

## Research Article

# A Convolve-And-MERge Approach for Exact Computations on High-Performance Reconfigurable Computers

Esam El-Araby,<sup>1</sup> Ivan Gonzalez,<sup>2</sup> Sergio Lopez-Buedo,<sup>2</sup> and Tarek El-Ghazawi<sup>3</sup>

<sup>1</sup> Department of Electrical Engineering and Computer Science, The Catholic University of America, Washington, DC 20064, USA

<sup>2</sup> Departments of Computer Engineering at Escuela Politecnica Superior of Universidad Autonoma de Madrid, 28049 Madrid, Spain

<sup>3</sup> Department of Electrical and Computer Engineering, The George Washington University, Washington, DC 20052, USA

Correspondence should be addressed to Esam El-Araby, aly@cua.edu

Received 31 October 2011; Revised 1 February 2012; Accepted 13 February 2012

Academic Editor: Thomas Steinke

Copyright © 2012 Esam El-Araby et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

This work presents an approach for accelerating arbitrary-precision arithmetic on high-performance reconfigurable computers (HPRCs). Although faster and smaller, fixed-precision arithmetic has inherent rounding and overflow problems that can cause errors in scientific or engineering applications. This recurring phenomenon is usually referred to as numerical nonrobustness. Therefore, there is an increasing interest in the paradigm of exact computation, based on arbitrary-precision arithmetic. There are a number of libraries and/or languages supporting this paradigm, for example, the GNU multiprecision (GMP) library. However, the performance of computations is significantly reduced in comparison to that of fixed-precision arithmetic. In order to reduce this performance gap, this paper investigates the acceleration of arbitrary-precision arithmetic on HPRCs. A Convolve-And-MERge approach is proposed, that implements virtual convolution schedules derived from the formal representation of the arbitrary-precision multiplication problem. Additionally, dynamic (nonlinear) pipeline techniques are also exploited in order to achieve speedups ranging from 5x (addition) to 9x (multiplication), while keeping resource usage of the reconfigurable device low, ranging from 11% to 19%.

## 1. Introduction

Present-day computers built around fixed-precision components perform integer and/or floating point arithmetic operations using fixed-width operands, typically 32 and/or 64 bits wide. However, some applications require larger precision arithmetic. For example, operands in public-key cryptography algorithms are typically thousands of bits long. Arbitrary-precision arithmetic is also important for scientific and engineering computations where the roundoff errors arising from fixed-precision arithmetic cause convergence and stability problems. Although many applications can tolerate fixed-precision problems, there is a significant number of other applications, such as finance and banking, in which numerical overflow is intolerable. This recurring phenomenon is usually referred to as numerical nonrobustness [1]. In response to this problem, exact computation, based on exact/arbitrary-precision arithmetic, was first introduced in 1995 by Yap and Dube [2] as an emerging numerical

computation paradigm. In arbitrary-precision arithmetic, also known as *bignum* arithmetic, the size of operands is only limited by the available memory of the host system [3, 4].

Among other fields, arbitrary-precision arithmetic is used, for example, in computational metrology and coordinate measuring machines (CMMs), computation of fundamental mathematical constants such as  $\pi$  to millions of digits, rendering fractal images, computational geometry, geometric editing and modeling, and constraint logic programming (CLP) languages [1–5].

In the earlier days of computers, there were some machines that supported arbitrary-precision arithmetic in hardware. Two examples of these machines were the IBM 1401 [6] and the Honeywell 200 Liberator [7] series. Nowadays, arbitrary-precision arithmetic is mostly implemented in software, perhaps embedded into a computer compiler. Over the last decade, a number of *bignum* software packages have been developed. These include the GNU multiprecision (GMP) library, CLN, LEDA, Java.math, BigFloat, BigDigits,

and Crypto++ [4, 5]. In addition, there exist stand-alone application software/languages such as PARI/GP, Mathematica, Maple, Macsyma, dc programming language, and REXX programming language [4].

Arbitrary-precision numbers are often stored as large-variable-length arrays of digits in some base related to the system word-length. Because of this, arithmetic performance is slower compared to fixed-precision arithmetic which is closely related to the size of the processor internal registers [2]. There have been some attempts for hardware implementations. However, those attempts usually amounted to specialized hardware for small-size discrete multiprecision and/or to large-size fixed-precision [8–12] integer arithmetic rather than to real arbitrary-precision arithmetic.

High-performance reconfigurable computers (HPRCs) have shown remarkable results in comparison to conventional processors in those problems requiring custom designs because of the mismatch with operand widths and/or operations of conventional ALUs. For example, speedups of up to 28,514 have been reported for cryptography applications [13, 14], up to 8,723 for bioinformatics sequence matching [13], and up to 32 for remote sensing and image processing [15, 16]. Therefore, arbitrary-precision arithmetic seemed to be a good candidate for acceleration on reconfigurable computers.

This work explores the use of HPRCs for arbitrary-precision arithmetic. We propose a hardware architecture that is able to implement addition, subtraction and multiplication, as well as convolution, on arbitrary-length operands up to 128 ExibiByte. The architecture is based on virtual convolution scheduling. It has been validated on a classic HPRC machine, the SRC-6 [17] from SRC Computers, showing speedups ranging from 2 to 9 in comparison to the portable version of the GMP library. This speedup is in part attained due to the dynamic (nonlinear) pipelining techniques that are used to eliminate the effects of deeply pipelined reduction operators.

The paper is organized as follows. Section 2 presents a short overview of HPRC machines. The problem is formulated in Section 3. Section 4 describes the proposed approach and architecture augmented with a numerical example for illustrating the details of the proposed approach. Section 5 shows the experimental work. Implementation details are also given in Section 5, as well as performance comparison to the SW version of GMP. Finally, Section 6 presents the conclusions and future directions.

## 2. High Performance Reconfigurable Computing

In the recent years, the concept of high-performance reconfigurable computing has emerged as a promising alternative to conventional processing in order to enhance the performance of computers. The idea is to accelerate a parallel computer with reconfigurable devices such as FPGAs where a custom hardware implementation of the critical sections of the code is performed in the reconfigurable device. Although the clock frequency of the FPGA is typically one order of

magnitude less than the one of high-end microprocessors, significant speedups are obtained due to the increased parallelism of hardware. This performance is especially important for those algorithms not matching the architecture of conventional microprocessors, because of either the operand lengths (e.g., bioinformatics) or the operations performed (e.g., cryptography). Moreover, the power consumption is reduced in comparison to conventional platforms, and the use of reconfigurable devices brings flexibility closer to that of SW, as opposed to other HW-accelerated solutions such as ASICs. In other words, the goal of HPRC machines is to achieve the synergy between the low-level parallelism of hardware with the system-level parallelism of high-performance computing (HPC) machines.

In general, HPRCs can be classified as either nonuniform node uniform systems (NNUSs) or uniform node nonuniform systems (UNNSs) [13]. NNUSs consist of only one type of nodes. Nodes are heterogeneous containing both FPGAs and microprocessors. FPGAs are connected directly to the microprocessors inside the node. On the other hand, UNNS nodes are homogeneous containing either FPGAs or microprocessors which are linked via an interconnection network. The platform used in this paper, SRC-6 [17], belongs to the second category.

SRC-6 platform consists of one or more general-purpose microprocessor subsystems, one or more MAP reconfigurable processor subsystems, and global common memory (GCM) nodes of shared memory space [17]. These subsystems are interconnected through a Hi-Bar Switch communication layer; see Figure 1. Multiple tiers of the Hi-Bar Switch can be used to create large-node count scalable systems. Each microprocessor board is based on 2.8 GHz Intel Xeon microprocessors. Microprocessors boards are connected to the MAP boards through the SNAP interconnect. The SNAP card plugs into the memory DIMM slot on the microprocessor motherboard to provide higher data transfer rates between the boards than the less efficient but common PCI solution. The peak transfer rate between a microprocessor board and the MAP board is 1600 MB/sec. Hardware architecture of the SRC-6 MAP processor is shown in Figure 1. The MAP Series C board is composed of one control FPGA and two user FPGAs, all Xilinx Virtex II-6000-4. Additionally, each MAP unit contains six interleaved banks of on-board memory (OBM) with a total capacity of 24 MB. The maximum aggregate data transfer rate among all FPGAs and on-board memory is 4800 MB/s. The user FPGAs are configured in such a way that one is in the master mode and the other is in the slave mode. The two FPGAs of a MAP are directly connected using a bridge port. Furthermore, MAP processors can be chained together using a chain port to create an array of FPGAs.

## 3. Problem Formulation

Exact arithmetic uses the four basic arithmetic operations (+, −, ×, ÷) over the rational field  $\mathbf{Q}$  to support exact computations [1–5]. Therefore, the problem of exact computation is reduced to implementing these four basic arithmetic operations with arbitrary-sized operands.

TABLE 1: Computational complexity of arithmetic operations [18].

Operation	Input	Output	Algorithm	Complexity
Addition	Two $n$ -digit numbers	One $(n + 1)$ -digit number	Basecase/Schoolbook	$O(n)$
Subtraction	Two $n$ -digit numbers	One $(n + 1)$ -digit number	Basecase/Schoolbook	$O(n)$
Multiplication	Two $n$ -digit numbers	One $2n$ -digit number	Basecase/Schoolbook	$O(n^2)$
			Karatsuba	$O(n^{1.585})$
			3-way Toom-Cook	$O(n^{1.465})$
			$k$ -way Toom-Cook	$O(n^{1+\epsilon}), \epsilon > 0$
			Mixed-level Toom-Cook	$O(n(\log n)2^{\sqrt{(2\log n)}})$
			Schönhage-Strassen	$O(n(\log n)(\log \log n))$
Note: The complexity of multiplication will be referred to as $M(n)$ in the following				
Division	Two $n$ -digit numbers	One $n$ -digit number	Basecase/Schoolbook	$O(n^2)$
			Newton's method	$O(M(n))$
			Goldschmidt	$O(M(n))$
Square root	One $n$ -digit number	One $n$ -digit number	Newton's method	$O(M(n))$
			Goldschmidt	$O(M(n))$
Polynomial evaluation	$n$ fixed-size polynomial coefficients	One fixed size	Horner's method	$O(n)$
			Direct evaluation	$O(n)$

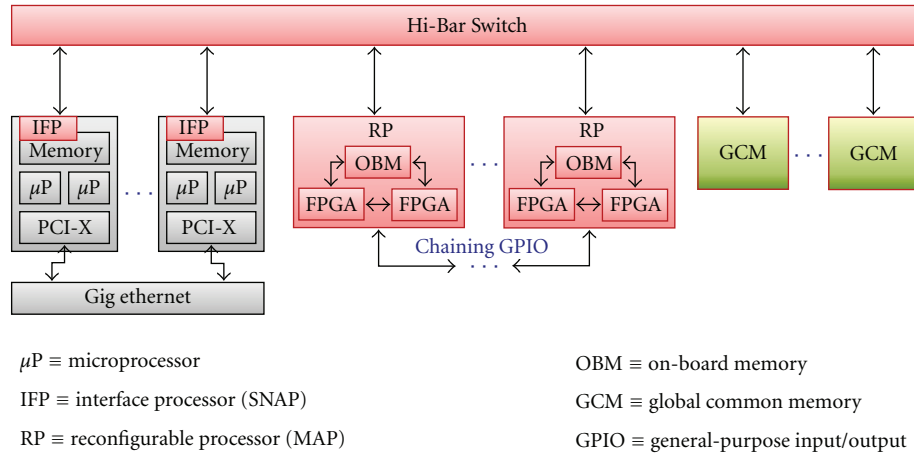


FIGURE 1: Hardware architecture of SRC-6.

The asymptotic computational complexity of each operation depends on the bit length of operands [5, 18], see Table 1. Our formal representation of the problem will consider only the multiplication operation. This is based on the fact that multiplication is a core operation from which the other basic arithmetic operations can be easily derived, for example, division as a multiplication using Newton-Raphson's approximation or Goldschmidt algorithm [19, 20]. Our proposed arithmetic unit can perform arbitrary-precision addition, subtraction, multiplication, as well as convolution operations. We decided to follow the Basecase/Schoolbook algorithm. Although this algorithm is not the fastest algorithm having a complexity of  $O(n^2)$ , see Table 1, it is the simplest and most straightforward algorithm with the least overhead. In addition, this algorithm is usually the starting point for almost all available software implementations of arbitrary-precision arithmetic. For example,

in the case of GMP, the Basecase algorithm is used up to a predetermined operand size threshold, 3000–10,000 bit length depending on the underlying microprocessor architecture, beyond which the software adaptively switches to a faster algorithm [20, 21].

The main challenge of implementing arbitrary-precision arithmetic in HPRC machines is the physical/spatial limitations of the reconfigurable device. In other words, the reconfigurable device (FPGA) has limited physical resources, which makes it unrealistic to accommodate for the resource requirements of arbitrarily large-precision arithmetic operators. Therefore, the problem can be formulated as given a fixed-precision arithmetic unit, for example,  $p$ -digit by  $p$ -digit multiplier, how to implement an arbitrary-precision arithmetic unit, for example, arbitrary large-variable-size  $m_1$ -digit by  $m_2$ -digit multiplier. Typically,  $p$  is dependent on the underlying hardware word-length, for example, 32-bit

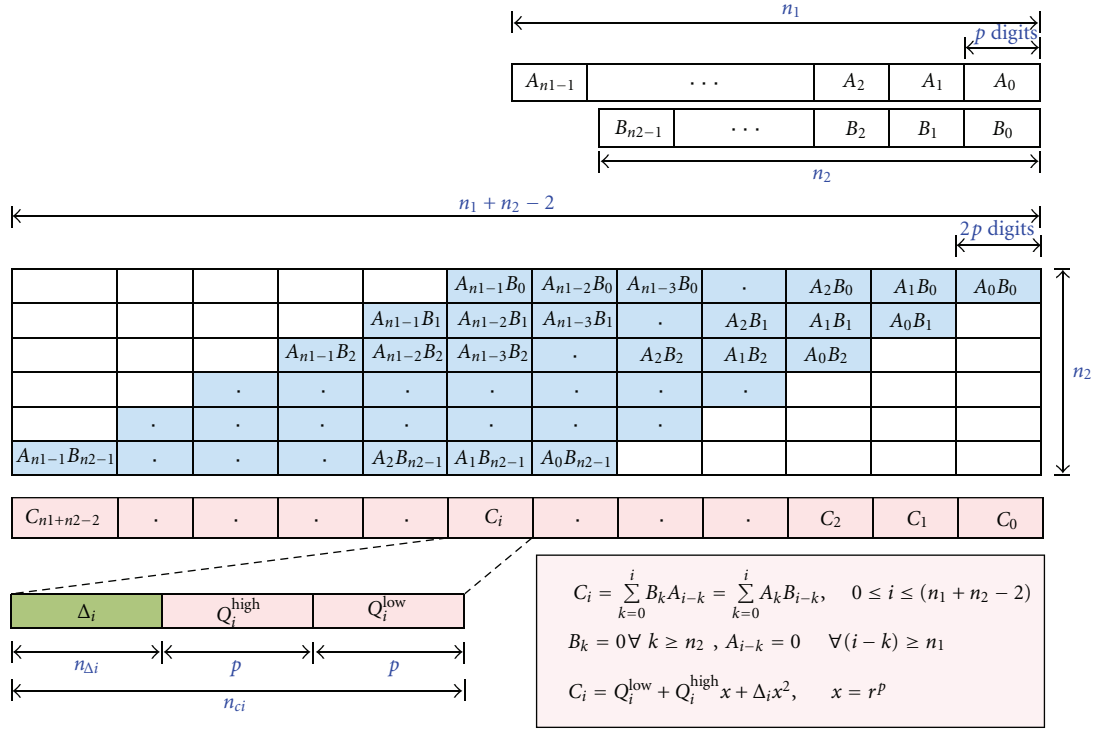


FIGURE 2: Multiplication matrix of high-precision numbers.

or 64-bit. In achieving this objective, our approach is based on leveraging previous work and concepts that were introduced for solving similar problems. For example, Tredennick and Welch [22] proposed architectural solutions for variable-length byte string processing. Similarly, Olariu et al. [23] formally analyzed and proposed solutions for the problem of sorting arbitrary large number of items using a sorting network of small fixed I/O size. Finally, ElGindy and Ferizis [24] investigated the problem of mapping recursive algorithms on reconfigurable hardware.

#### 4. Approach and Architectures

**4.1. Formal Problem Representation.** An arbitrary-precision  $m$ -digit number in arbitrary numeric base  $r$  can be represented by

$$A = a_0 + a_1 r + a_2 r^2 + \dots + a_{m-1} r^{m-1}, \quad (1)$$

$$A = \sum_{j=0}^{m-1} a_j r^j, \quad 0 \leq a_j < r.$$

It can also be interpreted as an  $n$ -digit number with base  $r^p$ , where  $p$  is dependent on the underlying hardware word-length, for example, 32-bit or 64-bit. This is represented by (2) as follows:

$$A = \sum_{i=0}^{n-1} \sum_{j=ip}^{(i+1)p-1} a_j r^j = \sum_{i=0}^{n-1} \sum_{k=0}^{p-1} a_{k+ip} r^{k+ip}$$

$$= \sum_{i=0}^{n-1} \left[ \sum_{k=0}^{p-1} a_{k+ip} r^k \right] r^{ip} = \sum_{i=0}^{n-1} A_i r^{ip},$$

$$\text{where } A_i = \sum_{k=0}^{p-1} a_{k+ip} r^k, \quad n = \left\lceil \frac{m}{p} \right\rceil. \quad (2)$$

Multiplication, accordingly, can be formulated as shown in Figure 2 and expressed by (3). In other words, as implied by (4a), multiplication of high-precision numbers can be performed through two separate processes in sequence. The first is a low fixed precision, that is,  $p$ -digits, multiply-accumulate (MAC) process for calculating the coefficients/partial products  $C_i$ s as given by (4b). This is followed by a merging process of these coefficients/partial products into a final single high-precision product as given by (4a). Equation (4b) shows that the coefficients  $C_i$ s can be represented at minimum by  $2p$ -digit precision.

The extra digits are due to the accumulation process. Therefore,  $C_i$ s can be expressed as shown by (4c);

$$A = \sum_{i=0}^{n_1-1} A_i r^{ip} = \sum_{i=0}^{n_1-1} A_i x^i = A(x),$$

$$B = \sum_{i=0}^{n_2-1} B_i r^{ip} = \sum_{i=0}^{n_2-1} B_i x^i = B(x), \quad (3)$$

$$C = AB = A(x) \cdot B(x) = C(x),$$

$$\text{where } n_1 = \left\lceil \frac{m_1}{p} \right\rceil, \quad n_2 = \left\lceil \frac{m_2}{p} \right\rceil, \quad x = r^p.$$

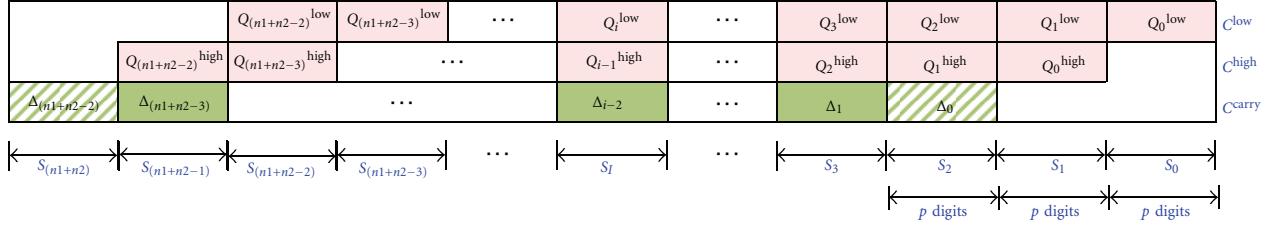


FIGURE 3: Merging schedule.

$$C(x) = C_0 + C_1x + C_2x^2 + C_3x^3 + \dots + C_{(n1+n2-2)}x^{(n1+n2-2)}, \quad (4a)$$

where

$$C_i = \sum_{k=0}^i B_k A_{i-k} = \sum_{k=0}^i A_k B_{i-k}, \quad 0 \leq i \leq (n_1 + n_2 - 2),$$

$$B_k = 0, \quad \forall k \geq n_2,$$

$$A_{i-k} = 0, \quad \forall (i - k) \geq n_1, \quad (4b)$$

$$C_i = Q_i^{\text{low}} + Q_i^{\text{high}}x + \Delta_i x^2, \quad x = r^p. \quad (4c)$$

**4.2. Multiplication as a Convolve-And-MERge (CAME) Process.** It can be easily noticed that the coefficients  $C_i$ s given by (4b) are in the form of a convolution sum. This led us to believe that virtualizing the convolution operation and using it as a scheduling mechanism will be a straightforward path for implementing multiplication and hence the remaining arithmetic operations. The different sub operands, that is,  $A$ s and  $B$ s, being stored in the system memory, will be accessed according to the convolution schedule and passed to the MAC process. The outcome of the MAC process is then delivered to the merging process which merges the partial products, according to another merging schedule, into the final results. The final results are then scheduled back into the system memory according to the same convolution schedule.

The convolution schedule, on one hand, can be derived from (4b). It is simply a process that generates the addresses/indexes for  $A$ s,  $B$ s, and  $C$ s governed by the rules given in (4b). On the hand, the merging schedule can be derived from (5) which results from substituting (4c) into (4a). Figure 3 shows the merging schedule as a high-precision addition of three components. The first component is simply a concatenation of all the first  $p$ -digits of the MAC output. The second component is a  $p$ -digit shifted concatenation of all the second  $p$ -digits of the MAC output. Finally, the third component is a  $2p$ -digit shifted concatenation of all the third  $p$ -digits of the MAC output:

$$C(x) = \sum_{i=0}^{n1+n2-2} Q_i^{\text{low}} x^i + \left( \sum_{i=0}^{n1+n2-2} Q_i^{\text{high}} x^i \right) \cdot x + \left( \sum_{i=0}^{n1+n2-2} \Delta_i x^i \right) \cdot x^2,$$

$$C(x) \equiv C^{\text{low}} + C^{\text{high}} \cdot x + C^{\text{carry}} \cdot x^2,$$

$$\text{where } C^{\text{low}} = \sum_{i=0}^{n1+n2-2} Q_i^{\text{low}} x^i, \quad C^{\text{high}} = \sum_{i=0}^{n1+n2-2} Q_i^{\text{high}} x^i,$$

$$C^{\text{carry}} = \sum_{i=0}^{n1+n2-2} \Delta_i x^i. \quad (5)$$

The merging schedule, as described above, is a high-precision schedule which will work only if the merging process is performed after the MAC process has finished completely. Given the algorithm complexity  $O(n^2)$  and allowing the two processes to work sequentially one after another would dramatically impact the performance. However, modifying the merging process to follow a small-fixed-precision scheduling scheme that works in parallel and in synchrony with the MAC process would bring back the performance to its theoretical complexity  $O(n^2)$ . The modified merging scheme can be very easily derived either from (5) or Figure 3 resulting in (6):

$$S_i = \delta_{i-1} + Q_i^{\text{low}} + Q_{i-1}^{\text{high}} + \Delta_{i-2},$$

$$i = 0, 1, 2, \dots, (n_1 + n_2),$$

$$\delta_i = S_i \cdot x^{-1} = S_i \cdot r^{-p} = \text{SHR}(S_i, p \text{ digits}), \quad (6)$$

$$Q_k^{\text{low}} = Q_k^{\text{high}} = \Delta_k = 0 \quad \forall k < 0, k > (n_1 + n_2 - 2),$$

$$\delta_k = 0 \quad \forall k < 0, k \geq (n_1 + n_2).$$

This would mean that, as soon as the MAC process finishes one partial result,  $C_i$  in  $3p$ -digit precision, the merging process, in-place, produces a final partial result  $S_i$  in  $p$ -digit precision, see Figure 4. This precision matches the word-length of the supporting memory system which allows easy storage of the final result without stalling either the MAC or the merging process. The merging process registers the remaining high-precision digits for use in subsequent calculations of  $S_i$ s.

In addition to performing multiplication, the derived architecture in Figure 4 can also natively perform the convolution operation for sequences of arbitrary size. This is because the MAC process generates the coefficients in (4b) according to a convolution schedule and in fact they are the direct convolution result. In other words, only the MAC process is needed for the convolution operation. Furthermore, the same unit can be used to perform addition and/or

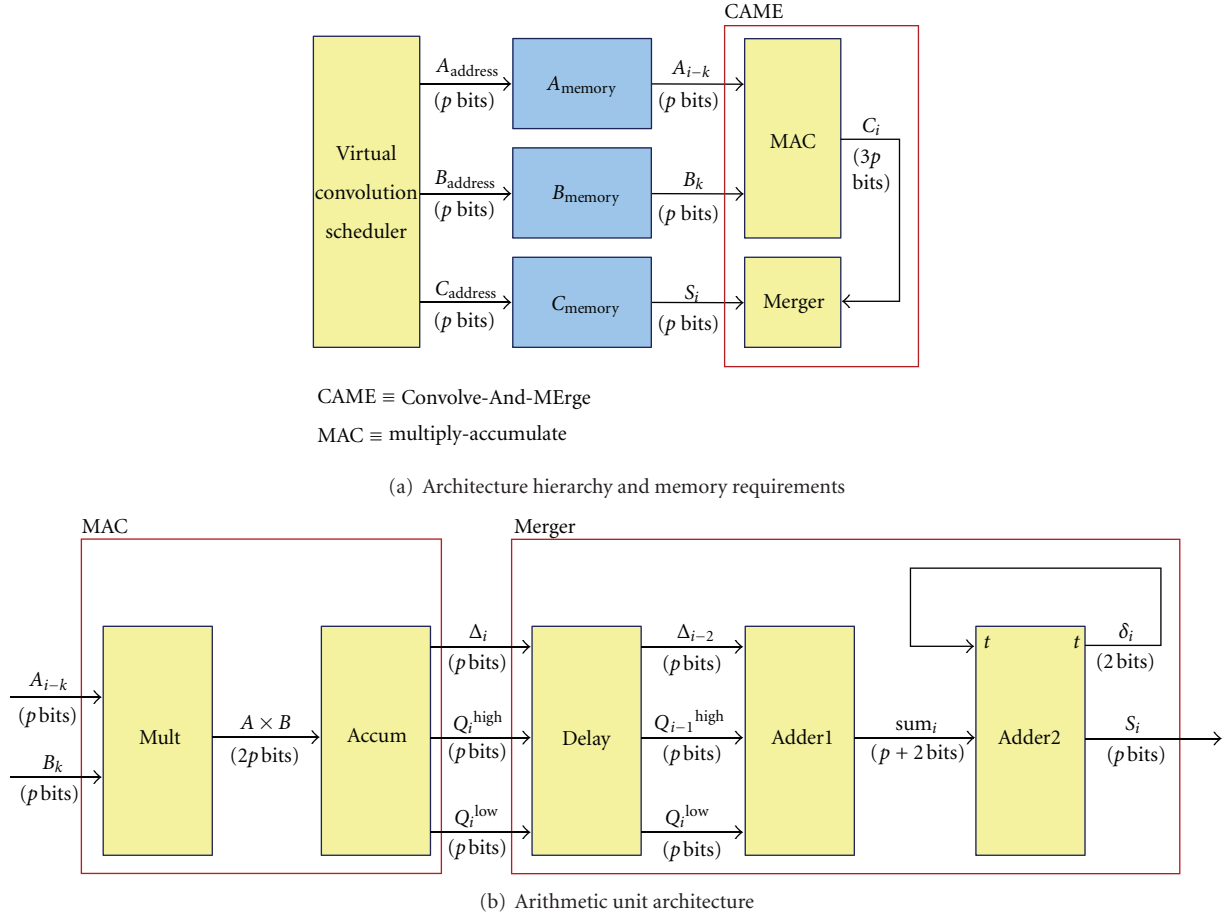


FIGURE 4: CAME architecture.

subtraction by passing the input operands directly to the merging process without going through the MAC process.

**4.3. Illustrative Example.** In order to show the steps of the proposed methodology, we will consider multiplying two decimal numbers,  $A = 987654321$  and  $B = 98765$ . We will assume that the underlying hardware word-length is 2 digits. The formal representation for this example is

$$r = 10, \quad p = 2 \Rightarrow x = r^p = 100,$$

$$A = 987654321 \Rightarrow m_1 = 9, \quad n_1 = \left\lceil \frac{m_1}{p} \right\rceil = \left\lceil \frac{9}{2} \right\rceil = 5,$$

$$B = 98765 \Rightarrow m_2 = 5, \quad n_2 = \left\lceil \frac{m_2}{p} \right\rceil = \left\lceil \frac{5}{2} \right\rceil = 3,$$

$$A(x) = \sum_{i=0}^{5-1} A_i x^i = A_4 x^4 + A_3 x^3 + A_2 x^2 + A_1 x + A_0,$$

$$A(x) = 09x^4 + 87x^3 + 65x^2 + 43x + 21,$$

$$B(x) = \sum_{i=0}^{3-1} B_i x^i = B_2 x^2 + B_1 x + B_0 = 09x^2 + 87x + 65,$$

$$C(x) = 81x^6 + 1566x^5 + 8739x^4 + 11697x^3 + 8155x^2 + 4622x + 1365.$$

(7)

As described in Section 4.2, multiplication is performed as a two-step CAME process, as shown in Figure 5. The MAC process is first applied to calculate the convolution of the two numbers, see Figure 5(a), after which the partial results are then merged, as shown in Figure 5(b), to obtain the final result.

**4.4. Precision of the Arithmetic Unit.** One challenge of implementing an arbitrary-precision arithmetic unit is to ensure that it is possible to operate with any realistic size of operands. It is therefore necessary to investigate the growth of the MAC process as it represents an upper bound for the unit precision. As discussed earlier, shown in Figure 2, and given by (4c), the outcome of the MAC process,  $C_i$ , consists of three parts: the multiply digits,  $Q_i^{\text{low}}$ ,  $Q_i^{\text{high}}$ , and the accumulation digits,  $\Delta_i$ . The corresponding number of digits,



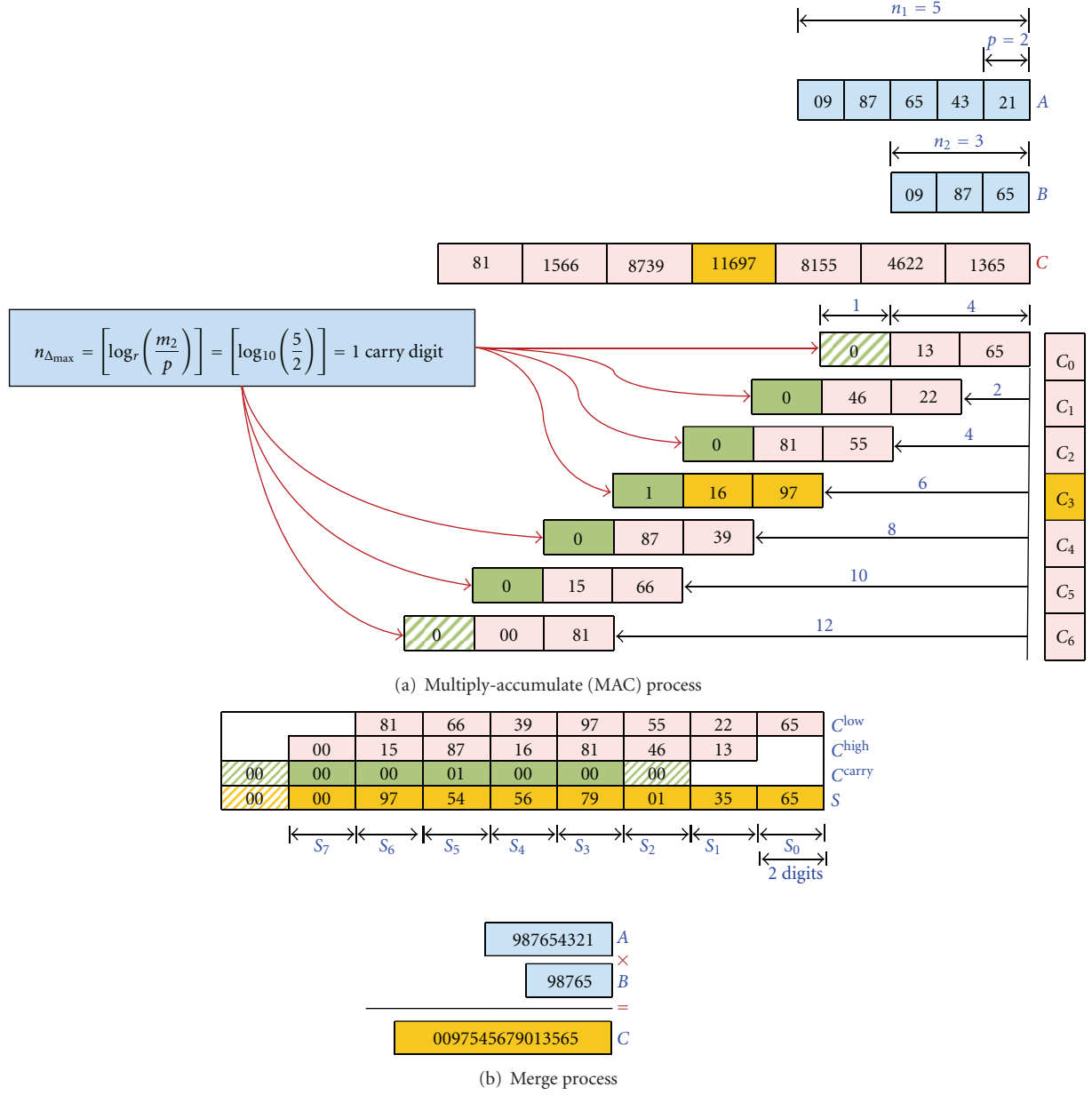


FIGURE 5: Example of decimal multiplication.

$n_{C_i}$ ,  $n_{Q_i^{\text{low}}}$ ,  $n_{Q_i^{\text{high}}}$ ,  $n_{\Delta_i}$ , can be expressed as given by (8a), (8b), (8c), and shown in Figure 6:

$$n_{C_i} = n_{Q_i^{\text{low}}} + n_{Q_i^{\text{high}}} + n_{\Delta_i}, \quad n_{Q_i^{\text{low}}} = n_{Q_i^{\text{high}}} = p, \quad (8a)$$

$$n_{\Delta_i} = \begin{cases} \log_r(i+1), & 0 \leq i \leq (n_2 - 2), \\ \log_r(n_2), & (n_2 - 1) \leq i \leq (n_1 - 1), \\ \log_r(n_1 + n_2 - 1 - i), & n_1 \leq i \leq (n_1 + n_2 - 2), \end{cases} \quad (8b)$$

$$n_{\Delta_{\max}} = \left\lceil \log_r \left( \frac{m_2}{p} \right) \right\rceil,$$

$$0 \leq n_{\Delta_{\max}} \leq p \Rightarrow p \leq m_2 \leq p \cdot r^p,$$

when

$$p = 64 \Rightarrow 64 \leq m_2 \leq 64 \cdot 2^{64} \equiv 128 \text{ EiB (ExibiByte)}. \quad (8c)$$

Controlling the growth of the MAC process by keeping the accumulation/carry digits less than or equal to  $p$ -digits, (8c) gives the upper bound on the precision of the input operands. This is in terms of the hardware unit word-length, that is,  $p$ -digits, and the numeric system base  $r$ . For example, for a binary representation, that is,  $r = 2$ , and a 64-bit arithmetic unit, that is,  $p = 64$ , the accommodated operand precision is 128 ExibiByte which is beyond any realistic storage system. In other words, a hardware unit with such parameters can provide almost infinite-precision arithmetic.

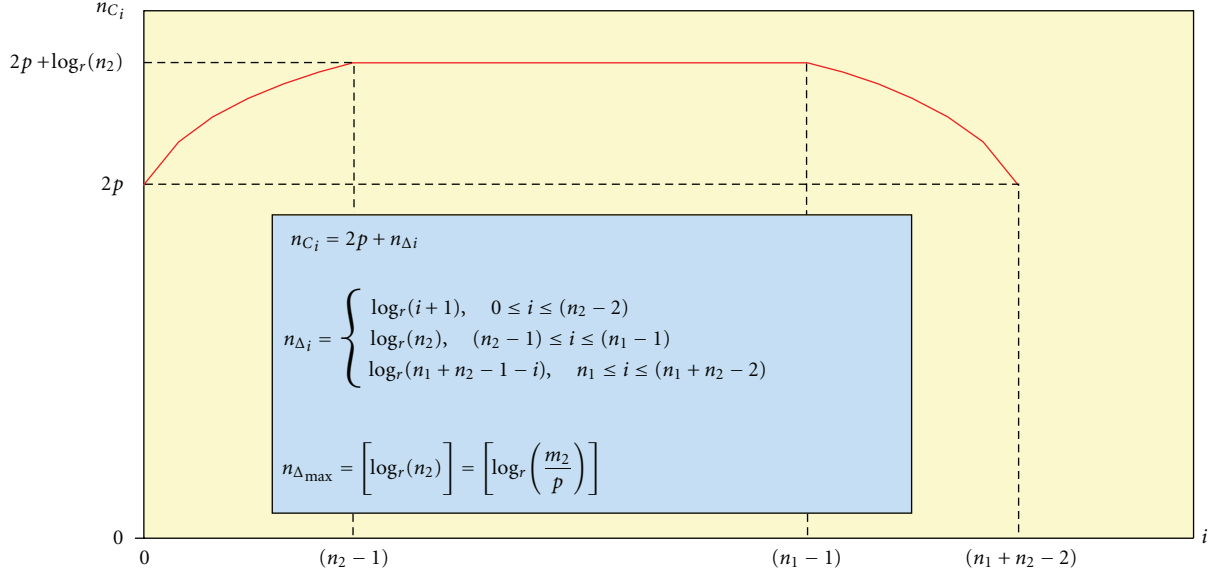


FIGURE 6: Growth of MAC process (accumulation/carry digits).

## 5. Experimental Work

Our experiments have been performed on one representative HPRC systems, SRC-6 [17], previously described in Section 2. The proposed architecture was developed partly in Xilinx System Generator environment as well as in VHDL. In both environments, the architectures were highly parameterized.

The hardware performance was referenced to one of the most efficient [21] software libraries supporting arbitrary-precision arithmetic, namely, GMP library on Xeon 2.8 GHz. We considered two versions of GMP. The first was the compiled version of GMP which is a precompiled and highly optimized version for the underlying microprocessor. The other was the highly portable version of GMP without any processor-specific optimizations.

**5.1. Implementation Issues.** Large-precision reduction operations used in both the MAC (i.e.,  $3p$ -digit accumulation) and the merging processes proved to be a challenge due to critical-path issues. For example, the accumulation of a stream of integers can be impractical for FPGA-based implementations when the number of values is large. The resultant circuit can significantly reduce the performance and consume an important portion of the FPGA.

To eliminate those effects of reduction operations, techniques of deep pipelining [25, 26] and those of nonlinear pipelining [27, 28] were considered. The buffering mechanism, presented in [25, 26], showed either low throughput and efficiency, or high latency and resources usage for our case, see Figure 7(a). Therefore, we leveraged the techniques of nonlinear pipelines [27, 28] which proved to be effective, see Figure 7(b). Furthermore, we derived a generalized architecture for nonlinear pipelined accumulation; refer to

(9). The main component of this structure is a  $p$ -digit accumulation stage in which delay elements are added for synchronization purposes, see Figure 8. The  $p$ -digit stages are arranged in a manner such that overflow digits from a given stage  $j$  are passed to the subsequent stage  $j + 1$ . The total number of stages  $n_s$  depends on the size of the input operand  $m_A$  as well as on the number of accumulated operands  $N$ , see (9). In doing so, we have proved

$$\begin{aligned}
 S &= \sum_{i=0}^{N-1} A_i, & A_i &= \sum_{j=0}^{n_A-1} A_{i,j} x^j, \\
 S &= \sum_{i=0}^{N-1} \left( \sum_{j=0}^{n_A-1} A_{i,j} x^j \right) = \sum_{j=0}^{n_A-1} \left( \sum_{i=0}^{N-1} A_{i,j} x^j \right) \\
 &= \sum_{j=0}^{n_A-1} \left( x^j \sum_{i=0}^{N-1} A_{i,j} \right), \\
 S &= \sum_{j=0}^{n_A-1} S_{N-1,j} x^j, \\
 S_{N-1,j} &= \sum_{i=0}^{N-1} A_{i,j} \iff S_{i,j} = S_{i-1,j} + A_{i,j}, \\
 &\quad \forall i \in [0, N-1], \quad S_{-1,j} = 0, \\
 S_{i,j} &= Q_{i,j} + \Delta_{i,j} \cdot x, \\
 Q_{i,j} + \Delta_{i,j} \cdot x &= Q_{i-1,j} + \Delta_{i-1,j} \cdot x + A_{i,j}, \\
 Q_{i-1,j} + A_{i,j} &= Q_{i,j} + c_{i,j} \cdot x, \\
 \implies \Delta_{i,j} &= \Delta_{i-1,j} + c_{i,j} \iff \Delta_{N-1,j} = \sum_{i=0}^{N-1} c_{i,j},
 \end{aligned}$$



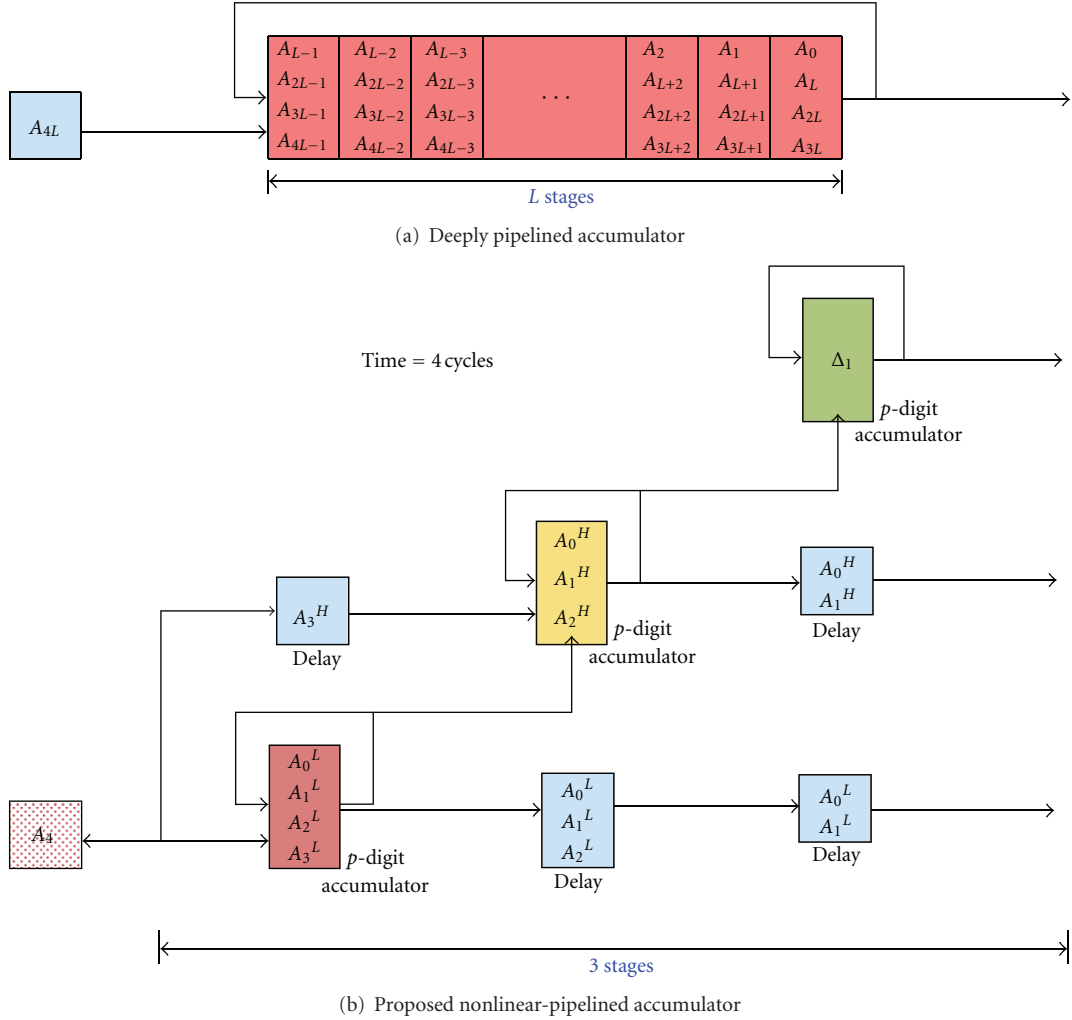


FIGURE 7: Accumulator of the MAC process.

where  $x = r^p$ ,  $m_S = m_A + m_\Delta$ ,  $m_\Delta = \log_r N$ ,

$$n_A = \left\lceil \frac{m_A}{p} \right\rceil, \quad n_\Delta = \left\lceil \frac{m_\Delta}{p} \right\rceil,$$

$$n_S = n_A + n_\Delta = \left\lceil \frac{m_A}{p} \right\rceil + \left\lceil \frac{\log_r N}{p} \right\rceil, \quad (9)$$

that large-size accumulation operations which are prone to deeply-pipelined effects and hardware critical-path issues can be substituted with multiple and faster smaller-size accumulations. In other words, a single large-size  $(p \cdot n_s)$ -digit accumulator can efficiently be implemented using  $n_s$  faster  $p$ -digit accumulators, see Figure 8.

We analyzed the pipeline efficiency, as defined in [27, 28], of our proposed architecture. This is expressed by (10a), (10b), and shown in Figure 10. The pipeline efficiency  $\eta$  as expressed by (10a) can be easily derived by considering the pipeline reservation table [27, 28]. As shown in Figure 9, the example reservation table is used to calculate the pipeline efficiency  $\eta$ . We implemented two versions of the arithmetic

unit, that is, 32-bit and 64-bit. For very large data, the efficiency for the 32-bit unit was lower bounded to 80% while the efficiency for the 64-bit unit was lower bounded to 84.62%, see (10b) and Figure 10:

$$\eta = \frac{\text{Full Cells}}{\text{Total Cells}} = 1 - \frac{\text{Empty Cells}}{\text{Total Cells}},$$

$$\eta = 1 - \frac{L_{\text{merger}}(n_1 - 1)(n_2 - 1)}{L_{\text{total}} n_1 n_2}$$

$$= 1 - \frac{1}{1 + L_{\text{mac}}/L_{\text{merger}}} \cdot \left(1 - \frac{1}{n_1}\right) \left(1 - \frac{1}{n_2}\right),$$

where  $L_{\text{mac}} \equiv$  Latency of the MAC – process,

$L_{\text{merger}} \equiv$  Latency of the merging – process,

$L_{\text{total}} = L_{\text{mac}} + L_{\text{merger}}$ ,

(10a)

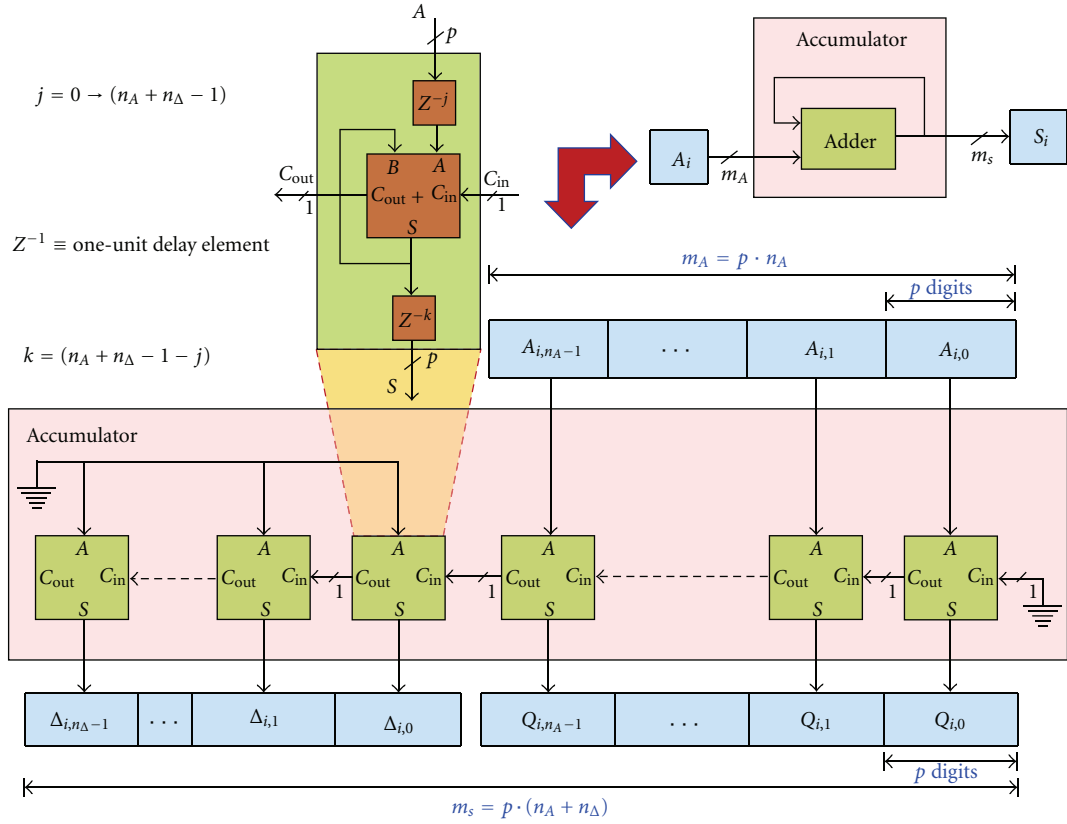


FIGURE 8: Generalized nonlinear-pipelined accumulator.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
$S_0$	X	X	X	X	X	X	X	X	X													$L_{\text{mac}}$
$S_1$		X	X	X	X	X	X	X	X	X												
$S_2$			X	X	X	X	X	X	X	X	X											
$S_3$				X	X	X	X	X	X	X	X	X										
$S_4$					X	X	X	X	X	X	X	X	X									
$S_5$						X	X	X	X	X	X	X	X	X								
$S_6$							X	X	X	X	X	X	X	X	X							
$S_7$								X	X	X	X	X	X	X	X	X						
$S_8$									X	X	X	X	X	X	X	X	X					
$S_9$										X	X	X	X	X	X	X	X	X				
$S_{10}$											X	X	X	X	X	X	X	X	X			
$S_{11}$												X		X			X		X	X	$L_{\text{merger}}$	
$S_{12}$													X		X			X		X		X

Data size =  $3p$  bits (i.e.,  $n_1 = n_2 = 3$ )

$$\eta = \frac{\text{Full cells}}{\text{Total cells}} = 1 - \frac{\text{Empty cells}}{\text{Total cells}} = 1 - \frac{2 \times 4}{13 \times 9} = 93.16\%$$

FIGURE 9: Example pipeline reservation table.

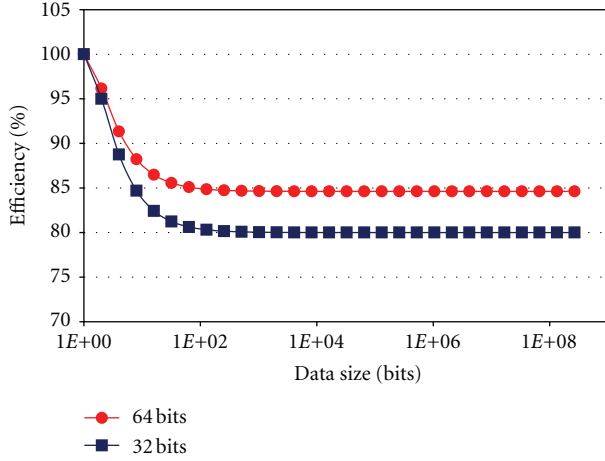


FIGURE 10: Pipeline efficiency.

TABLE 2: FPGA resource utilization.

	32-bit unit	64-bit unit
Slice flip flops	6,155 (9%)	9,183 (13%)
4-input LUTs	2,524 (3%)	6,311 (9%)
Occupied slices	3,983 (11%)	6,550 (19%)
18 × 18b multipliers	4 (2%)	10 (6%)
Clock frequency	102.8 MHz	100.4 MHz

$$\eta_{\infty} = \lim_{n_1, n_2 \rightarrow \infty} \eta = \frac{L_{\text{mac}}}{L_{\text{total}}},$$

when

$$\begin{aligned} p = 32 \text{ bits}, \quad L_{\text{mac}} = 4, \quad L_{\text{merger}} = 1 &\Rightarrow \eta_{\infty} = 80.00\%, \\ p = 64 \text{ bits}, \quad L_{\text{mac}} = 11, \quad L_{\text{merger}} = 2 &\Rightarrow \eta_{\infty} = 84.62\%. \end{aligned} \quad (10b)$$

**5.2. Experimental Results.** Our experiments were performed for two cases, that is, 32-bit and 64-bit units. FPGA resources usage and clock frequency are shown in Table 2. While resource usage in the 64-bit unit is larger, as expected, clock frequency is similar in both cases due to the clock frequency requirement imposed by SRC-6 (100 MHz). As it can be seen in Table 2, the proposed architecture consumes relatively low hardware resources requiring at maximum 19% of the reconfigurable device for the 64-bit unit. These implementation results allow taking advantage of the inherent parallelism of the FPGA and adding more than one operational unit upper bounded by the number of available memory banks. For example, in SRC-6, there are six memory banks which make it possible to allocate two units per FPGA, see Figure 4(a).

Next, we show the results of the 64-bit Basecase algorithm for addition/subtraction, and the 32-bit and 64-bit Basecase algorithm for multiplication.

The arbitrary-precision addition/subtraction performance was measured on SRC-6 and compared to both

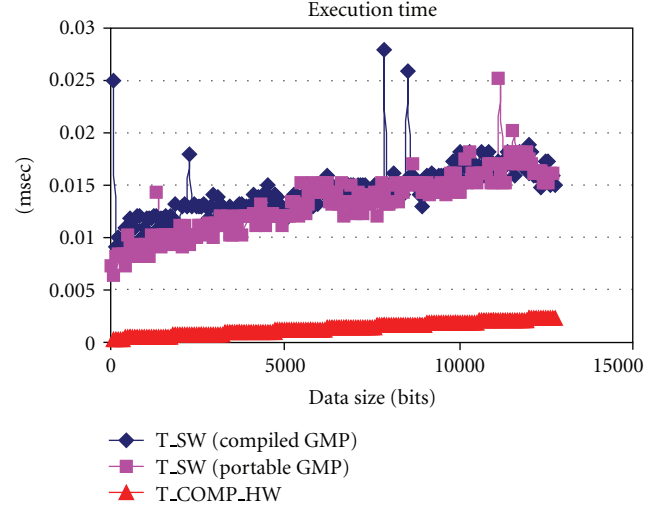


FIGURE 11: Addition/subtraction execution time (64-bit).

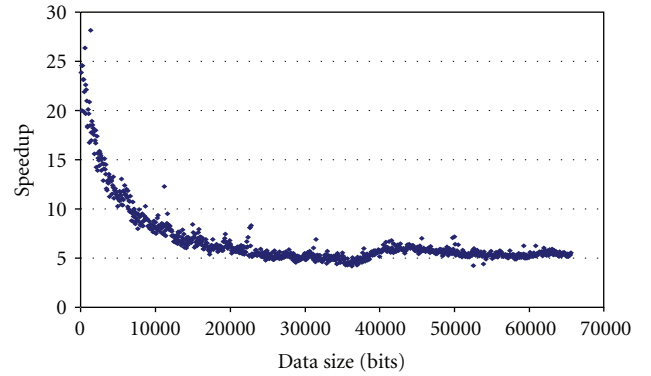


FIGURE 12: Addition/subtraction hardware speedup versus GMP (64-bit).

the compiled and portable versions of GMP. As shown in Figure 11, T\_COMP\_HW, that is, the total computation time of the hardware on SRC-6, is lower than the execution time of both the compiled and portable versions of GMP. The performance speedup is shown in Figure 12. The hardware implementation asymptotically outperforms the software, by a factor of approximately 5, because of the inherent parallelism exploited by the hardware. We can also notice that, for small-precision addition/subtraction, the speedup factor starts from approximately 25. This is due to the large overhead, relative to the data size, associated with the software, while the only overhead associated with the hardware is due to the pipeline latency. This latency is independent on the data size. It is also worth to notice the linear behavior,  $O(n)$ , of both the software and the hardware. This is because both execute the same algorithm, that is, Basecase addition/subtraction [20, 21], see Table 1.

In the case of multiplication, we notice a nonlinear behavior  $O(n^{1+e})$ ,  $0 < e < 1$ ; see Figure 13 and Table 1. We notice also a similar behavior to the addition/subtraction for small-size operands. Figures 13(a) and 13(b) show a significant performance for the hardware compared to the portable

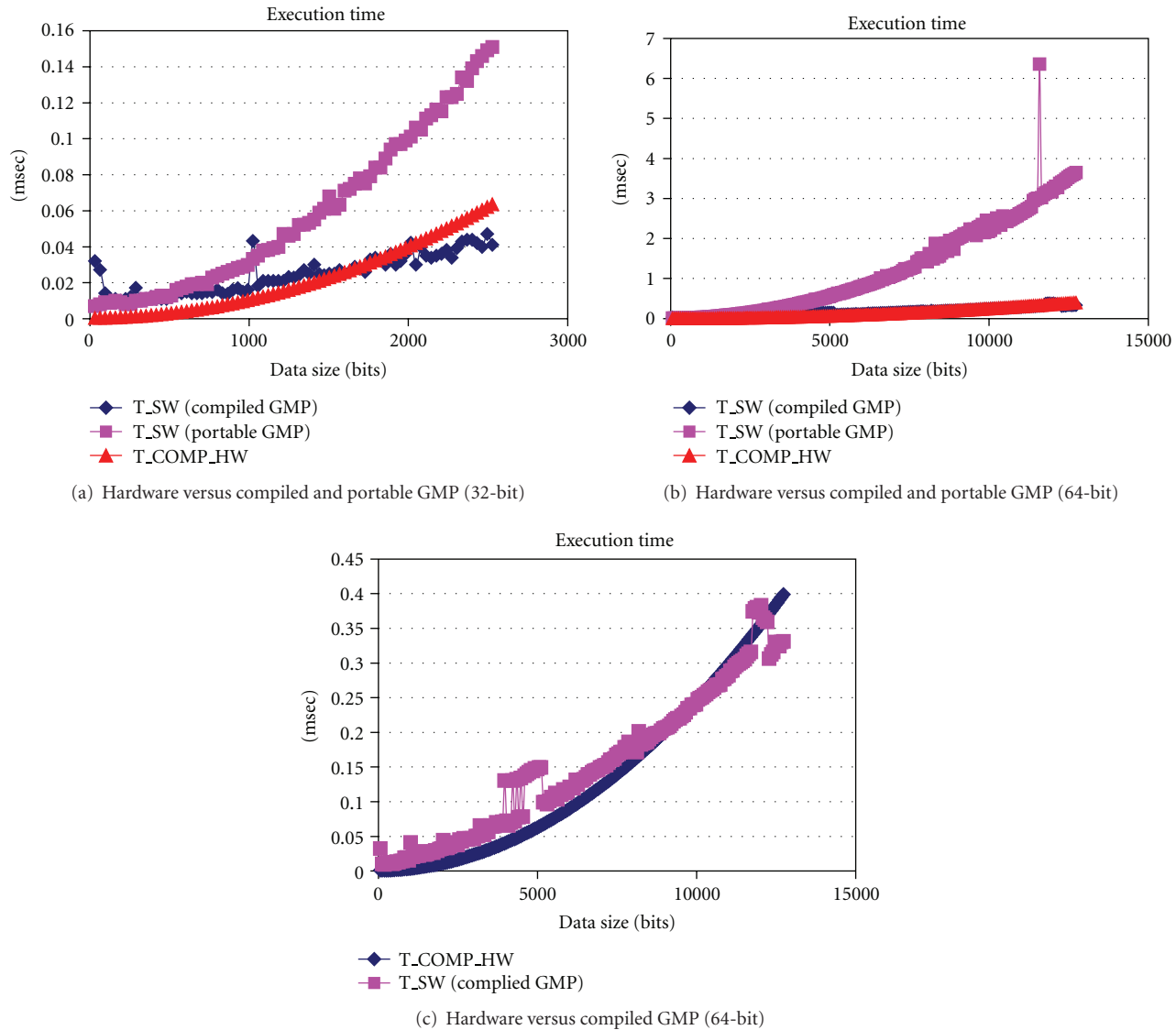


FIGURE 13: Multiplication execution time.

version of GMP. This is because this version of GMP uses the same algorithm as ours, that is, Basecase with  $O(n^2)$  see Table 1, independent of the data size [20, 21]. As shown in Figure 14, the hardware behavior asymptotically outperforms the portable GMP multiplication by a factor of approximately 2 for the 32-bit multiplication, see Figure 14(a), and 9 for the 64-bit multiplication, see Figure 14(b). However, this is not the case with the compiled GMP multiplication which is highly optimized and adaptive. Compiled GMP uses four multiplication algorithms, and adaptively switches from a slower to a faster algorithm depending on the data size and according to predetermined thresholds [20, 21]. For these reasons, the hardware, as can be seen from Figure 13(c), outperforms the compiled GMP up to a certain threshold, approximately 10 Kbits, beyond which the situation reverses.

## 6. Conclusions and Future Work

This paper shows the feasibility of accelerating arbitrary-precision arithmetic on HPRC platforms. While exact computation presents many benefits in terms of numerical robustness, its main drawback is the poor performance that is obtained in comparison to fixed-precision arithmetic. The results presented in this work show the possibility of reducing the performance gap between fixed-precision and arbitrary-precision arithmetic using HPRC machines.

The proposed solution, the Convolve-And-Merge (CAME) methodology for arbitrary-precision arithmetic, is derived from a formal representation of the problem and is based on virtual convolution scheduling. For the formal analysis, only the multiplication operation was considered. This decision was made due to the fact that multiplication

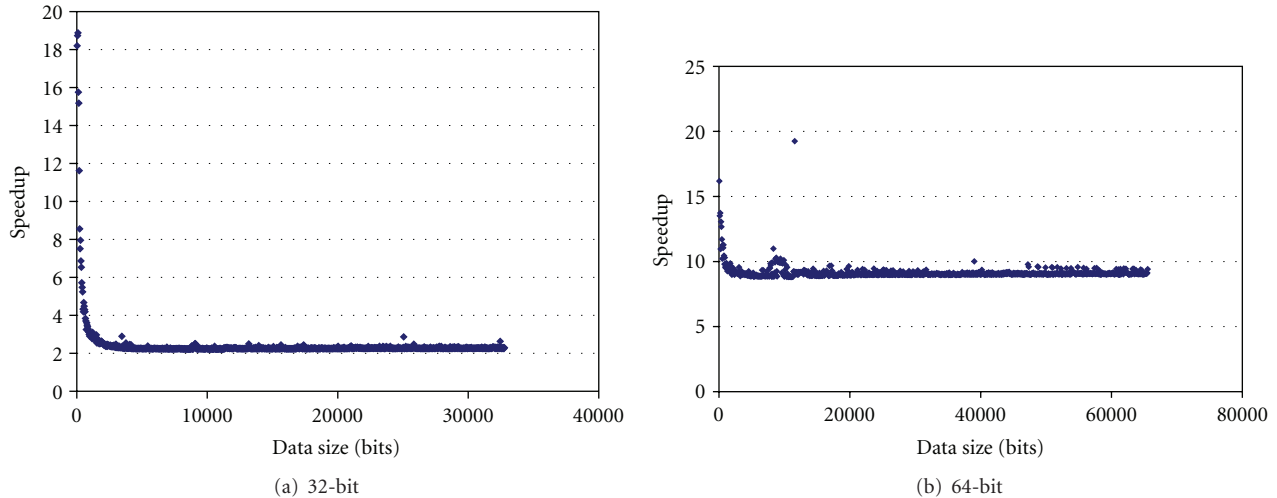


FIGURE 14: Multiplication hardware speedup versus portable GMP.

is a core operation from which other basic arithmetic operations, such as division and square root, can be easily derived.

Our proposed arithmetic unit can perform arbitrary-precision addition, subtraction, multiplication, as well as arbitrary-length convolution operations. Our approach in implementing the CAME process was based on leveraging previous work and concepts that were introduced for solving similar problems. Dynamic (nonlinear) pipelines techniques were exploited to eliminate the effects of deeply pipelined reduction operators. The use of these techniques allowed us reaching a minimum of 80% pipeline utilization for 32-bit units and reaching 84.6% efficiency for 64-bit units. This implementation was verified for both correctness and performance in reference to the GMP library on the SRC-6 HPRC. The hardware outperformed GMP by a factor of 5x speedup for addition/subtraction, while the speedup factor was lower bounded to 9x compared to the portable version of GMP multiplication.

Future directions may include investigating hardware support for floating-point arbitrary precision, considering faster algorithms than the Basecase/Schoolbook presented in this paper, as well as adopting methods for adaptive algorithm switching based on the length of the operands. In addition to, full porting of an arbitrary-precision arithmetic library such as GMP to HPRC machines might also be favorable.

## References

- [1] V. Sharma, *Complexity analysis of algorithms in algebraic computation*, Ph.D. dissertation, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 2007.
- [2] C. K. Yap and T. Dube, "The exact computation paradigm," in *Computing in Euclidean Geometry*, D. Z. Du and F. K. Hwang, Eds., vol. 4 of *Lecture Notes Series on Computing*, pp. 452–492, World Scientific Press, Singapore, 2nd edition, 1995.
- [3] D. E. Knuth, "The art of computer programming," in *Seminumerical Algorithms*, vol. 2, Addison-Wesley, 3rd edition, 1998.
- [4] [http://en.wikipedia.org/wiki/Arbitrary\\_precision\\_arithmetic](http://en.wikipedia.org/wiki/Arbitrary_precision_arithmetic).
- [5] C. Li, *Exact geometric computation: theory and applications*, Ph.D. dissertation, Department of Computer Science, Institute of Mathematical Sciences, New York University, 2001.
- [6] [http://bitsavers.org/pdf/ibm/140x/A24-1401-1\\_1401\\_System\\_Summary\\_Sep64.pdf](http://bitsavers.org/pdf/ibm/140x/A24-1401-1_1401_System_Summary_Sep64.pdf).
- [7] <http://ibm-1401.info/1401-Competition.html#IntroHoneywell200>.
- [8] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (Revisiting Iterative Refinement for Linear Systems)," in *Proceedings of the ACM/IEEE SC Conference*, Tampa, Fla, USA, November 2006.
- [9] J. Hormigo, J. Villalba, and E. L. Zapata, "CORDIC processor for variable-precision interval arithmetic," *Journal of VLSI Signal Processing Systems*, vol. 37, no. 1, pp. 21–39, 2004.
- [10] S. Balakrishnan and S. K. Nandy, "Arbitrary precision arithmetic—SIMD style," in *Proceedings of the 11th International Conference on VLSI Design: VLSI for Signal Processing*, p. 128, 1998.
- [11] A. Saha and R. Krishnamurthy, "Design and FPGA implementation of efficient integer arithmetic algorithms," in *Proceedings of the IEEE Southeastcon '93*, vol. 4, no. 7, April 1993.
- [12] D. M. Chiarulli, W. G. Rudd, and D. A. Buell, "DRAFT—a dynamically reconfigurable processor for integer arithmetic," in *Proceedings of the 7th Symposium on Computer Arithmetic*, pp. 309–321, IEEE Computer Society Press, 1989.
- [13] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *IEEE Computer*, vol. 41, no. 2, pp. 69–76, 2008.
- [14] A. Michalski, D. Buell, and K. Gaj, "High-throughput reconfigurable computing: design and implementation of an idea encryption cryptosystem on the SRC-6e reconfigurable computer," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 681–686, August 2005.
- [15] E. El-Araby, T. El-Ghazawi, J. Le Moigne, and K. Gaj, "Wavelet spectral dimension reduction of hyperspectral imagery on

- a reconfigurable computer,” in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '04)*, pp. 399–402, Brisbane, Australia, December 2004.
- [16] E. El-Araby, M. Taher, T. El-Ghazawi, and J. Le Moigne, “Prototyping Automatic Cloud Cover Assessment (ACCA) algorithm for remote sensing on-board processing on a reconfigurable computer,” in *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT '05)*, pp. 207–214, Singapore, December 2005.
  - [17] SRC Computers, *SRC Carte C Programming Environment v2.2 Guide (SRC-007-18)*, 2006.
  - [18] [http://en.wikipedia.org/wiki/Computational\\_complexity\\_of\\_mathematical\\_operations](http://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations).
  - [19] T. Granlund and P. L. Montgomery, “Division by invariant integers using multiplication,” in *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI '94)*, pp. 61–72, June 1994.
  - [20] GMP Manual, *GNU MP The GNU Multiple Precision Arithmetic Library*, 4.2.1 edition, 2006.
  - [21] <http://gmplib.org/>.
  - [22] H. L. Tredennick and T. A. Welch, “High-speed buffering for variable length operands,” *Proceedings of the 4th Annual Symposium on Computer Architecture (ISCA '77)*, vol. 5, no. 7, pp. 205–210, March 1977.
  - [23] S. Olariu, M. C. Pinotti, and S. Q. Zheng, “How to sort N items using a sorting network of fixed I/O size,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 5, pp. 487–499, 1999.
  - [24] H. ElGindy and G. Ferizis, “Mapping basic recursive structures to runtime reconfigurable hardware,” in *Proceedings of the FPL*, August 2004.
  - [25] L. Zhou, G. R. Morris, and V. K. Prasanna, “High-performance reduction circuits using deeply pipelined operators on FPGAs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 10, pp. 1377–1392, 2007.
  - [26] L. Zhuo and V. K. Prasanna, “High-performance and area-efficient reduction circuits on FPGAs,” in *Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '05)*, pp. 52–59, Rio de Janeiro, Brazil, October 2005.
  - [27] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGrawHill, 1993.
  - [28] K. Hwang and Z. Xu, *Scalable Parallel Computing: Technology, Architecture, Programming*, McGrawHill, 1998.



