Comparativa del uso de HLLs en FPGA, GPU y Multicore para la aceleración de una aplicación de red IP

Víctor Moreno Martínez, Francisco J. Gómez-Arribas, Iván González Martínez,

Diego Sánchez-Román, Gustavo Sutter, Sergio López-Buedo

victor.moreno@uam.es, francisco.gomez@uam.es, ivan.gonzalez@uam.es

Escuela Politécnica Superior, Universidad Autónoma de Madrid.

Resumen

Afrontar la creación de aplicaciones a partir de lenguajes de alto nivel (*High Level Lenguajes - HLLs*) tiene la incuestionable ventaja de reducir el tiempo de desarrollo. Con ello, es viable una evaluación anticipada del prototipo para conocer cuanto antes si se alcanza el rendimiento especificado como objetivo. En este trabajo se han utilizando tres tecnologías de aceleración: multicore, coprocesador gráfico y coprocesador reconfigurable, que aprovechan el procesamiento paralelo, y se ha realizado una discusión crítica de su experiencia de uso a través de abstracciones de alto nivel.

Las soluciones basadas en el uso de coprocesadores no sólo han aportado un nivel de rendimiento superior a la aplicación que se deseaba acelerar, sino que con vista al sistema final en producción, estas alternativas son especialmente interesantes por dejar libres los *cores* de procesamiento para la realización de las operaciones de envío/recepción en la red de comunicaciones.

1. Introducción

El ámbito de aplicación de los lenguajes de alto nivel es cada vez más amplio. Así tradicionalmente, en aplicaciones para procesadores de propósito general programar en ensamblador es testimonial frente a la cantidad de desarrollos realizados en lenguaje C o similar.

Actualmente, en el diseño de sistemas digitales en *FPGAs* se está produciendo una situación que guarda cierta similitud. Se plantea un compromiso en términos de tiempo de desarrollo, frente a rendimiento, según se utilice la posibilidad de compilación *hardware* [1] desde lenguajes de alto nivel [2] como *Impulse-C* o *HandelC*, o bien la síntesis desde lenguajes de descripción de *Hardware* (*Hardware Description Languages - HDLs*) como *VHDL* o *Verilog*.

Extrapolando la filosofía subyacente en los escenarios anteriores, se puede considerar situaciones similares en aplicaciones desarrolladas con *GPGU* (*General Purpose Graphic Processing Unit*). Al utilizar como coprocesador una *GPU* se consigue acelerar aplicaciones que encajen con el modelo de ejecución *SPMD* (*Single-Program Multiple-Data*). La arquitectura interna de una GPU es un sistema *ManyCore* [2] cuyos recursos pueden aprovecharse utilizando el entorno de programación *CUDA* (*Compute Unified Device Architecture*) [3] en *GPUs* de *Nvidia*. Esta aproximación es similar al uso de *HLLs* ya que se describe la aplicación en lenguaje de propósito general, concretamente lenguaje C, complementado con un *API* que abstrae los detalles internos de la arquitectura.

Siguiendo un criterio similar, se puede considerar comparable la manera de programar que aprovecha el paralelismo de un sistema *multicore* con memoria compartida. Esta arquitectura está presente hoy en día en prácticamente todo sistema que incorpore un procesador de propósito general. Utilizando *OpenMP* [5-6] se consigue explotar el paralelismo *multicore* con modificaciones mínimas, que consisten en incluir directivas y *pragmas* en el código de la aplicación descrita en un lenguaje de alto nivel como *FORTRAN* o *C*.

Además de la consiguiente disminución del tiempo de llegada al mercado que se consigue con aproximaciones basadas en *HLLs*, se debe valorar muy positivamente el decremento en el tiempo de formación de un ingeniero, hasta alcanzar un

grado de dominio razonable que le permita producir soluciones operativas. Esto favorece también que se incremente el número de ingenieros dispuestos a desarrollar implementaciones *hardware* (*HW*) al acercarse al paradigma de desarrollo de aplicaciones tradicional en *software* (*SW*).

En este trabajo se compara la utilización de lenguajes de alto nivel para acelerar el rendimiento de una aplicación en tres tecnologías muy diferentes: *GPUs*, *Mulicore* y *FPGA*. Los resultados muestran que aunque el rendimiento alcanzado en los tres casos puede ser posiblemente mejorado si se utilizan una metodología de diseño de más bajo nivel, esto no suele ser necesario porque ya se alcanza el rendimiento requerido.

2. Aplicación de red *TCP/IP*

En el marco de un proyecto industrial se está desarrollando un discriminador de tráfico para su incorporación en una red de altas prestaciones. Como se representa en el esquema de la figura 1, en el caso que nos ocupa, el escenario de comunicaciones de red acumula a la entrada de un único equipo, tráfico de diferentes fuentes, que además de recibirse simultáneamente por varias interfaces de red ópticas llega también con tasas muy elevadas. Adicionalmente se detecta la problemática de que parte del tráfico llega replicado debido a una configuración que tiene la ventaja de garantizar la redundancia de la arquitectura de red.

En este escenario, la agrupación de gran cantidad de tráfico, junto con la presencia de duplicados y triplicados de tramas hace que se incremente la tasa de línea que debe ser capaz de procesar el resto de equipos de supervisión. Por esta razón, se necesita eliminar los paquetes duplicados, tras lo cual se debe realizar una discriminación del tráfico sin duplicados entre varios interfaces de salida. Así se logrará balancear la tasa de tráfico que sale por cada interfaz. El objetivo final del discriminador es ajustar la tasa de paquetes de cada una de las salidas para que sea viable realizar un posterior análisis *DPI* (*Deep Packet Inspection*) en equipos externos con necesidades menores en términos de capacidad de procesamiento.

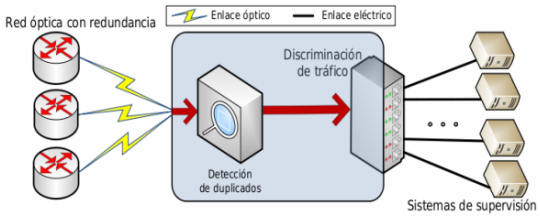

Figura 1. Esquema de interconexión del sistema discriminador en la red.

En un primer estudio realizado sobre la red del cliente, se obtuvieron estadísticas sobre los duplicados que aparecían en la red. El análisis de la distribución de duplicados mostraba que las ventanas de tiempo en que se detectaban los duplicados eran inferiores al minuto. Teniendo en cuenta esta característica, se consideró la cantidad de paquetes consecutivos en las que puede aparecer un duplicado como la menor cantidad que detectaba el *100%* de los duplicados considerando la ventana temporal. Este valor resultó ser de *600* paquetes en la red analizada y se definió como valor por defecto para el parámetro de configuración que define el tamaño de ventana de detección de duplicados

Asimismo, con vistas a mejorar la eficiencia de la solución, fue necesaria la obtención de la cantidad de bytes de cada paquete que se utilizarían para la comparación de duplicidad. Se comprobó que con los primeros 54 bytes de cada paquete (*14* cabecera *Ethernet* + *20* cabecera *IP* + *20* cabecera *TCP*), se cumplía la condición deseada. No obstante, para evitar problemas de alineamiento de memoria, se decidió realizar una implementación que tiene en cuenta los primeros 64 bytes de cada paquete.

Con los parámetros extraídos en el análisis del tráfico se propone evaluar el rendimiento de un algoritmo de búsqueda de duplicados basado en comparaciones realizadas contra todos los paquetes que caben en la ventana. Esta aproximación es fácil de implementar desde *HLLs*.

El sistema completo constará de un módulo de eliminación de duplicados y un módulo de discriminación. Como trabajo previo se van a evaluar el rendimiento alcanzado en diferentes tecnologías en las que se implementa el algoritmo de la aplicación desde *HLLs*. Se pretende elegir la tecnología que mejor se adapte, logrando un compromiso entre tiempo de desarrollo, precio y

rendimiento. El punto de partida es una arquitectura similar a la de un servidor de computación *multicore* con memoria compartida en el que se pueda incorporar según las necesidades capacidad de coprocesamiento por GPU o con FPGA. Como paso previo se van a comparar las diferentes tecnologías sobre una misma plataforma de prototipado que se describe en la siguiente sección. La arquitectura del sistema final se decidirá en función de los resultados obtenidos en la comparación considerando su escalabilidad y el rendimiento máximo que se puede extrapolar en cada caso.

3. Plataforma de prototipado

El sistema en el que se ha llevado a cabo el estudio es un servidor con una placa base *dual Xeon*, a la cual están conectados un procesador *Intel Quad-Core L5408* y un acelerador *In-Socket* basado en *FPGA*: el *XD2000i* de *XtremeData* [7]. El acelerador *XD2000i*, conectado al segundo *socket* de procesador de la placa, utiliza la infraestructura de *CPU* existente en la placa base para crear un entorno de coprocesamiento completamente funcional.

La conexión entre el procesador Intel y el acelerador re realiza a través del *Front-Side Bus* (*FSB*), pudiendo así sacar partido del alto ancho de banda y la baja latencia de este método de interconexión.

El acelerador *In-Socket XD2000i* se compone de tres *FPGAs Altera Stratix III* [8]. Una de estas *FPGAs* ejerce la labor de *bridge* hacia el *FSB* (*Stratix III SL150*), mientras que las otras dos (*Stratix III SE260*, con *255K* puertas lógicas*, 768* multiplicadores embebidos y *15 Mb* de memoria interna) están disponibles para alojar la lógica implementada por el usuario. Estas dos *FPGAs* están a su vez directamente conectadas a través de dos canales unidireccionales de *64*-bits funcionando a una velocidad de *400 MT*/s (millones de transacciones por segundo). Además, el módulo *XD2000i* dispone de dos bancos de memoria *QDRII+ SRAM* de *8 Mb* cada uno, cada uno de ellos conectado a una de las *FPGAs* de usuario. Toda la placa *XD2000i* funciona a una frecuencia de *100 MHz*.

En la misma máquina, hay además conectada a través del bus *PCI-Express* una *GPU Tesla C1060* de *Nvidia* [9]. Este coprocesador gráfico se conecta a través de un bus *PCIe x16*, y aporta al sistema *240 cores* de procesamiento, que pueden hacer uso de una memoria interna compartida por todos ellos de *4GB*.

4. Implementaciones desde lenguajes de alto nivel

4.1. *OpenMP* en *multicore* (*MPC*)

La solución al problema planteada desde el punto de vista puramente *MPC* (*Multi-Processor Computer*) es la más sencilla posible, basándose en el paradigma de programación de memoria compartida, concretamente en el uso de *OpenMP*. La aplicación consta de un bucle que recorre todos los paquetes que se desean analizar, y que para cada paquete se recorre los $Tam_{VENTANA}$ paquetes siguientes en busca del primer paquete que cumpla la condición de duplicado. Para ejecutar el programa aprovechando el paralelismo existente en la arquitectura destino, simplemente se ha colocado un *pragma OpenMP* delante del bucle más externo.

La compilación de esta implementación se ha realizado el compilador de libre distribución *gcc*, añadiéndole la opción *–O3* con vistas a obtener los mejores resultados de rendimiento ofrecido por esta herramienta de compilación.

4.2. *CUDA* en *manycore* (*GPU*)

Tomando la solución *MPC* como punto de partida, se ha creado una solución basada en tecnología *GPGPU*. Mientras que la solución *MPC* puede parece demasiado burda, encaja en la filosofía de ejecución de las *GPGPU*: tenemos muchos paquetes, de modo que cada uno de los *cores* de la *GPU* se encargará de realizar el procesamiento necesario para uno de los paquetes.

El tiempo requerido para desarrollar la solución *GPU* es también muy reducido utilizando el paradigma de programación *CUDA* que ofrece el fabricante *NVidia* para sus dispositivos.

4.3. *Impulse-C* en *FPGA*

Para el desarrollo del sistema basado en *FPGA*, se ha elegido utilizar una metodología basado en el empleo de lenguajes de alto nivel, con vistas a agilizar el tiempo transcurrido hasta la obtención de un producto funcional. Concretamente, se ha optado por el lenguaje *Impulse-C* [10-11]. La elección de este *HLL* para

la realización del desarrollo se debe por un lado a la existencia de un paquete de soporte de la plataforma de la que se disponía (*Plattform Support Package - PSP*) y por otro lado a resultados previos obtenidos en otros trabajos con resultados muy satisfactorios [12]. Este lenguaje tiene además un paradigma de programación basado en *streams* de datos, que facilita en gran medida su utilización para propósitos de coprocesamiento.

La primera aproximación es una comparación "por fuerza bruta", análogo con los desarrollos llevados a cabo en *MPC* y *GPGU*. De este modo, a la *FPGA* le irían llegando los paquetes en serie, y para cada paquete realizaría la comparación de duplicidad con los $Tam_{VENTANA}$ anteriores. Una vez terminadas estas comparaciones, se almacena el paquete en la memoria interna de la *FPGA* y se retorna el resultado (duplicado/no duplicado) a través de una interfaz de *streaming*.

Debido a que Impulse-C permite explotar la característica de las memorias internas de la *FPGA* de permitir el acceso a dos datos en cada ciclo de reloj, tendríamos que para obtener el resultado de cada paquete tardaríamos $Tam_{VENTANA}/2$ ciclos de reloj. En el caso nuestra aplicación ($Tam_{VENTANA} = 600$) esto nos limita (sin tener en cuenta otro tipo de *overheads*) a un *throughput* de:

$$\left(\frac{Tam_{VENTANA}}{2} \cdot \frac{1}{100Mhz}\right)^{-1} = (300 \cdot 10ns)^{-1} \approx 3{,}3 \cdot 10^5 \; pps$$

Con vistas a obtener una solución cuyo rendimiento no sea tan dependiente del tamaño de ventana, se propone una arquitectura un poco más elaborada, ilustrada en la figura 2. En esta aproximación, se basa en la utilización de un *hash* que resuma el contenido del paquete. De este modo, dos paquetes con distintos valores de *hash* no pueden ser duplicados. Basándonos en esta idea, creamos una tabla indexada con los valores del *hash* en la que se almacena la posición del último paquete que llegó con ese mismo valor. Puesto que sólo se desea comparar con los últimos $Tam_{VENTANA}$ paquetes, a cada paquete entrante se le asigna un *timestamp* que será almacenado en la misma tabla en que se guarda su posición. Los pasos del algoritmo seguidos son los siguientes:

1.) Para cada paquete entrante se calcula un valor de *hash.*

2.) Si el *timestamp* del último paquete con ese valor de *hash* es de hace más de $Tam_{VENTANA}$ paquetes (o todavía no ha llegado un paquete con ese valor de *hash*), entonces el paquete no es un duplicado.

3.) Si no se cumplen las condiciones de la etapa anterior, se procede a comparar el paquete entrante con el paquete almacenado en la memoria que tenía el mismo valor de hash. Si los paquetes son iguales, el paquete entrante es entonces un duplicado.

Llegado este punto, no podemos afirmar que si el paquete es diferente al que residía en la memoria, el paquete entrante no es un duplicado: puede ocurrir que hayan llegado otros paquetes con el mismo valor de *hash* que el entrante antes que el que nos apuntaba la tabla. Por tanto, hemos de tener una segunda tabla (que nos resuelve las "dobles colisiones") que nos indica cuando fue el instante de tiempo en que llegó el último paquete con un valor de *hash* igual al paquete entrante, pero diferente del que se tiene almacenado en la tabla. Una vez se cuenta con esta tabla, el algoritmo puede continuar:

4.) Se comprueba en la segunda tabla si hace menos de $Tam_{VENTANA}$ paquetes llegó un paquete con el mismo valor *hash* que el entrante, pero diferente al que apuntaba la primera tabla *hash*. Si no se cumple esta condición, entonces el paquete es no es un duplicado.

5.) Si se cumple la condición de la etapa anterior, se ha de comparar el paquete con todos los paquetes residentes en memoria en busca de duplicados.

La idea de esta aproximación es por lo tanto reducir la cantidad de veces que se ha de recorrer toda la memoria en busca de los posibles duplicados.

Puede realizarse un análisis teórico para comprobar si compensa o no una solución de este tipo. Si aceptamos que la función de *hash* genera valores de manera uniforme, y suponemos independencia entre todos los paquetes (o más bien entre sus valores de *hash*), tenemos que la probabilidad de que se produzca una colisión (que produzcan el mismo valor de *hash*) entre dos paquetes es:

$$p = P(colisión) = \frac{1}{Tam_{TABLA\_HASH}} \quad (1)$$

Queremos calcular ahora la probabilidad de que se produzca una colisión entre el paquete entrante con cualquiera de los $Tam_{VENTANA}$ anteriores. Para obtener este valor, se ha de utilizar la fórmula del principio de inclusión-exclusión:

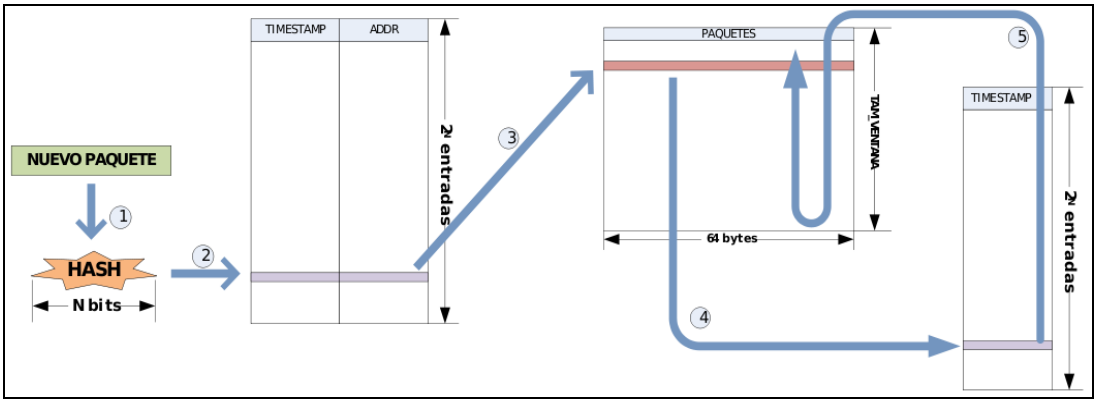

Figura 2. Diagrama de funcionamiento de la solución basada en *HASH*

$$p_{anteriores} = P\left(\bigcup_{i=1}^{Tam_{VENTANA}} col_i\right) =$$

$$= \sum_{i=1}^{Tam_{VENTANA}} P(col_i) - \sum_{j<i=1}^{Tam_{VENTANA}} P(col_i \cap col_j) + \ldots +$$

$$+ (-1)^{k-1} \sum_{j_k<\ldots<j_1=1}^{Tam_{VENTANA}} P(col_{j_1} \cap \ldots \cap col_{j_k}) + \ldots \quad (2)$$

Donde col_k es el suceso que indica que habido una colisión entre el paquete entrante y el paquete *k*. En nuestro caso esta expresión puede reducirse a:

$$P_{anteriores} = \sum_{k=1}^{Tam_{VENTANA}} (-1)^{k-1} \binom{Tam_{VENTANA}}{k} p^k \quad (3)$$

Debido a que la probabilidad de que el paquete entrante colisione con *j* paquetes anteriores es el producto de las probabilidades de que el paquete entrante colisione con cada uno de ellos por separado (por la propiedad de independencia), que es precisamente p^j. Los coeficientes binomiales aparecen debido a la cantidad de posibles elecciones de listas de *k* paquetes entre los $Tam_{VENTANA}$ anteriores (combinaciones sin repetición de $Tam_{VENTANA}$ elementos tomados de k en k). Si evaluamos esta expresión para los valores $Tam_{VENTANA}=600$ y $Tam_{TABLA\_HASH}=1024$ (*10* bits) nos da que la probabilidad de una colisión con los paquetes anteriores es de *0,444*. La probabilidad de que haya que recorrer la memoria entera es la probabilidad de que haya una doble colisión, que es por lo tanto, por independencia de sucesos, el cuadrado de lo que acabamos de definir como $P_{anteriores}$. Podemos ahora calcular una estimación del tiempo que se tardará en procesar cada paquete:

$$T = p_{anteriores}^2 \cdot \frac{Tam_{VENTANA}}{2} + t_{decision} \quad (4)$$

Donde denominamos a $t_{decision}$ como el tiempo por el que todo paquete debe pasar (las etapas 1-4 en la descripción del algoritmo). Se ha comprobado a través de una simulación comportamental que el valor de este parámetro $t_{decision}$ es de *40* ciclos de reloj (*=400ns*). Si sustituimos los valores de las variables de nuestro problema en la ecuación 4, podemos obtener una estimación conservadora de la capacidad procesamiento de esta aproximación de $2{,}02 \cdot 10^6$ paquetes por segundo. Puede apreciarse una mejora de casi un orden de magnitud en los rendimientos teóricos esperados por estas dos implementaciones, por lo que parece razonable invertir esfuerzo en la implementación de esta segunda solución basada en *Impulse-C*.

5. Resultados

5.1. Implementaciones *MPC* y GPU

En la figura 3 y la tabla 1 se presenta una comparativa del rendimiento obtenido utilizando las soluciones basadas en *MPC* y *GPU*, evaluado en términos de la cantidad de paquetes por segundo que son capaces de procesar. Se ha considerado para la realización de esta gráfica como caso peor aquel en que ningún paquete es un

duplicado, de modo que todos los paquetes deben compararse contra el resto, y como caso mejor aquel en que todos los paquetes son duplicados y encuentran un paquete similar a ellos en la primera comparación realizada.

Puede apreciarse una diferencia de rendimiento en el caso mejor a favor de la solución *multicore*. Esta diferencia es debida al *overhead* que supone realizar el envío de los datos y la recogida de los resultados pertinentes hacia/desde la *GPU* a través del bus *PCIe*.

	Caso mejor (pps)	Caso real (pps)	Caso peor (pps)
Open MP	$2{,}9 \cdot 10^7$	$3 \cdot 10^5$	$2{,}0 \cdot 10^5$
GPU	$1{,}4 \cdot 10^7$	$2{,}6 \cdot 10^6$	$1{,}9 \cdot 10^6$

Tabla 1. Throughput máximo (paquetes por segundo) MPC vs GPGPU

5.2. Implementación en *FPGA*

Se han observado varias peculiaridades en el comportamiento de la solución basada en *Impulse-C*. En la Figura 4, que corresponde a medidas realizadas sobre tráfico real, puede apreciarse que el rendimiento obtenido en función del número de paquetes enviados en cada transacción experimenta tendencias crecientes que caen repentinamente.

Este comportamiento se produce debido al *API* que el fabricante de la placa de aceleración que se ha utilizado [1] provee para su uso. Este *API* cuenta con una función encargada de enviar los datos sobre los que se quiere operar al coprocesador y esperar a que se generen los resultados.

Se ha observado que la espera hasta que se generan los resultados no se produce de forma asíncrona, pese a ofrecer al usuario esa impresión: el proceso se suspende hasta que el *driver* le comunica que los datos están disponibles enviándole una señal. Sin embargo, en base al comportamiento observado en la figura 5 se ha comprobado que la implementación del *driver* comprueba el estado de disponibilidad de los datos de salida cada milisegundo, obteniendo de éste modo tiempos de cómputo múltiplos de un milisegundo. Nos encontramos de este modo con que en términos de tiempo, nos es indiferente enviar al coprocesador *15.000* ó *16.000* paquetes en una misma transacción.

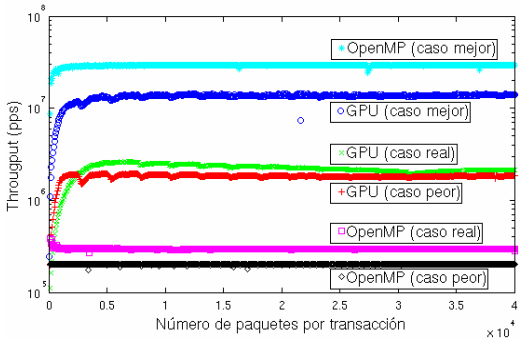

Figura 3. Throughput *MPC* vs *GPGPU*

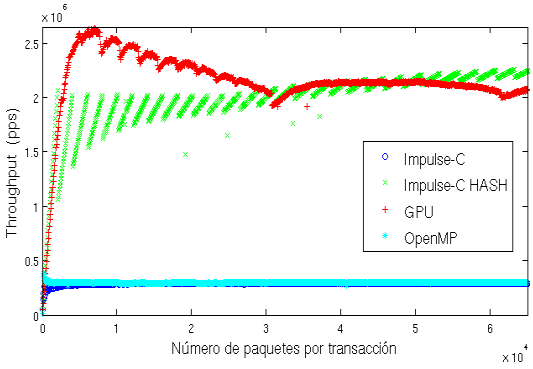

Figura 4. Throughput obtenido para tráfico real

Al utilizar más tarde esta solución en el sistema en producción se ha tenido en cuenta el efecto de estas discontinuidades para seleccionar adecuadamente la cantidad de paquetes enviados por transacción.

Tras realizar un análisis sobre cómo afecta el tamaño de la ventana de paquetes que se tienen en cuenta para la caracterización de duplicados al rendimiento del sistema. Lo más evidente de deducir sería que según se aumenta el tamaño de esta ventana, peor será el rendimiento obtenido, puesto que la cantidad de comparaciones a realizar es mayor. En la figura 6 puede observarse que esto no es necesariamente cierto.

En esta figura pueden apreciarse dos comportamientos dignos de mención:

1.) El rendimiento mejora en gran medida, llegando a un 4x, para tamaños de ventana potencias de 2. Esto es debido a que para almacenar los paquetes en la memoria del chip de cara a futuras

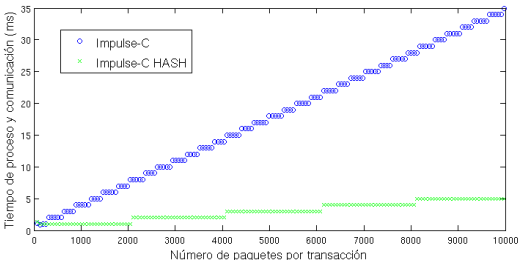

Figura 5. Efecto del *driver* del fabricante en el tiempo de cómputo del acelerador FPGA

comparaciones, se utiliza un contador que se incrementa módulo $Tam_{VENTANA}$. La operación *"%"*, comúnmente utilizadas en programación de alto nivel, tiene en este caso efectos colaterales. Obviamente, cuando el módulo se realiza con 2^k, simplemente hay que quedarse con los k bits menos significativos del contador. En el resto de casos, Impulse instancia un componente de su librería de gran latencia. Realizando un sencillo cambio de la sentencia

```
contador=(contador+1)%TamVENTANA;
```

por un incremento seguido de una sentencia condicional:

```
contador++;
if(contador==TamVENTANA)
  contador=0;
```

se traduce en un rendimiento considerablemente superior al de la *GPU* con nuestra *FPGA* ($7{,}3 \cdot 10^6$ paquetes por segundo con la *FPGA* frente a los $2{,}6 \cdot 10^6$ de la *GPU* manteniendo un tamaño de ventana igual a *600*).

2.) Separando los casos en que el tamaño de ventana es potencia de dos y los que no, al aumentar el tamaño de la ventana de comparación el rendimiento no se ve prácticamente afectado. Esto es debido a que gracias a la utilización de los valores de *hash*, la cantidad de veces que se ha de recorrer la memoria en busca de los paquetes es muy reducida.

6. Discusión del rendimiento obtenido sobre tráfico real

La Tabla 2 resume los resultados de rendimiento representados en la figura 4, comparando en términos de cantidad de paquetes por segundo que cada solución es capaz de procesar con tráfico real.

Puede observarse a partir de estos resultados que las soluciones basadas en *GPU* y en *FPGA*, aparecen como las opciones que ofrecen un mayor rendimiento en términos de capacidad de proceso del sistema. Ambas soluciones poseen como atractivo adicional la propiedad de dejar libres los *cores* de la máquina para poder encargarse de la transferencia de los datos desde y hacia la red de comunicaciones. Se ha comprobado de forma empírica que este proceso de envío/recepción de los paquetes por las diversas interfaces de red es el cuello de botella que limita el rendimiento del sistema, de modo que la utilización de coprocesadores adquiere un interés mucho mayor que las basadas en *MPC*.

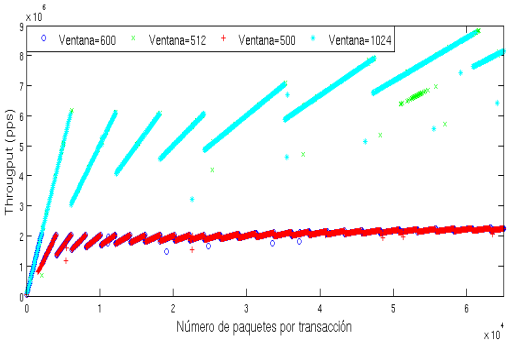

Figura 6. Efecto del tamaño de la ventana en el rendimiento del sistema

	Open MP	GPU	Impulse-C	Impulse-C HASH
Throughput (pps)	$3 \cdot 10^5$	$2{,}6 \cdot 10^6$	$2{,}9 \cdot 10^5$	$2{,}3 \cdot 10^6$

Tabla 2. Throughput máximo de las cuatro implementaciones llevadas a cabo con $Tam_{VENTANA}=600$

De cara a realizar una comparación justa podría implementarse una solución basada en una tabla *hash* también en *GPU* ó en *MPC*. Sin embargo, el paralelismo *SIMD* inherente en estas abstracciones requeriría de mecanismos de sincronización que complicarían en gran medida la implementación y posiblemente degradarían el rendimiento. Para la aplicación de red considerada, el procesamiento en *stream* de la *FPGA*, en el que los paquetes se procesan de uno en uno, favorece la aplicación de soluciones basadas en *hash* para esta tecnología.

7. Conclusiones y trabajo futuro

Se ha comprobado la utilidad de varias aproximaciones de programación de alto nivel para diferentes plataformas hardware. Este tipo de

enfoque ha servido para agilizar el tiempo de desarrollo de un prototipo funcional respecto a otras aproximaciones con un nivel de abstracción menor: *threads POSIX* para la programación en *MPC*, *NVAPI* (un *API* que provee *Nvidia* para comunicarse directamente con el driver de la tarjeta) en *GPU*, y lenguajes *HDL* para *FPGA*.

La curva de aprendizaje del lenguaje *Impulse-C* es la más lenta de las tres. No obstante, es una curva considerablemente más rápida que su equivalente en lenguajes de descripción hardware, *HDL*s. La utilización de aproximaciones de alto nivel para la utilización de *FPGAs* hace accesible a un público mucho más amplio que los *HDLs* la posibilidad de explotación de la capacidad de cómputo de las mismas.

De los resultados se concluye que las soluciones basadas en el uso de coprocesadores han aportado un nivel de rendimiento superior a la aplicación que se deseaba acelerar. De cara al sistema final en producción, estas alternativas son especialmente interesantes por dejar libres los *cores* de procesamiento para la realización de las operaciones de entrada/salida de/hacia la red de comunicaciones.

Para la versión en producción, se ha utilizado la solución basada en *GPU*. En esta misma versión, se ha procedido a implementar la parte de discriminación del tráfico saliente en la propia *GPU*, lo que ha supuesto una significativa degradación en la capacidad de cómputo del sistema. Esta discriminación del tráfico se hace en base a unas reglas similares a los de los mecanismos de filtrado de red tradicionales (basadas en rango de direcciones IP, rango de puertos, protocolos,…). Este tipo de operaciones degradan en gran medida el rendimiento de la *GPU* debido a que provocan que cada uno de los hilos de ejecución vaya por una rama de ejecución diferente.

Por esta razón, como línea de trabajo futuro se propone realizar una versión basada en *FPGA*, en la que la parte de decisión de la interfaz de salida de cada paquete pueda realizarse de forma muy eficiente.

8. Referencias

[1] J.M.P. Cardoso; P.C. Diniz; M. Weinhardt; "Compiling for reconfigurable computing: A survey". ACM Computing Surveys (CSUR), Vol.42 nº.4, páginas1-65, June 2010.

[2] El-Araby E.; Nosum, P.; El-Ghazawi, T.; "A Framework for Evaluating High-Level Design Methodologies for High-Performance Reconfigurable Computers", IEEE Transactions on Parallel and Distributed Systems, pag. 33-45, Jan. 2011.

[3] T.R. Halfhill; "Parallel Processing with Cuda Nvidia's High-Performance Computing Platform Uses Massive Multithreading", Microproc. Report, Vol. 22, No. 1, Enero 2008.

[4] Song Jun Park; Dale R. Shires; Brian J. Henz."Coprocessor Computing with FPGA and GPU". Proceedings Of The Hpcmp Users Group Conference 2008, páginas 366-370.

[5] L. Dagum; R. Menon; "OpenMP: an industry standard API for shared-memory programming". IEEE Computational Science & Engineering, Jan-Mar 1998.

[6] Alonso, P.; Cortina, R.; Martinez-Zaldvar, F.; Ranilla, J.; "Neville elimination on multi and many-core systems: OpenMP, MPI and CUDA", The Journal of Supercomputing, pag. 1-11, 2009.

[7] XtremeData, Inc., "XtremeData XD2000i in-socket_accelerator", http://www.xtremedata.com/products/accelerators/in-socket-accelerator/xd2000i.

[8] Altera Corporation, "Altera FPGA devices", http://www.altera.com.

[9] Nvidia Tesla Computing Processor. http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C1060_US_Jan10_lores_r1.pdf.

[10] Impulse Accelerated Technologies, "Impulse-C", http://www.impulseaccelerated.com.

[11] D. Pellerin; S. Thibault; "Practical fpga programming in C". Prentice Hall Press. ISBN:131543180 .

[12] D. Sanchez-Roman; G. Sutter; S. López-Buedo; I. González; F.J. Gomez-Arribas; J. Aracil; "An Euler solver accelerator in FPGA for computational fluid dynamics applications". VII Southern Programmable Logic Conference, SPL2011.