaction unit. The most direct impact of this decision is the available bandwidth, and therefore these generally need to be scaled as we modify the amount of parallelism we exploit. In our HLS design experience,

we find it useful to bundle the data that is processed together, e.g. using structs or explicit bit packets. These tend to reduce the required resources (e.g. using packed memories, instead of partitioned ones), simplify control (fewer signals to control data movements), and favor synchronization between the packed variables (e.g. not using metadata of element n with element $n+1$). Additionally, this generally lead to cleaner code.

Finally, to improve performance, sometimes it is necessary to add processes to convert between access patterns. The Block Buffer and the Binary Stack are examples of this, as previously indicated in section 12.3.1.1. Xilinx also recommends the load-compute-store pattern for kernels, meaning that apart from the compute processes, explicit parallel processes should be added to copy from and to main memory. This allows not only to adapt access patterns, but also to hide the large access latency and improve bandwidth through the use of large transaction bursts [230]. Interestingly, the addition of this code to better interface with main memory (accelerator card application) was recently automatized by Sohrabizadeh *et al.* [208].

12.3.4 Analysis of Compilation Results

12.3.4.1 HLS QoR Estimation Issues

When designing hardware and particularly during the DSE phase, we require feedback of the QoR each solution achieves. Yet, HLS tools systematically over-estimates resources and their timing model tends to be over-pessimistic. In our experience, we found that frequency estimations are particularly under-estimated for low congestion implementations, even when the data path was composed of a predesigned RTL IP (conditional stalling evaluation examples used in chapter 10), and can reach a 100% relative error, inline with Dai *et al.* [231] study of the matter. Regarding LUTs, HLS estimation, in normal circumstances, can also have a relative error over 100%, which may be explained by subsequent implementation steps, like constant propagation and LUT combining optimizations. DSP and BRAMs estimations are generally accurate, except when ROMs can be shared. Although HLS reports replicated ROMs, these may be latter combined during RTL synthesis, which happened in both LHE quantizer and TSG coder developments.

Of course, this inaccurate estimations have a negative impact on the DSE productivity HLS tools promise to provide. Therefore, several works have focused on improving them [231–233]. Notice, however, that the precision of these estimations is limited by the information available to the HLS tool. Final QoR depends on many factors not normally

known by the HLS tool, like synthesis and implementation optimization configurations, and interactions with the rest of the systems (e.g. congestion).

12.3.4.2 Recommendations

These reports have some utility, particularly when comparing different versions of a given module, but the considerations presented here need to be kept in mind during the design process. In addition, there is a set of recommendations that can help to palliate these issues. A relatively obvious one is to periodically use RTL synthesis to obtain better frequency and resource feedback. Although these may differ from final implementation results, we found that Vivado synthesis estimations are very accurate, and considerably faster to obtain compared to post place and route ones. Also, as mentioned before, prior to refactoring code so that shared ROMs are inferred, we recommend to first verify it on the post-synthesis results.

Nane *et al.* [225] suggested as a rule of thumb to provide a higher target frequency to the HLS compiler than the actual desired one, while Sarkar *et al.* [215] recommended doing target frequency sweeps. We found a combination of these useful, but combined with specific II requirements (and sometimes also latency ones) to prevent the compiler to increment the II in favor is higher probabilities to met timing (again, normally overly pessimistic). The use of these more restrictive directives was particularly important when dependencies were present, like in the flow metering core and image encoders. Until the last optimization steps of the design, we used a HLS target frequency equal to the desired one. Then, as a fine-tuning step, we performed target frequency sweeps above the desired frequency (using implementation feedback), to improve the operating frequency of the slowest module of the clock domain.

12.4 Summary

- Use HLS to build quicker a functionally correct integrated system, from which to obtain high-quality design feedback and upon which optimizations can be applied gradually.
- Iteratively partition systems into dataflow structures to better decouple and identify problems. Also, perform separate HLS compilations of dataflow modules for:
  - Higher hardware implementation visibility in combination with probes.
  - Improve performance:

    - Doing faster DSE (faster compilation and RTL simulation).
    - Balancing throughput using different clock domains.
    - Isolate a procedure that is not efficiently implemented in HLS, and replace it by an RTL IP.
    - Using partitioning and resource balancing optimizations, and getting at placement aware reset logic.
    - Tuning HLS compilation and RTL synthesis configuration for each module (Fine-tune step).

- Leverage function specialization (in particular using C++ templates) and late algorithmic optimizations to improve QoR.
- Use a hardware oriented design mentality to optimize logic and interfaces.
- During the DSE process, take into account the limited accuracy of the QoR estimations before RTL synthesis. In particular, use specific constraints on throughput and periodically use RTL synthesis for higher quality feedback

12.5 Conclusion

We analyzed the obtained results and employed methods in this thesis when designing with HLS. Combined with the literature on HLS design optimization methodologies, we identified key methodological aspects that lead to better quality of results. Considering these, a set of recommendations was compiled.

These recommendations are: (1) Use HLS for earlier system integration and gradual refinement. (2) Embrace modular partitioning and refinement for easier and more impactful optimization. (3) Design HLS code with a hardware design mentality. (4) Design interfaces with performance in mind, and favoring streaming interfaces. (5) Leverage wider algorithmic exploration and function specialization to obtain better hardware. (6) Take into account the limited accuracy of quality of result estimations before RTL synthesis.

CHAPTER 13

CONCLUSIONS

This chapter summarizes the research and development results of the thesis. Throughout this work, we aimed at contributing to the different stages of the hardware design process, focusing on FPGAs as a target, to better confront the computation challenges that we face. For this reason, the contributions span from algorithms to layout tools, passing through high-level design, low-level optimizations, and design methodologies. Appendix A presents the list of publications and their relationship with the chapters of the thesis.

13.1 Main Contributions

LOCO-ANS Algorithm Chapter 7 presented enhancements to the JPEG-LS standard, improving its coding efficiency at a low computational overhead, particularly for hardware implementations, resulting in the LOCO-ANS algorithm. LOCO-ANS features a novel adaptive coder for sources with a two-sided geometrical (TSG) distribution, designed for the prediction residuals resulting from the decorrelation process. This coder, using Tabled Asymmetric Numeral Systems as the underlying technology, has a complexity similar to a Huffman coder but with efficiencies that closely approach the model's entropy. Another key aspect resides in the distribution parameter estimation, where low complexity and more precise quantizers were studied and developed. The resulting codec

was evaluated in depth to understand the sources of inefficiencies and how these relate to its parameters. The software prototype plus auxiliary code to configure the codec and run experiments were open sourced to the community [126].

When compared to JPEG-LS baseline compressing photographic images, LOCO-ANS, using the same context size, achieves up to a 1.6%, 6% and 37.6% mean compression improvement for an error tolerance set to 0, 1 and 10, respectively. Allowing an increase in the context size and image tiling, a 2.3% lower bandwidth is obtained for lossless compression, also improving near-lossless compression. Moreover, LOCO-ANS approaches the lossless compression rates of more complex software encoders, like JPEG-XL and WebP, even surpassing them in near-lossless compression, while attaining a much faster encoder speed.

LOCO-ANS Hardware Encoder Chapter 8 described the developed LOCO-ANS encoder hardware architecture. The obtained results show that this hardware encoder excels in near-lossless compression, accomplishing the fastest pixel rate so far with up to 40.5 MPixel/s/lane for a low-end Zynq 7020 and 124.15 MPixel/s/lane for a Zynq Ultrascale+ device. At the same time, a balanced configuration of the presented encoder can achieve 7.4%, 16.7%, 25.1%, and 33.0% better compression than the previous fastest JPEG-LS-like near-lossless implementation (for an error tolerance in [0..3], respectively). In this way, the presented encoder can cope with higher image resolutions or FPS than previous near-lossless encoders while achieving higher compression and keeping encoding latency below 100 μs. Thus, it is a great tool for real-time video compression and, in general, for highly constrained scenarios like many remote sensing applications.

We consider that mainly two factors made these results possible. First, the algorithm-hardware co-design approach. In the decorrelator, for example, the modified gradient quantization (enabled by the TSG coder) not only significantly increases compression, but also relaxes the near-lossless implementation bottleneck, given that it reduces the logic depth. Another important example is that we took advantage of the fact that the decorrelator is the bottleneck of the system to increase the coder compression efficiency without reducing the system throughput. Yet, the new coder still had to achieve a high performance, and a trivial implementation of tANS would not suffice because of its memory requirements. Then, exploiting a property of the symbols' distribution and using an alternative TSG distribution model, we devised a coding method that keeps resources low, closely approaches to the model's entropy, and achieves high throughput. This development also gave rise, to our knowledge, to the first adaptive ANS hardware

encoder implementation in the literature.

Methodology for Hardware Design using High-Level Synthesis The other main reason for these great results was the HLS design, and the methodology behind it. Chapter 12 presents an analysis of the key methodological aspects that lead to better quality of results when using HLS. As a result, we consider modular partitioning and refinement as powerful tool that allows easier and more capable optimization, and also, that with currently available HLS compilers higher performing systems are obtained optimizing HLS logic and interfaces with a hardware design (RTL-like) mentality. In addition, although HLS enables a faster design space exploration, it is important to take into account the limited accuracy of the quality of result estimations before RTL synthesis. Finally, with HLS, we can leverage wider algorithmic exploration and function specialization to obtain better hardware.

Dual read-update architecture and 100GbE flow metering However, HLS compilers can still be improved. The LOCO-ANS hardware encoder can attain even higher performance using the architecture studied in chapter 9. This was designed to address the problem of implementing high-throughput and complex read-update processes, specially when dealing with significant propagation delays associated with the memory system. For slower link speeds, an HLS implementation could handle maximum packet rates, but given that network links bandwidth grow faster than silicon speed, we needed to implement this critical module using an optimized architecture and RTL code.

This architecture enabled to completely offload a 100GbE VPN volumetry application. Also, it allowed to implement an FPGA TCP flow metering system capable of supporting the maximum 100GbE packet rate with a single acceleration core using up to 64Ki flows cache with 8-ways. According to our experiments (chapter 5), this translates into a significant offload of the task, reducing the output packet rate and bandwidth to 40.6% and 45%, resp. In this way, we demonstrated the FPGA capabilities for processing 100GbE traffic with state (as opposed to stateless, packet-wise processing) and also proposed an architecture to improve performance of systems with the read-update pattern, specially when numerous on-chip memory blocks need to be cascaded and/or having two parallel memory update process is desired.

Study of conditional stalling However, the need for larger memories and more complex systems can inevitably lead to increased latencies in the read-update cycle,

establishing hard to optimize bottlenecks. For this reason, in chapter 10 we studied the conditional stalling technique, a throughput optimization for logic with data dependencies. This dynamic scheduling method consists of running a pipeline at full rate when no conflicts are detected, using scheduling logic to insert bubbles to ensure that no dependency is violated. We analyzed the footprint and frequency penalties associated with this optimization, reaching the conclusion that most designs will not reduce their frequency and that few extra resources are required.

We also examined the performance when using this method as a function of input data and architecture characteristics, showing that it is specially useful when unavoidable latencies are present in the dependency path. Also, we demonstrated that to correctly optimize throughput, we must take into account both the address statistics and the evolution of frequency as the pipeline is deepened. What is more, it could be integrated within HLS compilers, which can use the models provided to make design decisions, resulting in better quality of results and increased designers' productivity. This optimization is compatible with the high-performance architecture developed for the read-update pattern (chapter 5), so their improvements add up.

Automatic partitioning and resource balancing In chapter 9, we also presented arrays of flow metering cores which enable scaling up flow tables, increasing the offloading capabilities, while still supporting the maximum packet rate. However, as observed for the large convolutional neural networks implementations described in chapter 6, frequency penalties appeared as we crossed the boundary of a monolithic region of a chip (SLR). To address this issue along with a set of related problems in a unified manner, in chapter 11 we presented a partitioning and resource balancing optimization tool. In particular, it addresses the control set connection of large designs in multi-SLR FPGAs, and balances multiple resources across FPGA regions and/or chips, while it minimizes the communication cost among them. The tool natively maps systems to a multi-node implementation if it does not fit in a single FPGA, and it is capable of choosing (or letting the user choose) among different types of model-parallelism (MP), including the presented transparent MP, which abstracts the complexities of handling a multi-node accelerator.

The application of this optimization increased the frequency and computational units per area of the tested MobileNetV1 and ResNet-50-v1.5 dataflow accelerators, demonstrating up to a 103% performance improvement over an already optimized monolithic implementation. Targeting multi-node platforms allowed to increase compute

density, and when combined with model-parallelism, enabled lower latency and more power-efficient implementations. As a result of its utilization, our design productivity was greatly increased by simultaneously improving the quality of results and reducing design time.

13.2 Future Work

In this thesis, several optimizations to HLS have been studied and tested for particular contexts. Despite the partitioner and resource balancing tool being included in the FINN NN compiler, an important future work is to integrate the proposed optimizations in HLS compilers. In this way, these will have their greatest impact.

Before this integration, although we consider that these optimizations have a wide application, further evaluation using a more diverse set of designs needs to be carried out to confirm whether the obtained results extrapolate to other contexts. For example, the architecture presented in chapter 9, could be applied, among others, to the LOCO-ANS hardware encoder, and applications based on histogram computation. Also, conditional stalling has the potential of increasing the mean throughput of the LOCO-ANS lossless encoder.

Furthermore, these optimizations can be pushed forward. Conditional stalling could be analyzed in combination with other techniques like out-of-order processing and merging sequential dependent operations. Additionally, the pipeline optimization performed by conditional stalling requires the address statistics to remain unchanged. Therefore, it would be interesting to study the implementation of an adaptive system with multiple processing units of varying depth (and clock frequency), choosing at runtime the higher throughput alternative according to the collected conflict statistics. As for the partitioner, given the generality of the algorithm, a very interesting venue of future work is to apply this optimization to heterogeneous platforms, that is, create hybrid (FPGA,CPU,GPU,...) mappings of accelerators.

Finally, on the application side, the presented 100GbE flow metering system employed on-chip memory, which is not enough to completely offload the task. Then, the next natural step is to use off-chip memory, like HBM. However, this will not be immediate if significant packet rate penalties are to be avoided, given the large latencies these memories have. We consider that the architecture optimizations suggested in chapter 9, in combination with techniques like conditional stalling, should be evaluated for this goal. In addition to these, the exploration of flow cache policies (to determine what to

keep on-chip), for example based on heavy hitter flow detection, can also be key, not only for 100 GbE, but for the new higher speed links support.

CHAPTER 14

CONCLUSIONES

Este capítulo resume los resultados de investigación y desarrollo de la tesis. A lo largo de este trabajo, hemos buscado contribuir a las diferentes etapas del proceso de diseño de hardware, centrándonos en las FPGAs como objetivo, para afrontar mejor los retos de computación a los que nos enfrentamos. Por esta razón, las contribuciones abarcan desde algoritmos hasta las herramientas de diseño, pasando por diseños de alto nivel, las optimizaciones de bajo nivel y las metodologías de diseño. En el Apéndice A se presenta la lista de publicaciones y su relación con los capítulos de la tesis.

14.1 Principales Contribuciones

Algoritmo LOCO-ANS El capítulo 7 presentó modificaciones al estándar JPEG-LS, mejorando su eficiencia de codificación con una baja sobrecarga computacional, en particular para las implementaciones de hardware, dando como resultado el algoritmo LOCO-ANS. LOCO-ANS presenta un novedoso codificador adaptativo para fuentes con una distribución geométrica de doble lado (TSG, por sus siglas en inglés), diseñado para los símbolos resultantes del proceso de decorrelación. Este codificador, que utiliza sistemas numéricos asimétricos tabulados (tANS, por sus siglas en inglés) como tecnología subyacente, tiene una complejidad similar a la de un codificador Huffman pero con

eficiencias que se acercan mucho a la entropía del modelo. Otro aspecto clave reside en la estimación de los parámetros de distribución, donde se estudiaron y desarrollaron cuantificadores de baja complejidad y mayor precisión. El codec resultante se evaluó en profundidad para entender las fuentes de ineficiencia y cómo estas se relacionan con sus parámetros. El prototipo software y el código auxiliar para configurar el codec y ejecutar los experimentos se pusieron a disposición de la comunidad [147].

En comparación con la configuración de base de JPEG-LS para la compresión de imágenes fotográficas, LOCO-ANS, utilizando el mismo tamaño de contexto, consigue una mejora media de 1,6%, 6% y 37,6% en la compresión para una tolerancia de error establecida en 0, 1 y 10, respectivamente. Permitiendo un aumento del tamaño del contexto y el particionamiento de la imagen, se obtiene una mejora del 2,3% para la compresión sin pérdidas, mejorando también la compresión casi sin pérdidas. Además, LOCO-ANS se aproxima a las tasas de compresión sin pérdidas de codificadores de software más complejos, como JPEG-XL y WebP, superándolos incluso en la compresión casi sin pérdidas, al tiempo que alcanza una velocidad de codificación mucho mayor.

Codificador hardware LOCO-ANS El capítulo 8 describe la arquitectura de hardware del codificador LOCO-ANS desarrollado. Los resultados obtenidos muestran que este codificador hardware destaca en la compresión casi sin pérdidas (*near-lossless*), logrando la tasa de píxeles más rápida hasta el momento con hasta 40,5 MPixel/s/vía para un Zynq 7020 (gama baja) y 124,15 MPixel/s/vía para un dispositivo Zynq Ultrascale+. Al mismo tiempo, una configuración equilibrada del codificador presentado puede lograr una compresión un 7,4%, 16,7%, 25,1% y 33,0% mejor que la anterior implementación más rápida de tipo JPEG-LS near-lossless (para una tolerancia al error en [0..3], respectivamente). De este modo, el codificador presentado puede hacer frente a resoluciones o FPS más elevados que los codificadores anteriores, a la vez que consigue una mayor compresión y mantiene la latencia de codificación por debajo de 100 μs. Por lo tanto, es una gran herramienta para la compresión de vídeo en tiempo real y, en general, para escenarios muy limitados como muchas aplicaciones de sensores remotos.

Consideramos que principalmente dos factores han hecho posible estos resultados. En primer lugar, el enfoque de co-diseño algoritmo-hardware. En el decorrelador, por ejemplo, la modificación de la cuantización de los gradientes (habilitada por el codificador TSG) no solo aumenta significativamente la compresión, sino que también relaja el cuello de botella de la implementación, dado que reduce la profundidad lógica. Otro ejemplo importante es que aprovechamos que el decorrelador es el cuello de botella del sistema

para aumentar la eficiencia de compresión del codificador sin reducir el rendimiento del sistema. Aun así, el nuevo codificador tenía que alcanzar un alto rendimiento, y una implementación trivial de tANS no sería suficiente debido a sus requisitos de memoria. Luego, explotando una propiedad de la distribución de los símbolos y utilizando un modelo de distribución TSG alternativo, ideamos un método de codificación que mantiene los recursos bajos, se aproxima a la entropía del modelo y logra un alto rendimiento. Este desarrollo también dio lugar, hasta donde sabemos, a la primera implementación de codificador hardware adaptativo ANS en la literatura.

Metodología de diseño de hardware usando síntesis de alto nivel La otra razón principal de estos excelentes resultados fue el diseño usando síntesis de alto nivel (HLS, por sus siglas en inglés) y la metodología detrás de él. En el capítulo 12 se presenta un análisis de los aspectos metodológicos clave que conducen a una mejor calidad de los resultados cuando se utiliza HLS. Como resultado, consideramos que el particionamiento y refinamiento modular es una poderosa herramienta que permite una optimización más fácil y capaz, y también, que con los compiladores HLS actualmente disponibles se obtienen sistemas de mayor rendimiento optimizando la lógica e interfaces HLS con una mentalidad de diseño hardware. Además, aunque HLS permite una exploración más rápida del espacio de diseño, es importante tener en cuenta la limitada precisión de la calidad de las estimaciones de resultados antes de la síntesis RTL. Por último, con HLS podemos aprovechar una exploración algorítmica más amplia y la especialización de funciones para obtener un mejor hardware.

Arquitectura de doble lectura-actualización y medición de flujos de 100GbE Sin embargo, los compiladores HLS aún pueden mejorarse. El codificador de hardware LOCO-ANS puede alcanzar un rendimiento incluso mayor utilizando la arquitectura estudiada en el capítulo 9. Ésta fue diseñada para abordar el problema de la implementación de procesos de lectura-actualización complejos y de alto rendimiento, especialmente cuando se enfrentan retrasos de propagación significativos asociados al sistema de memoria. Para velocidades de enlace más lentas, una implementación HLS podía manejar las tasas máximas de paquetes, pero dado que el ancho de banda de los enlaces de red crece más rápido que la velocidad del silicio, necesitábamos implementar este módulo crítico utilizando una arquitectura optimizada y código RTL.

Esta arquitectura posibilitó la descarga completa de una aplicación de volumetría VPN de 100 GbE. También permitió implementar un sistema de medición de flujos

TCP en FPGA capaz de soportar la máxima tasa de paquetes de 100 GbE con un único núcleo de aceleración que utiliza una caché de flujos de hasta 64Ki con 8 vías. Según nuestros experimentos (capítulo 5), esto se traduce en una importante descarga de la tarea, reduciendo la tasa de paquetes de salida y el ancho de banda al 40,6% y al 45%, respectivamente. De este modo, demostramos las capacidades de la FPGA para el procesamiento con estado del tráfico de 100 GbE (en contraposición al procesamiento sin estado, paquete a paquete) y también propusimos una arquitectura para mejorar el rendimiento de los sistemas con el patrón de lectura-actualización, especialmente cuando es necesario poner en cascada numerosos bloques de memoria dentro del chip y/o se desea tener dos procesos de actualización de memoria en paralelo.

Estudio del bloque condicional Sin embargo, la necesidad de memorias más grandes y sistemas más complejos puede conducir inevitablemente a un aumento de las latencias en el ciclo de lectura-actualización, estableciendo cuellos de botella difíciles de optimizar. Por este motivo, en el capítulo 10 estudiamos la técnica de bloqueo condicional, una optimización del rendimiento para la lógica con dependencias de datos. Este método de planificación dinámica consiste en ejecutar un pipeline a pleno rendimiento cuando no se detectan conflictos, utilizando la lógica de planificación para insertar burbujas que garanticen que no se viole ninguna dependencia. Analizamos los recursos y las penalizaciones de frecuencia asociadas a esta optimización, llegando a la conclusión de que la mayoría de los diseños no verán reducida su frecuencia y que se necesitan pocos recursos adicionales.

También examinamos el rendimiento al utilizar este método en función de las características de los datos de entrada y de la arquitectura, mostrando que es especialmente útil cuando hay latencias inevitables en el camino de dependencia. Además, demostramos que para optimizar correctamente el rendimiento hay que tener en cuenta tanto las estadísticas de direcciones como la evolución de la frecuencia a medida que se profundiza en el pipeline. Además, podría integrarse en los compiladores HLS, que pueden utilizar los modelos proporcionados para tomar decisiones de diseño, lo que redundaría en una mejor calidad de los resultados y en una mayor productividad de los diseñadores. Esta optimización es compatible con la arquitectura de alto rendimiento desarrollada para el patrón de lectura-actualización (capítulo 5), por lo que sus mejoras se suman.

Partición automática y equilibrio de recursos En el capítulo 9, también presentamos arreglos de núcleos de medición de flujos que permiten escalar las tablas de

flujo, aumentando las capacidades de descarga, sin dejar de soportar la tasa máxima de paquetes. Sin embargo, como se observó en las implementaciones de grandes redes neuronales convolucionales descritas en el capítulo 6, aparecieron penalizaciones de frecuencia a medida que cruzábamos el límite de una región monolítica del chip (SLR, por sus siglas en inglés). Para abordar esta cuestión junto con un grupo de problemas relacionados de forma unificada, en el capítulo 11 presentamos una herramienta de optimización de particionamiento y equilibrio de recursos. En particular, determina la conexión de las señales de control de grandes diseños en FPGAs multi-SLR, y equilibra múltiples recursos a través de las regiones y/o chips FPGA, al tiempo que minimiza el coste de comunicación entre ellos. La herramienta mapea de forma nativa los sistemas a una implementación multi-nodo si no caben en una sola FPGA, y es capaz de elegir (o dejar que el usuario elija) entre diferentes tipos de paralelismo de modelo, incluyendo el paralelismo de modelo transparente presentado, el cual abstrae las complejidades de manejar un acelerador multi-nodo.

La aplicación de esta optimización aumentó la frecuencia y las unidades computacionales por área de los aceleradores de segmentados de MobileNetV1 y ResNet-50-v1.5, demostrando una mejora del rendimiento de hasta el 103% respecto a una implementación monolítica ya optimizada. La utilización de plataformas multi-nodales permitió aumentar la densidad de cálculo y, cuando se combinó con el paralelismo de modelo, permitió una latencia más baja y unas implementaciones más eficientes desde el punto de vista energético. Como resultado de su utilización, la productividad se incrementó notablemente, mejorando simultáneamente la calidad de los resultados y reduciendo el tiempo de diseño.

14.2 Trabajo Futuro

En esta tesis se han estudiado y probado varias optimizaciones para HLS en contextos particulares. A pesar de que la herramienta de particionamiento y de equilibrio de recursos se incluyó en el compilador de redes neuronales FINN, un importante trabajo futuro es integrar las optimizaciones propuestas en los compiladores HLS. De este modo, estas tendrán su mayor impacto.

Antes de esta integración, aunque consideramos que estas optimizaciones tienen una amplia aplicación, es necesario realizar una evaluación adicional utilizando un conjunto más diverso de diseños para confirmar si los resultados obtenidos se extrapolan a otros contextos. Por ejemplo, la arquitectura presentada en el capítulo 9, podría aplicarse,

entre otros, al codificador hardware LOCO-ANS, y a aplicaciones basadas en el cálculo de histogramas. Asimismo, el bloqueo condicional tiene también el potencial de aumentar el rendimiento medio del codificador sin pérdidas LOCO-ANS.

Por otra parte, estas optimizaciones pueden mejorarse. El bloque condicional podría analizarse en combinación con otras técnicas como el procesamiento fuera de orden y la fusión de operaciones dependientes secuenciales. Además, la optimización del pipeline realizada por el bloqueo condicional requiere que las estadísticas de direcciones no cambien. Por lo tanto, sería interesante estudiar la implementación de un sistema adaptativo con múltiples unidades de procesamiento de diferente profundidad (y frecuencia de reloj), eligiendo en tiempo de ejecución la alternativa de mayor rendimiento en función de las estadísticas de conflictos recogidas. En cuanto al particionador, dada la generalidad del algoritmo, una línea de trabajo futura muy interesante es aplicar esta optimización a plataformas heterogéneas, es decir, crear mapeos híbridos (FPGA, CPU, GPU, ...) de aceleradores.

Por último, en lo que concierne a las aplicaciones, el sistema de medición de flujo de 100 GbE presentado emplea memoria en el chip, lo que no es suficiente para descargar completamente la tarea. Entonces, el siguiente paso natural es utilizar memoria fuera del chip, como la HBM. Sin embargo, esto no será inmediato si se quieren evitar penalizaciones significativas en la tasa de paquetes, dadas las grandes latencias que tienen estas memorias. Consideramos que las mejoras de la arquitectura sugeridas en el capítulo 9, en combinación con técnicas como el bloqueo condicional, deben ser evaluadas para este objetivo. Además de esto, la exploración de políticas de caché de flujo (para determinar qué mantener en el chip), por ejemplo, basadas en la detección de flujos largos, también puede ser clave, no solo para 100 GbE, sino para el nuevo soporte de enlaces de mayor velocidad.

APPENDIX A

LIST OF PUBLICATIONS

A.1 Publications Included in the Contents of this Thesis

A.1.1 Journals & Magazines

1. **Alonso, T.**, Sutter, G.,& López de Vergara, J. E. (2021). **An FPGA-Based LOCO-ANS Implementation for Lossless and Near-Lossless Image Compression Using High-Level Synthesis**. Electronics, 10(23), 2934.

   This publication is part of **chapter 4** (Image Processing: Compression in Constrained Scenarios) and **chapter 8** (LOCO-ANS Image Codec: Hardware Implementation). Also, it contributes to **chapter 12** (Lessons Learned Designing with HLS).

   Journal information:

   - ISSN:2079-9292
   - Indexed in JCR 2020. Impact Factor: **2.397**. **Q3 (T2)** in Engineering, Electrical & Electronic (145 out of 273). **Q3 (T2)** in Computer Science, Information Systems (93 out of 161).

- Indexed in Scimago 2020. SJR: **0.36**. **Q2** in Computer Networks and Communications. **Q2** Electrical and Electronic Engineering. **Q3** in Hardware and Architecture. **Q3** in Signal Processing.

2. **Alonso, T.**, Sutter, G., & López de Vergara, J. E. (2021). **LOCO-ANS: An optimization of JPEG-LS using an efficient and low complexity coder based on ANS**, in IEEE Access, vol. 9, pp. 106606-106626, 2021, doi: 10.1109/ACCESS.2021.3100747.

   This publication is part of **chapter 4** (Image Processing: Compression in Constrained Scenarios) and **chapter 7** (LOCO-ANS Image Codec: Algorithm).

   Journal information:

   - ISSN:2169-3536
   - Indexed in JCR 2020. Impact Factor: **3.367**. **Q2 (T2)** in Engineering, Electrical & Electronic (94 out of 273). **Q2 (T2)** in Computer Science, Information Systems (65 out of 161). **Q2 (T2)** in Telecommunications (36 out of 91).
   - Indexed in Scimago 2020. SJR: **0.587**. **Q1** in Computer Science. **Q1** in Engineering.

3. **Alonso, T.**, Petrica, L., Ruiz, M., Petri-Koenig J., Umuroglu Y., Stamelos I., Koromilas E., Blott, M. & Vissers K. (2021). **Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning**. ACM Trans. Reconfigurable Technol. Syst. 15, 2, Article 15 (June 2022), 34 pages. doi:10.1145/3470567

   This publication is part of **chapter 6** (AI: Implementation of Large CNN Accelerators) and **chapter 11** (Automatic Partitioning and Resource Balancing). Also, it contributes to **chapter 12** (Lessons Learned Designing with HLS).

   Journal information:

   - ISSN: 1936-7414
   - Indexed in JCR 2020. Impact Factor: **2.085**. **Q3 (T2)** in Computer Science, Hardware & Architecture (30 out of 53).
   - Indexed in Scimago 2020. SJR: **0.355**. **Q2** in Computer Science.

A.1.2 International Conferences

4. **Alonso, T.**, Ruiz, M., Sutter, G., López-Buedo, S., & López de Vergara, J. E. (2019, April). **Towards 100 GbE FPGA-Based Flow Monitoring**. In 2019 X Southern

Conference on Programmable Logic (SPL), Buenos Aires, Argentina (pp. 9-16). IEEE.

This publication is part of **chapter 5** (Computer Networks: 100 GbE Flow Metering) and **chapter 9** (100GbE Flow Metering & Dual Read-Update Architectures).

Conference information:

- Indexed in Scimago 2020. SJR: **0.11**.

A.2 Other Publications Related to this Thesis

A.2.1 International Conferences

5. Petrica, L., **Alonso, T.**, Kroes, M., Fraser, N., Cotofana, S., & Blott, M. (2020, December). **Memory-efficient dataflow inference for deep CNNs on FPGA**. In 2020 International Conference on Field-Programmable Technology (FPT), Maui, HI, USA (pp. 48-55). IEEE.

   This publication is related to **chapter 6** (AI: Implementation of Large CNN Accelerators).

   Conference information:

   - GGS Class: "**Work in Progress**"; GSS Rating "**B-**".

6. **Alonso, T.**, Ruiz, M., García-Arias, Á. L., Sutter, G., & López de Vergara, J. E. (2018, August). **Submicrosecond Latency Video Compression in a Low-End FPGA-based System-on-Chip**. In 2018 28th International Conference on Field Programmable Logic and Applications (FPL), Dublin, Ireland (pp. 355-3554). IEEE.

   This publication is related to **chapter 12** (Lessons Learned Designing with HLS) and it was the starting point of the developments of **chapter 7** (LOCO-ANS Image Codec: Algorithm) and **chapter 8** (LOCO-ANS Image Codec: Hardware Implementation).

   Conference information:

   - ISSN: 1946-1488.
   - Indexed in Scimago 2020. SJR: **0.27**.
   - GGS Class: "**2**"; GSS Rating "**A-**".

A.2.2 Other Communications

7. **Alonso, T.**, Ruiz, M., Sutter, G., Sisterna, C., López-Buedo, S., & López de Vergara, J. E.. (2018). **Monitorización con FPGAs de flujos y sesiones TCP en enlaces de 40 Gbit/s**. In III Jornadas de Computación Empotrada y Reconfigurable (JCER2018), Teruel, Spain.

   This publication is related to **chapter 5** (Computer Networks: 100 GbE Flow Metering) and **chapter 9** (100GbE Flow Metering & Dual Read-Update Architectures).

APPENDIX

LOCO-ANS COMPRESSION EXAMPLES

Figure B.1: Picture of a flower (8-bit, 2268x1512) from the Rawzor dataset. Lossless compression bpp using LOCO-ANS (Nt6_Stcg7_ANS6): 1.983

Figure B.2: Decoded picture of a flower (8-bit, 2268x1512) from the Rawzor dataset using LOCO-ANS (Nt6_Stcg7_ANS6) with $NEAR = 3$. bpp: 0.251. PSNR: 44.16 dB

Figure B.3: Picture of traffic (8-bit, cropped to 2048x1320, and converted to gray) from the Challenge on learned image compression (CLIC) dataset (Available: `http://compression.cc/tasks/`). Lossless compression bpp using LOCO-ANS (Nt6_Stcg7_ANS6): 3.524

Figure B.4: Decoded picture of traffic (8-bit, cropped to 2048x1320, and converted to gray) from the CLIC dataset using LOCO-ANS (Nt6_Stcg7_ANS6) with $NEAR = 3$. bpp: 1.122. PSNR: 42.91 dB

BIBLIOGRAPHY

[1] N. N. Group., "Nielsen's law of internet bandwidth," https://www.nngroup.com/articles/law-of-bandwidth/, accessed: 2022-3-31.

[2] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2021.

[3] K. Rupp, "48 years of microprocessor trend data," Jul. 2020, 2020 update of the popular chart hosted at https://github.com/karlrupp/microprocessor-trend-data. [Online]. Available: https://doi.org/10.5281/zenodo.3947824

[4] G. E. Moore, "Cramming more components onto electronic circuits," *Electronics Magazine*, vol. 4, 1965.

[5] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[6] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, 1967, p. 483–485. [Online]. Available: https://doi.org/10.1145/1465482.1465560

[7] J. L. Gustafson, "Reevaluating Amdahl's Law," *Commun. ACM*, vol. 31, no. 5, p. 532–533, may 1988. [Online]. Available: https://doi.org/10.1145/42411.42415

[8] J. Henkel, H. Khdr, S. Pagani, and M. Shafique, "New trends in dark silicon," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[9] S. P. E. Corporation, "SPEC CPU2017 Results," https://www.spec.org/cpu2017/results/, accessed: 2022-04-01.

[10] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after moore's law?" *Science*, vol. 368, no. 6495, p. eaam9744, 2020. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aam9744

[11] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, p. 181, 2008.

[12] D. A. P. John L. Hennessy, "A New Golden Age for Computer Architecture," https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext, accessed: 2022-04-01.

[13] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2015.

[14] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019.

[15] A. V. Aho and J. E. Hopcroft, *The Design and Analysis of Computer Algorithms*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1974.

[16] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, no. 7, p. 321, jul 1961. [Online]. Available: https://doi.org/10.1145/366622.366644

[17] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring). New York, NY, USA: Association for Computing Machinery, 1968, p. 307–314. [Online]. Available: https://doi.org/10.1145/1468075.1468121

[18] S. M. S. Trimberger, "Three ages of fpgas: A retrospective on the first thirty years of fpga technology: This paper reflects on how moore's law has driven the design of fpgas through three epochs: The age of invention, the age of expansion, and the age of accumulation," *IEEE Solid-State Circuits Magazine*, vol. 10, no. 2, pp. 16–29, 2018.

[19] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.

[20] T. Alonso, G. Sutter, and J. E. López de Vergara, "LOCO-ANS: An Optimization of JPEG-LS Using an Efficient and Low-Complexity Coder Based on ANS," *IEEE Access*, vol. 9, pp. 106 606–106 626, 2021.

[21] T. Alonso, G. Sutter, and J. E. López de Vergara, "An fpga-based loco-ans implementation for lossless and near-lossless image compression using high-level synthesis," *Electronics*, vol. 10, no. 23, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/23/2934

[22] A. Kiely and M. Klimesh, "The ICER progressive wavelet image compressor," *IPN Progress Report*, vol. 42, no. 155, pp. 1–46, 2003.

[23] S. J. Visser, A. S. Dawood, and J. A. Williams, "FPGA based satellite adaptive image compression system," *Journal of Aerospace Engineering*, vol. 16, no. 3, pp. 129–137, 2003.

[24] B. Sushma, "Endoscopic wireless capsule compressor: A review of the existing image and video compression algorithms," in *Sustainable Communication Networks and Application*, P. Karuppusamy, I. Perikos, F. Shi, and T. N. Nguyen, Eds. Singapore: Springer Singapore, 2021, pp. 275–293.

[25] Q. Al-Shebani, P. Premaratne, P. J. Vial, and D. J. McAndrew, "The development of a clinically tested visually lossless image compression system for capsule endoscopy," *Signal Processing: Image Communication*, vol. 76, pp. 135–150, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092359651830290X

[26] P. Turcza and M. Duplaga, "Low-power image compression for wireless capsule endoscopy," in *2007 IEEE International Workshop on Imaging Systems and Techniques*, Cracovia, Poland, May 2007, pp. 1–4.

[27] X. Li, X. Chen, X. Xie, G. Li, Li Zhang, C. Zhang, and Z. Wang, "A low power, fully pipelined jpeg-ls encoder for lossless image compression," in *2007 IEEE International Conference on Multimedia and Expo*, 2007, pp. 1906–1909.

[28] G. Iddan, G. Meron, A. Glukhovsky, and P. Swain, "Wireless capsule endoscopy," *Nature*, vol. 405, no. 6785, pp. 417–417, 2000.

[29] M. R. Lone, "A high speed and memory efficient algorithm for perceptually-lossless volumetric medical image compression," *Journal of King Saud University - Computer and Information Sciences*, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157820303499

[30] T. Richter, J. Keinert, S. Foessel, A. Descampe, G. Rouvroy, and J.-B. Lorent, "JPEG-XS—A High-Quality Mezzanine Image Codec for Video Over IP," *SMPTE Motion Imaging Journal*, vol. 127, no. 9, pp. 39–49, 2018.

[31] New Infrared Technologies, "TACHYON 16k CAMERA," https://www.niteurope.com/wp-content/uploads/2017/01/TACHYON_16k_CAMERA_NIT.pdf, accessed: 2021-09-28.

[32] Y. Nagamatsu, F. Sugai, K. Okada, and M. Inaba, "Basic Implementation of FPGA-GPU Dual SoC Hybrid Architecture for Low-Latency Multi-DOF Robot Motion Control," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 7255–7260.

[33] H. Saidi, M. Turki, Z. Marrakchi, A. Obeid, and M. Abid, "Implementation of reed solomon encoder on low-latency embedded fpga in flexible soc based on arm processor," in *2020 International Wireless Communications and Mobile Computing (IWCMC)*, 2020, pp. 1347–1352.

[34] X. Zhang, X. Wei, Q. Sang, H. Chen, and Y. Xie, "An efficient fpga-based implementation for quantized remote sensing image scene classification network," *Electronics*, vol. 9, no. 9, 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/9/1344

[35] L. Li, S. Zhang, and J. Wu, "Efficient object detection framework and hardware architecture for remote sensing images," *Remote Sensing*, vol. 11, no. 20, 2019. [Online]. Available: https://www.mdpi.com/2072-4292/11/20/2376

[36] C. A. Lee, S. D. Gasster, A. Plaza, C.-I. Chang, and B. Huang, "Recent developments in high performance computing for remote sensing: A review," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 4, no. 3, pp. 508–527, 2011.

[37] International Telecommunication Union, "Information technology – lossless and near-lossless compression of continuous-tone still images: Baseline (itu-t t. 87—iso/iec 14495-1)," International Telecommunication Union, Tech. Rep., 1998.

[38] Ming Yang and N. Bourbakis, "An overview of lossless digital image compression techniques," in *48th Midwest Symposium on Circuits and Systems, 2005.*, 2005, pp. 1099–1102 Vol. 2.

[39] M. J. Weinberger, G. Seroussi, and G. Sapiro, "From LOCO-I to the JPEG-LS standard," in *Proceedings 1999 International Conference on Image Processing (Cat. 99CH36348)*, vol. 4, 1999, pp. 68–72 vol.4.

[40] L. Chen, L. Yan, H. Sang, and T. Zhang, "High-Throughput Architecture for Both Lossless and Near-lossless Compression Modes of LOCO-I Algorithm," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 29, no. 12, pp. 3754–3764, 2019.

[41] Y. Murat, "Key architectural optimizations for hardware efficient jpeg-ls encoder," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2018, pp. 243–248.

[42] L. Kau and S. Lin, "High performance architecture for the encoder of jpeg-ls on sopc platform," in *SiPS 2013 Proceedings*, 2013, pp. 141–146.

[43] P. Merlino and A. Abramo, "A Fully Pipelined Architecture for the LOCO-I Compression Algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 7, pp. 967–971, 2009.

[44] M. Ferretti and M. Boffadossi, "A parallel pipelined implementation of LOCO-I for JPEG-LS," in *Proceedings of the 17th International Conference on Pattern Recognition*, vol. 1, 2004, pp. 769–772.

[45] M. Klimesh, V. Stanton, and D. Watola, "Hardware implementation of a lossless image compression algorithm using a field programmable gate array," *Mars (Pathfinder)*, vol. 4, no. 4.69, pp. 5–72, 2001.

[46] International Telecommunication Union, "Information technology - lossless and near-lossless compression of continuous-tone still images: Extensions (ITU-T T. 870—ISO/IEC 14495-21)," International Telecommunication Union, Tech. Rep., 2003.

[47] J. Duda, "Asymmetric numeral systems," *CoRR*, vol. abs/0902.0271, 2009. [Online]. Available: http://arxiv.org/abs/0902.0271

[48] ——, "Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding," *CoRR*, vol. abs/1311.2540, 2013. [Online]. Available: http://arxiv.org/abs/1311.2540

[49] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp, "The use of asymmetric numeral systems as an accurate replacement for huffman coding," in *2015 Picture Coding Symposium (PCS)*. IEEE, 2015, pp. 65–69.

[50] International Telecommunication Union, "Digital compression and coding of continuous tone still images - requirements and guidelines, (itu-t t.81—iso/iec is 10918-1)," International Telecommunication Union, Tech. Rep., 1993.

[51] G. Roelofs, "Portable Network Graphics (PNG)," http://www.libpng.org/pub/png/libpng.html, accessed: 2021-03-16.

[52] M. J. Weinberger, G. Seroussi, and G. Sapiro, "The LOCO-I lossless image compression algorithm: Principles and standardization into JPEG-LS," *IEEE Transactions on Image processing*, vol. 9, no. 8, pp. 1309–1324, 2000.

[53] S. Golomb, "Run-length encodings," *IEEE Trans. Inform. Theory*, vol. vol. IT-12, pp. 399–401, 1966.

[54] X. Wu, N. Memon, and K. Sayood, "A context-based, adaptive, lossless/nearly-lossless coding scheme for continuous-tone images," *ISO/IEC JTC 1/SC 29/WG*, vol. 1, 1995.

[55] X. Wu and N. Memon, "Context-based, adaptive, lossless image coding," *IEEE transactions on Communications*, vol. 45, no. 4, pp. 437–444, 1997.

[56] J. J. Rissanen, "Generalized kraft inequality and arithmetic coding," *IBM Journal of Research and Development*, vol. 20, no. 3, pp. 198–203, 1976.

[57] S. M. Najmabadi, Z. Wang, Y. Baroud, and S. Simon, "High throughput hardware architectures for asymmetric numeral systems entropy coding," in *2015 9th international symposium on image and signal processing and analysis (ISPA)*. IEEE, 2015, pp. 256–259.

[58] S. M. Najmabadi, H. S. Tungal, T. Tran, and S. Simon, "Hardware-based architecture for asymmetric numeral systems entropy decoder," in *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2017, pp. 1–6.

[59] S. M. Najmabadi, T.-H. Tran, S. Eissa, H. S. Tungal, and S. Simon, "An architecture for asymmetric numeral systems entropy decoder-a comparison with a canonical huffman decoder," *Journal of Signal Processing Systems*, vol. 91, no. 7, pp. 805–817, 2019.

[60] Rawzor, "Rawzor test images," http://imagecompression.info/test_images/, accessed: 2020-07-30.

[61] T. Richter, "libjpeg implementation," https://github.com/thorfdbg/libjpeg, accessed: 2021-06-15.

[62] N. Merhav, G. Seroussi, and M. J. Weinberger, "Coding of sources with two-sided geometric distributions and unknown parameters," *IEEE Transactions on Information Theory*, vol. 46, no. 1, pp. 229–236, 2000.

[63] P. G. Howard and J. S. Vitter, "Practical implementations of arithmetic coding," in *Image and text compression*. Springer, 1992, pp. 85–112.

[64] T. Alonso, M. Ruiz, G. Sutter, S. López-Buedo, and J. E. López De Vergara, "Towards 100 gbe fpga-based flow monitoring," in *2019 X Southern Conference on Programmable Logic (SPL)*, 2019, pp. 9–16.

[65] T. Zseby, B. Claise, J. Quittek, and S. Zander, "Requirements for IP Flow Information Export (IPFIX)," RFC 3917, Oct. 2004. [Online]. Available: https://www.rfc-editor.org/info/rfc3917

[66] Cisco Inc., "Cisco Netflow Collection Engine," https://www.cisco.com/en/US/products/sw/netmgtsw/ps1964/products_implementation_design_guide09186a00800d6a11.html, accessed: 2019-02-06.

[67] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An overview of ip flow-based intrusion detection." *IEEE Communications Surveys and Tutorials*, vol. 12, no. 3, pp. 343–356, 2010.

[68] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, "Deriving traffic demands for operational ip networks: Methodology and experience," *IEEE/ACM Transactions on Networking (ToN)*, vol. 9, no. 3, pp. 265–280, 2001.

[69] A. Callado, C. Kamienski, G. Szabó, B. P. Gero, J. Kelner, S. Fernandes, and D. Sadok, "A survey on internet traffic identification," *IEEE communications surveys & tutorials*, vol. 11, no. 3, 2009.

[70] C. Vega, P. Roquero, and J. Aracil, "Multi-gbps http traffic analysis in commodity hardware based on local knowledge of tcp streams," *Computer Networks*, vol. 113, pp. 258–268, 2017.

[71] G. Sadasivan, N. Brownlee, B. Claise, and J. Quittek, "Architecture for IP Flow Information Export," RFC 5470, Mar. 2009. [Online]. Available: https://www.rfc-editor.org/info/rfc5470

[72] R. Hofstede, P. Čeleda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2037–2064, 2014.

[73] Cisco Inc., "Flexible NetFlow Configuration Guide," https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/fnetflow/configuration/xe-16/fnf-xe-16-book.html, accessed: 2019-01-06.

[74] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 3, pp. 270–313, 2003.

[75] E. Miravalls-Sierra, D. Muelas, J. Ramos, J. E. López de Vergara, D. Morató, and J. Aracil, "Online detection of pathological tcp flows with retransmissions in high-speed networks," *Computer Communications*, vol. 127, pp. 95–104,

2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366417307284

[76] P. Roquero, J. Ramos, V. Moreno, I. González, and J. Aracil, "High-speed TCP Flow Record Extraction Using GPUs," *The Journal of Supercomputing*, vol. 71, no. 10, pp. 3851–3876, 2015.

[77] M. Forconesi, G. Sutter, S. Lopez-Buedo, and J. Aracil, "Accurate and Flexible Flow-based Monitoring for High-speed Networks," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–4.

[78] V. Puš, P. Velan, L. Kekely, J. Kořenek, and P. Minařík, "Hardware Accelerated Flow Measurement of 100 Gb Ethernet," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE, 2015, pp. 1147–1148.

[79] J. L. Garcia-Dorado, J. A. Hernandez, J. Aracil, J. E. Lopez de Vergara, F. J. Monserrat, E. Robles, and T. P. de Miguel, "On the duration and spatial characteristics of internet traffic measurement experiments," *IEEE Communications Magazine*, vol. 46, no. 11, pp. 148–155, 2008.

[80] J. L. García-Dorado and J. Aracil, "Flow-concurrence and bandwidth ratio on the internet," *Computer Communications*, vol. 136, pp. 43–52, 2019.

[81] Cisco Inc., "Cisco IOS Flexible NetFlow Command Reference," https://www.cisco.com/c/en/us/td/docs/ios/fnetflow/command/reference/fnf_book/fnf_01.html, accessed: 2019-02-06.

[82] CAIDA, "The CAIDA UCSD Anonymized Internet Traces - 2018," http://www.caida.org/data/passive/passive_dataset.xml, accessed: 2019-02-06.

[83] T. Alonso, L. Petrica, M. Ruiz, J. Petri-Koenig, Y. Umuroglu, I. Stamelos, E. Koromilas, M. Blott, and K. Vissers, "Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 2, dec 2021. [Online]. Available: https://doi.org/10.1145/3470567

[84] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[85] Z. Ouyang, J. Niu, Y. Liu, and M. Guizani, "Deep cnn-based real-time traffic light detector for self-driving vehicles," *IEEE Transactions on Mobile Computing*, vol. 19, no. 2, pp. 300–313, 2020.

[86] C. Zhang, R. Li, W. Kim, D. Yoon, and P. Patras, "Driver behavior recognition via interwoven deep convolutional neural nets with multi-stream inputs," *IEEE Access*, vol. 8, pp. 191 138–191 151, 2020.

[87] D. N. Thang, L. A. Nguyen, P. T. Dung, T. D. Khoa, N. H. Son, N. T. Hiep, P. Van Nguyen, V. D. Truong, D. H. Toan, N. M. Hung, T.-D. Ngo, and X.-T. Truong, "Deep learning-based multiple objects detection and tracking system for socially aware mobile robot navigation framework," in *2018 5th NAFOSTED Conference on Information and Computer Science (NICS)*, 2018, pp. 436–441.

[88] R. Pereira, T. Barros, L. Garrote, A. Lopes, and U. J. Nunes, "An experimental study of the accuracy vs inference speed of rgb-d object recognition in mobile robotics," in *2020 29th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, 2020, pp. 588–595.

[89] A. Koul, S. Ganju, and M. Kasam, *Practical Deep Learning for Cloud, Mobile, and Edge*. O'Reilly Media, Inc., 2020.

[90] F. H. Araujo, R. R. Silva, F. N. Medeiros, D. D. Parkinson, A. Hexemer, C. M. Carneiro, and D. M. Ushizima, "Reverse image search for scientific data within and beyond the visible spectrum," *Expert Systems with Applications*, vol. 109, pp. 35–48, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417418302987

[91] Y. Zhang, P. Pan, Y. Zheng, K. Zhao, Y. Zhang, X. Ren, and R. Jin, "Visual search at alibaba," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, ser. KDD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 993–1001. [Online]. Available: https://doi.org/10.1145/3219819.3219820

[92] T. Kim, S. Kim, S. Na, H. Kim, M. Kim, and B.-K. Jeon, "Visual fashion-product search at sk planet," *arXiv preprint arXiv:1609.07859*, 2016.

[93] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions," *Journal of big Data*, vol. 8, no. 1, pp. 1–74, 2021.

[94] N. Tajbakhsh, J. Y. Shin, S. R. Gurudu, R. T. Hurst, C. B. Kendall, M. B. Gotway, and J. Liang, "Convolutional neural networks for medical image analysis: Full training or fine tuning?" *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1299–1312, 2016.

[95] C. J. Burke, P. D. Aleo, Y.-C. Chen, X. Liu, J. R. Peterson, G. H. Sembroski, and J. Y.-Y. Lin, "Deblending and classifying astronomical sources with Mask R-CNN deep learning," *Monthly Notices of the Royal Astronomical Society*, vol. 490, no. 3, pp. 3952–3965, 10 2019. [Online]. Available: https://doi.org/10.1093/mnras/stz2845

[96] E. Postnikov, A. Kryukov, S. Polyakov, and D. Zhurov, "Deep learning for energy estimation and particle identification in gamma-ray astronomy," *arXiv preprint arXiv:1907.10480*, 2019.

[97] I. Priyadarshini and V. Puri, "A convolutional neural network (cnn) based ensemble model for exoplanet detection," *Earth Science Informatics*, vol. 14, no. 2, pp. 735–747, 2021.

[98] A. A. Pol, G. Cerminara, C. Germain, M. Pierini, and A. Seth, "Detector monitoring with artificial neural networks at the cms experiment at the cern large hadron collider," *Computing and Software for Big Science*, vol. 3, no. 1, pp. 1–13, 2019.

[99] M. Paltenghi, "Time Series Anomaly Detection for CERN Large-Scale Computing Infrastructure," Ph.D. dissertation, Politecnico di Milano, Oct 2020, presented 02 Oct 2020. [Online]. Available: https://cds.cern.ch/record/2752641

[100] D. Paul and P. Sala, "Real-time server monitoring and cnn inference on fpga," CERN, Tech. Rep., 2019. [Online]. Available: https://openlab.cern/sites/default/files/2019-11/Report_Debdeep_Paul.pdf

[101] S. Li and X. Zhao, "Image-based concrete crack detection using convolutional neural network and exhaustive search technique," *Advances in Civil Engineering*, vol. 2019, 2019.

[102] B. Kim, N. Yuvaraj, K. Sri Preethaa, and R. Arun Pandian, "Surface crack detection using deep learning with shallow cnn architecture for enhanced computation," *Neural Computing and Applications*, vol. 33, no. 15, pp. 9289–9305, 2021.

[103] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[104] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[105] G. Cloud, "Cloud TPU," https://cloud.google.com/tpu, accessed: 2022-04-08.

[106] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating cnn inference on fpgas: A survey," *arXiv preprint arXiv:1806.01683*, 2018.

[107] X. Yu, Y. Wang, J. Miao, E. Wu, H. Zhang, Y. Meng, B. Zhang, B. Min, D. Chen, and J. Gao, "A data-center fpga acceleration platform for convolutional neural networks," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 151–158.

[108] J. Knapheide, B. Stabernack, and M. Kuhnke, "A high throughput MobileNetV2 FPGA implementation based on a flexible architecture for depthwise separable convolution," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020, pp. 277–283.

[109] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A configurable cloud-scale DNN processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 1–14.

[110] J. Fowers, K. Ovtcharov, M. K. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "Inside project brainwave's cloud-scale, real-time ai processor," *IEEE Micro*, vol. 39, no. 3, pp. 20–28, 2019.

[111] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs," in *2016 IEEE 24th Annual International*

Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2016, pp. 40–47.

[112] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[113] M. Ghasemzadeh, M. Samragh, and F. Koushanfar, "Rebnet: Residual binarized neural network," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 57–64.

[114] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[115] X. R. Labs, "FINN: Dataflow compiler for QNN inference on FPGAs," https://github.com/Xilinx/finn, accessed: 2021-01-07.

[116] A. Pappalardo, "Brevitas: Quantization-aware training in PyTorch," https://github.com/Xilinx/brevitas, accessed: 2021-01-07.

[117] X. R. Labs, "Brevitas: Quantization-aware training in PyTorch," https://github.com/Xilinx/finn-hlslib, accessed: 2021-01-07.

[118] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[119] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[120] X. R. Labs, "FINN dataflow accelerator examples," https://github.com/Xilinx/finn-examples, accessed: 2021-01-07.

[121] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," 2018.

[122] K. Saban, "Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency," https://docs.xilinx.com/v/u/en-US/wp380_Stacked_Silicon_Interconnect_Technology, accessed: 2022-04-08.

[123] J. F. Wakerly, *Digital design : principles and practices*, 4th ed. Upper Saddle River: Pearson Prentice Hall, 2006.

[124] i. Xilinx, "UltraFast Design Methodology Timing Closure Quick Reference Guide," https://docs.xilinx.com/v/u/en-US/ug1292-ultrafast-timing-closure-quick-reference, accessed: 2022-04-08.

[125] T. Wang, T. Geng, A. Li, X. Jin, and M. Herbordt, "Fpdeep: Scalable acceleration of cnn training on deeply-pipelined fpga clusters," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1143–1158, 2020.

[126] T. Alonso, "LOCO-ANS hardware implementation repository," https://github.com/hpcn-uam/LOCO-ANS-HW-coder, accessed: 2022-03-27.

[127] R. F. Rice, "Some practical universal noiseless coding techniques," Jet Propulsion Lab., Tech. Rep., 1979. [Online]. Available: https://ntrs.nasa.gov/citations/19790014634

[128] C. Team, "Charls implementation of jpeg-ls," https://github.com/team-charls/charls, commit: 8d12c74e8400b268042ca3bee8a32513181f94dc.

[129] T. Alonso, "CALIC implementation," https://github.com/Tobi-Alonso/gcif/tree/master/refs/calic, accessed: 2021-06-15.

[130] International Telecommunication Union, "Information technology – jpeg 2000 image coding system: Core coding system (itu-t t.800 | iso/iec 15444-1)," International Telecommunication Union, Tech. Rep., 2019.

[131] Joint Photographic Experts Group, "jpeg2000 implementation," https://github.com/uclouvain/openjpeg, commit: cc1919b183f76d5ac79cc9927fb899b47700d925.

[132] International Telecommunication Union, "Information technology – jpeg 2000 image coding system: High-throughput jpeg 2000 (itu-t t.814 | iso/iec 15444-15)," International Telecommunication Union, Tech. Rep., 2019.

[133] Joint Photographic Experts Group, "High throughput jpeg2000 implementation," https://github.com/aous72/OpenJPH, tag: 0.7.3.

[134] The WebM Project, “Webp homepage,” https://developers.google.com/speed/webp, accessed: 2021-06-15.

[135] ——, “libwebp implementation,” https://github.com/webmproject/libwebp, commit: f6d2924757f356fcdc620ddaf4c200728a78df09.

[136] ——, “libwebp2 implementation,” https://chromium.googlesource.com/codecs/libwebp2, commit: f1632cfe6c624fedb2fd47d137da0a2f5fe1fd42.

[137] Joint Photographic Experts Group, “Overview of JPEG XL,” https://jpeg.org/jpegxl/index.html, accessed: 2021-06-15.

[138] A. Rhatushnyak, J. Wassenberg, J. Sneyers, J. Alakuijala, L. Vandevenne, L. Versari, R. Obryk, Z. Szabadka, E. Kliuchnikov, I.-M. Comsa, K. Potempa, M. Bruse, M. Firsching, R. Khasanova, R. van Asseldonk, S. Boukortt, S. Gomez, and T. Fischbacher, “Committee draft of JPEG XL image coding system,” 2019.

[139] Joint Photographic Experts Group, “JPEG XL Reference Software,” https://gitlab.com/wg1/jpeg-xl, commit: f2ed004456dc7102b08e34fbcef08e1ef18f374d.

[140] Workshop and Challenge on Learned Image Compression, “Clic training dataset,” http://compression.cc/tasks/, accessed: 2021-06-15.

[141] Joint Photographic Experts Group, “CLIC images subset,” https://drive.google.com/drive/folders/1wMgmjf54iN46dVihvMnHhGk8oQT7a8Nd, accessed: 2021-06-15.

[142] A. V. Lotov and K. Miettinen, “Main terminology and notations used,” in *Multiobjective optimization*. Springer, 2008, ch. Preface,7, pp. X–XI.

[143] J. Alakuijala, J. Sneyers, L. Versari, and J. Wassenberg, “Jpeg white paper: Jpeg xl image coding system,” Joint Photographic Experts Group, Tech. Rep., 2021. [Online]. Available: http://ds.jpeg.org/whitepapers/jpeg-xl-whitepaper.pdf

[144] T. Liebchen, T. Moriya, N. Harada, Y. Kamamoto, and Y. A. Reznik, “The mpeg-4 audio lossless coding (als) standard-technology and applications,” in *Proc. 119th AES Conv*, 2005.

[145] Xiph.Org Foundation, “FLAC - Free Lossless Audio Codec,” https://xiph.org/flac/, accessed: 2021-04-9.

[146] H. Daryanavard, O. Abbasi, and R. Talebi, "Fpga implementation of jpeg-ls compression algorithm for real time applications," in *2011 19th Iranian Conference on Electrical Engineering*, 2011, pp. 1–4.

[147] T. Alonso, "LOCO-ANS software implementation repository," https://github.com/hpcn-uam/LOCO-ANS, accessed: 2021-03-16.

[148] Y. M. Mert, "Fpga-based jpeg-ls encoder for onboard real-time lossless image compression," in *Satellite data compression, communications, and processing XI*, vol. 9501. International Society for Optics and Photonics, 2015, p. 950106.

[149] M. Forconesi, G. Sutter, S. López-Buedo, J. E. López de Vergara, and J. Aracil, "Bridging the gap between hardware and software open source network developments," *IEEE Network*, vol. 28, no. 5, pp. 13–19, 2014.

[150] Forconesi, Marco, "Flexible NetFlow Configuration Guide," https://github.com/forconesi/HW-Flow-Based-Monitoring/tree/master/hls_approach/flow_surveyor, accessed: 2022-03-04.

[151] Xilinx Inc. SDNet PX Programming Language. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1016-px-programming

[152] B. Trammell and E. Boschi, "Bidirectional Flow Export Using IP Flow Information Export (IPFIX)," RFC 5103, Jan. 2008. [Online]. Available: https://www.rfc-editor.org/info/rfc5103

[153] R. Bush and R. Elz, "Serial Number Arithmetic," RFC 1982, Aug. 1996. [Online]. Available: https://www.rfc-editor.org/info/rfc1982

[154] Xilinx Inc., "Xilinx virtex ultrascale+ fpga vcu118 evaluation board," https://www.xilinx.com/products/boards-and-kits/vcu118.html, accessed: 2018-10-26.

[155] ——, "Alveo u200 data center accelerator card," https://www.xilinx.com/products/boards-and-kits/alveo/u200.html, accessed: 2022-3-25.

[156] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, p. 345–420, dec 1994.

[157] P. P. Chang, W. Y. Chen, S. A. Mahlke, and W.-m. W. Hwu, "Comparing static and dynamic code scheduling for multiple-instruction-issue processors," in *Proc.*

24th Annual International Symposium on Microarchitecture, ser. MICRO 24. New York, NY, USA: Association for Computing Machinery, 1991, p. 25–33.

[158] M. Griebl, P. Feautrier, and C. Lengauer, "Index set splitting," *Int. Journal of Parallel Programming*, vol. 28, no. 6, pp. 607–631, 2000.

[159] A. Morvan, S. Derrien, and P. Quinton, "Efficient nested loop pipelining in high level synthesis using polyhedral bubble insertion," in *2011 International Conference on Field-Programmable Technology*, 2011, pp. 1–10.

[160] J. Liu, S. Bayliss, and G. A. Constantinides, "Offline synthesis of online dependence testing: Parametric loop pipelining for hls," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 159–162.

[161] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides, "Polyhedral-based dynamic loop pipelining for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1802–1815, 2018.

[162] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[163] A. Nicolau, "Run-time disambiguation: coping with statically unpredictable dependencies," *IEEE Transactions on Computers*, vol. 38, no. 5, pp. 663–678, 1989.

[164] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-m. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proc. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI. New York, NY, USA: Association for Computing Machinery, 1994, p. 183–193.

[165] SIEMENS, "Catapult HLS," https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/, accessed: 2022-02-01.

[166] Intel High Level Synthesis Compiler Pro Edition: Reference Manual. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683349/21-4/pro-edition-reference-manual.html

[167] Xilinx, *Vitis High-Level Synthesis: User Guide*, Xilinx.

[168] J. Rohde, K. Müller, and C. Hochberger, "Improving hls generated accelerators through relaxed memory access scheduling," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 74–81.

[169] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–10.

[170] S. Derrien, T. Marty, S. Rokicki, and T. Yuki, "Toward speculative loop pipelining for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4229–4239, 2020.

[171] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang, "Dynamic hazard resolution for pipelining irregular loops in high-level synthesis," in *Proc. 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 189–194.

[172] T. Alonso, "Publication repository," https://github.com/hpcn-uam/hls-conditional-stalling, accessed: 2022-02-01.

[173] M. Weinberger, G. Seroussi, and G. Sapiro, "The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS," *IEEE Trans. Image Processing*, vol. 9, no. 8, pp. 1309–1324, 2000.

[174] R. T. Fernholz and R. Fernholz, "The universality of zipf's law for time-dependent rank-based random systems," *arXiv preprint arXiv:1707.04285*, 2017.

[175] M. A. Kader, E. Bastug, M. Bennis, E. Zeydan, A. Karatepe, A. S. Er, and M. Debbah, "Leveraging big data analytics for cache-enabled wireless networks," in *2015 IEEE Globecom Workshops (GC Wkshps)*, 2015, pp. 1–6.

[176] S. L. Harris and D. M. Harris, "7 - microarchitecture," in *Digital Design and Computer Architecture*, S. L. Harris and D. M. Harris, Eds. Boston: Morgan Kaufmann, 2016, pp. 384–484.

[177] K. Olukotun, L. Hammond, and J. Laudon, *Chip multiprocessor architecture: techniques to improve throughput and latency*. Morgan & Claypool Publishers, 2007, vol. 2.

[178] T. Alonso and L. Petrica, "Partitioner and resource balancer tool implementation in finn," https://github.com/Xilinx/finn-experimental/blob/main/src/finn/analysis/partitioning.py, accessed: 2021-06-06.

[179] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[180] Xilinx Inc, "Xilinx ai engine technology," available online at:https://www.xilinx.com/products/technology/ai-engine.html. Accessed: 2022-04-11.

[181] H. G. S. Tulio A. M. Toffolo, "The python MIP package," https://www.python-mip.com/, accessed: 2021-01-07.

[182] M. Ruiz, "XUP Vitis network example (VNx)," https://github.com/Xilinx/xup_vitis_network_example, accessed: 2021-01-07.

[183] X. U. Program, "Xilinx adaptive compute cluster (XACC) program," https://www.xilinx.com/support/university/XUP-XACC.html, accessed: 2021-01-07.

[184] Xilinx Inc, "Xilinx Runtime Library (XRT)," https://www.xilinx.com/products/design-tools/vitis/xrt.html, accessed: 2022-04-12.

[185] ——, "PYNQ: Python productivity for Xilinx platforms," https://pynq.readthedocs.io/en/latest/, accessed: 2022-04-12.

[186] Dask core developers, "Dask," https://dask.org/, accessed: 2022-04-12.

[187] C. Kachris, "Performance evaluation of InAccel ML scalable suite," InAccel, Tech. Rep., 2018. [Online]. Available: https://www.inaccel.com/wp-content/uploads/inaccel_white_paper.pdf

[188] Xilinx Inc, "Alveo Data Center Accelerator Card Platforms," https://docs.xilinx.com/v/u/en-US/ug1120-alveo-platforms, accessed: 2022-04-12.

[189] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[190] BROADCOM, "Single-Port 100 Gb/s QSFP56 Ethernet PCI Express 4.0 x16 OCP 3.0 SFF Network Adapter," https://docs.broadcom.com/doc/957504-N1100G-DS, accessed: 2022-04-12.

[191] NVIDIA, "NVIDIA Mellanox ConnectX-5 Ethernet Adapter Cards User Manual," https://img-en.fs.com/file/user_manual/connectx-5-ethernet-adapter-cards-user-manual.pdf, accessed: 2022-04-12.

[192] L. Petrica, T. Alonso, M. Kroes, N. Fraser, S. Cotofana, and M. Blott, "Memory-efficient dataflow inference for deep cnns on fpga," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 48–55.

[193] A. Henzinger, A. Noe, and C. Schulz, "Ilp-based local search for graph partitioning," *ACM J. Exp. Algorithmics*, vol. 25, jul 2020. [Online]. Available: https://doi.org/10.1145/3398634

[194] A. Hahn Pereira and V. Betz, "CAD and routing architecture for interposer-based multi-fpga systems," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 75–84.

[195] E. Nasiri, J. Shaikh, A. H. Pereira, and V. Betz, "Multiple dice working as one: CAD flows and routing architectures for silicon interposer FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 5, pp. 1821–1834, 2015.

[196] W.-H. Liu, M.-S. Chang, and T.-C. Wang, "Floorplanning and signal assignment for silicon interposer-based 3d ics," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[197] F. Mao, W. Zhang, B. Feng, B. He, and Y. Ma, "Modular placement for interposer based multi-FPGA systems," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*. IEEE, 2016, pp. 93–98.

[198] W.-S. Kuo, S.-H. Zhang, W.-K. Mak, R. Sun, and Y. K. Leow, "Pin Assignment Optimization for Multi-2.5D FPGA-Based Systems," in *Proceedings of the 2018 International Symposium on Physical Design*, ser. ISPD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 106–113. [Online]. Available: https://doi.org/10.1145/3177540.3178246

[199] N. Voss, P. Quintana, O. Mencer, W. Luk, and G. Gaydadjiev, "Memory mapping for multi-die FPGAs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 78–86.

[200] N. Tarafdar, G. Di Guglielmo, P. C. Harris, J. D. Krupa, V. Loncar, D. S. Rankin, N. Tran, Z. Wu, Q. Shen, and P. Chow, "Aigean: An open framework for machine learning on heterogeneous clusters," in *Sixth International Workshop on Heterogeneous High-performance Reconfigurable Computing)*. IEEE, 2020.

[201] W. Zhang, J. Zhang, M. Shen, G. Luo, and N. Xiao, "An efficient mapping approach to large-scale DNNs on multi-FPGA architectures," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1241–1244.

[202] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient CNN implementation on a deeply pipelined FPGA cluster," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016, pp. 326–331.

[203] C. Baskin, N. Liss, E. Zheltonozhskii, A. M. Bronstein, and A. Mendelson, "Streaming architecture for large-scale quantized neural networks on an fpga-based dataflow platform," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 162–169.

[204] W. Jiang, E. H.-M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, "Achieving super-linear speedup across multi-fpga for real-time dnn inference," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–23, 2019.

[205] J. Shan, M. T. Lazarescu, J. Cortadella, L. Lavagno, and M. R. Casu, "CNN-on-AWS: Efficient Allocation of Multikernel Applications on Multi-FPGA Platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 2, pp. 301–314, 2021.

[206] Z. Sun, K. Campbell, W. Zuo, K. Rupnow, S. Gurumani, F. Doucet, and D. Chen, "Designing high-quality hardware on a development effort budget: A study of the current state of high-level synthesis," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016, pp. 218–225.

[207] D. Singh and P. Yiannacouras, *OpenCL*. Springer International Publishing, 2016, pp. 97–114. [Online]. Available: https://doi.org/10.1007/978-3-319-26408-0_6

[208] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "Autodse: Enabling software programmers to design efficient fpga accelerators," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 27, no. 4, feb 2022. [Online]. Available: https://doi.org/10.1145/3494534

[209] J. Volder, "The cordic computing technique," in *Managing Requirements Knowledge, International Workshop on*, vol. 1. Los Alamitos, CA, USA: IEEE Computer Society, mar 1959, p. 257. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/AFIPS.1959.57

[210] H. Ochi, "Rtl design of parallel fft with block floating point arithmetic," in *2008 IEEE Conference on Soft Computing in Industrial Applications*, 2008, pp. 273–276.

[211] J.-M. Muller, "Elementary functions and approximate computing," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2136–2149, 2020.

[212] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1014–1029, 2021.

[213] J. Santiago da Silva, F.-R. Boyer, and J. P. Langlois, "Module-per-object: A human-driven methodology for c++-based high-level synthesis design," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 218–226.

[214] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[215] S. Sarkar, S. Dabral, P. K. Tiwari, and R. S. Mitra, "Lessons and experiences with high-level synthesis," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 34–45, 2009.

[216] Xilinx Inc, "Vitis hls hardware design methodology," Available online at: https://docs.xilinx.com/r/2021.1-English/ug1399-vitis-hls/Vitis-HLS-Hardware-Design-Methodology. Accessed: 2022-04-20.

[217] Microchip Technology Inc, "Legup: Optimization guide," available online at: https://download-soc.microsemi.com/FPGA/HLS-EAP/docs/legup-9.1-docs/optimizationguide.html#inferring-streaming-hardware-via-producer-consumer-pattern-with-pthreads. Accessed on: 2022-04-20.

[218] J. J. García Aranda, M. González Casquete, M. Cao Cueto, J. Navarro Salmerón, and F. González Vidal, "Logarithmical hopping encoding: a low computational complexity algorithm for image compression," *IET Image Processing*, vol. 9, no. 8, pp. 643–651, 2015.

[219] T. Alonso, M. Ruiz, A. López García-Arias, G. Sutter, and J. E. López de Vergara, "Submicrosecond latency video compression in a low-end fpga-based system-on-chip," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 355–3554.

[220] N. Dave, M. C. Ng, M. Pellauer, and Arvind, "A design flow based on modular refinement," in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, 2010, pp. 11–20.

[221] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Latency insensitive protocols," in *International Conference on Computer Aided Verification*. Springer, 1999, pp. 123–133.

[222] P. Mantovani, R. Margelli, D. Giri, and L. P. Carloni, "Hl5: A 32-bit risc-v processor designed with high-level synthesis," in *2020 IEEE Custom Integrated Circuits Conference (CICC)*, 2020, pp. 1–8.

[223] S. Rokicki, D. Pala, J. Paturel, and O. Sentieys, "What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications," in *ICCAD 2019 - 38th IEEE/ACM International Conference on Computer-Aided Design*. Westminster, CO, United States: IEEE, Nov. 2019, pp. 1–8. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02303453

[224] J. M. P. Cardoso and M. Weinhardt, *High-Level Synthesis*. Cham: Springer International Publishing, 2016, pp. 23–47. [Online]. Available: https://doi.org/10.1007/978-3-319-26408-0_2

[225] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[226] R. Venkatakrishnan, A. Misra, and V. Kindratenko, "High-level synthesis-based approach for accelerating scientific codes on fpgas," *Computing in Science Engineering*, vol. 22, no. 4, pp. 104–109, 2020.

[227] P. F. Silva, J. Bispo, and N. Paulino, "Fpgas as general-purpose accelerators for non-experts via hls: The graph analysis example," in *2021 International Conference on Field-Programmable Technology (ICFPT)*, 2021, pp. 1–4.

[228] A. Canis, J. Choi, B. Fort, B. Syrowik, R. L. Lian, Y. T. Chen, H. Hsiao, J. Goeders, S. Brown, and J. Anderson, *LegUp High-Level Synthesis*. Cham: Springer International Publishing, 2016, pp. 175–190. [Online]. Available: https://doi.org/10.1007/978-3-319-26408-0_10

[229] M. A. Özkan, O. Reiche, F. Hannig, and J. Teich, "Fpga-based accelerator design from a domain-specific language," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–9.

[230] Xilinx Inc., "Vivado High-Level Synthesis," available online at: https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis. Accessed: 2022-04-22.

[231] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Young, and Z. Zhang, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 129–132.

[232] A. Al-Zoubi and K. Tatas, “Rapid high-level fpga resource estimation for a novel heterogeneous platform scheduling scheme,” in *2020 11th International Conference on Information and Communication Systems (ICICS)*, 2020, pp. 378–381.

[233] Y. L. Aung, S.-K. Lam, and T. Srikanthan, “Rapid estimation of dsps utilization for efficient high-level synthesis,” in *2015 IEEE International Conference on Digital Signal Processing (DSP)*, 2015, pp. 1261–1265.