Methods for Reducing Floating-Point Computation Overhead

By

JON PIMENTEL
B.S. (University of California, Davis) June 2009
M.S. (University of California, Davis) December 2015

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Bevan M. Baas, Chair

Professor Soheil Ghiasi

Professor Matthew K. Farrens

Committee in charge
2017

Abstract

Despite floating-point (FP) being the most commonly used method for real number representation [1], certain architectures are still limited to fixed-point arithmetic due to the large area and power requirements of FP hardware. A software library, which emulates FP functions, is typically implemented when FP calculations need to be performed on a platform with a fixed-point datapath. However, software implementations of FP operations, despite not requiring any additional area, suffer from a low throughput. Conversely, hardware FP implementations provide high throughput, but require a large amount of additional area and consequently increase leakage. Therefore, it is desirable to increase the FP throughput provided by a software implementation without incurring the area overhead of a full hardware floating-point unit (FPU). Furthermore, the widths of data words in digital processors have a direct impact on area in application-specific ICs (ASICs) and field-programmable gate arrays (FPGAs). Circuit area impacts energy dissipation per workload and chip cost. Graphics and image processing workloads are very FP intensive, however, little exploration has been done into modifying FP word width and observing its effect on image quality and chip area.

This dissertation first presents hybrid FP implementations, which improve software FP performance without incurring the area overhead of full hardware FPUs. The proposed implementations are synthesized in 65 nm complementary metal oxide semiconductor (CMOS) technology and integrated into small fixed-point processors which use a reduced instruction set computing (RISC)-like architecture. Unsigned, shift-carry, and leading zero detection (USL) support is added to the processors to augment the existing instruction set architecture (ISA) and increase FP throughput with little area overhead. Two variations of hybrid implementations are created. USL support is additional general purpose hardware that is not specific to FP workloads (e.g., unsigned operation support), custom FP-specific (CFP) hardware is specifically for FP workload acceleration (e.g., exponent calculation logic). The first, hybrid implementations with USL support, increase software FP throughput per core by 2.18× for addition/subtraction, 1.29× for multiplication, 3.07–4.05× for division, and 3.11–3.81× for square root, and use 90.7–94.6% less area than dedicated fused multiply-add (FMA) hardware. The second type of hybrid implementations, those with CFP hardware, increase throughput per core over a fixed-point software kernel by 3.69–7.28× for addition/subtraction, 1.22–2.03× for multiplication, 14.4× for division, and 31.9× for square root, and use 77.3–97.0% less area than dedicated fused multiply-add hardware. The circuit area and throughput are found for 38 multiply-add, 8 addition/subtraction, 6 multiplication, 45 division, and 45 square root designs. 33 multiply-add implementations are presented which improve throughput per core versus a fixed-point software implementation by 1.11–15.9× and use 38.2–95.3% less area than dedicated FMA hardware.

In addition to proposing hybrid FP implementations, this dissertation investigates the effects of modifying FP word width. For the second portion of this dissertation, FP exponent and mantissa widths are independently varied for the seven major computational blocks of an airborne synthetic aperture radar (SAR) image formation engine. This image formation engine uses the backprojection algorithm [2]. SAR imaging uses pulses of microwave energy to provide day, night, and all-weather imaging and can be used for reconnaissance, navigation, and environment monitoring [3]. The backprojection algorithm is a frequently used tomographic reconstruction method similar to that used in computed tomography (CT) imaging [2, 4]. Additionally, trigonometric function evaluation, interpolation, and Fourier transforms are common to SAR backprojection and other biomedical image formation algorithms [5]. The circuit area in 65 nm CMOS and the peak signal-to-noise ratio (PSNR) and structural similarity index metric (SSIM) are found for 572 design points. With word width reductions of 46.9–79.7%, images with a 0.99 SSIM are created with imperceptible image quality degradation and a 1.9–11.4× area reduction.

The third portion of this dissertation covers the physical design of two many-core chips in 32 nm PD-SOI, KiloCore [6] and KiloCore2. In the first portion of this section, the design of KiloCore is covered, while the second portion details the adjustments made to the flow for the tape-out of KiloCore2. KiloCore features 1000 cores capable of independent program execution. The maximum clock frequency for the cores on KiloCore range from

1.70 GHz to 1.87 GHz at 1.10 V. KiloCore compares favorably against many other many-core and multi-core chips, as well as low power processors. At a supply voltage of 0.56 V, processors require 5.8 pJ per operation at a clock frequency of 115 MHz. KiloCore2 has 700 cores, 697 of which are programmable processor tiles, and three which are hardware accelerators (a fast Fourier transform (FFT) accelerator, and two Viterbi decoders). The assembled printed circuit boards (PCBs) with packaged KiloCore2 chips are expected to be ready in July.

The fourth portion of this dissertation explores implementing a scientific kernel on a many-core array, namely sparse matrix-vector multiplication. Twenty-three functionally equivalent sparse matrix times dense vector multiplication implementations are created for a fine-grained many-core platform with FP capabilities. These implementations are considered against two central processing unit (CPU) chips and two graphics processing unit (GPU) chips. The designs for the many-core array, CPUs, and GPUs are evaluated using the metrics of throughput per area and throughput per watt when operating on a set of five unstructured sparse matrices of varying dimensions, sourced from a wide range of domains including directed weighted graphs, computational fluid dynamics, circuit simulation, thermal problems (e.g., heat exchanger design), and eigenvalue/model reduction problems. Results using unscheduled and unoptimized code demonstrate that the implementations on the many-core platform increase power efficiency by up to 14.0x versus the CPU implementations, and by up to 27.9x versus the GPU implementations. Additionally, the implementations on the many-core platform increase area efficiency by as much as 17.8x versus the CPU implementations, and up to 36.6x versus the GPU implementations.

I dedicate this dissertation to my entire family.

Without their encouragement and love, this dissertation would not have been possible.

Acknowledgments

I would like to first thank my advisor Professor Bevan Baas. Professor Baas is a brilliant researcher who has guided and supported me throughout my undergraduate and graduate career over the past several years. He has helped develop me into a more productive researcher and taught me the importance of self-discipline, and writing clearly and accurately. I have been able to expand both my intellectual curiosity and ability to work independently. His guidance has allowed me to both see the fine-detail, as well as the big picture.

I would like to thank Professor Soheil Ghiasi and Professor Matthew Farrens for serving on my dissertation and qualifying examination committees. Thank you for taking the time to read this dissertation and provide me with valuable feedback. I would also like to thank Professor Rajeevan Amirtharajah and Professor Jinyi Qi for taking the time to serve on my qualifying examination committee and for their feedback on this work.

Thank you to Professor Bevan Baas, Professor Soheil Ghiasi, and Professor Venkatesh Akella, for whom I have had the privilege to serve as a teaching assistant for EEC 281, EEC 180A, and EEC 180B, respectively. These experiences taught me to be an effective leader, enhanced my ability to communicate clearly, and helped me develop a command of the subject matter.

Without the continued love and support from my family I wouldn't have been able to reach this milestone in my life, words cannot fully express the gratitude I have for all of them. Thank you to my wife, Frances, for your constant encouragement and support. Thank you to my mother and father, Vidalia and Tony, for always being there, providing for me, and encouraging me to do my best. Thank you to my brother, Jeffrey, for inspiring me to seek out every opportunity to grow and to never stop learning. Thank you to my sister-in-law, Nicole, my nieces Grace, Abigail, Olivia, and Elizabeth, my Uncle Lou, my Aunt Frankie, and James, for your support. Thank you also to my late grandmother and grandfather, Maria and Joe Cardoso, for their continual support throughout life and school and for helping raise me. Thank you to my late grandmother, Rosa Pimentel, for always being there to help raise me.

I thank all of the students in the VLSI Computation Laboratory, past and present, especially my coauthors, Aaron Stillmaker, Brent Bohnenstiehl, Timothy Andreas, Bin Liu, Anh Tran, and Emmanuel Adeagbo. Thank you to all of the VCL members I have had the privilege of working alongside whom I haven't already mentioned, Dean Truong, Tinoosh Mohsenin, Jeremy Webb, Nima Mostafavi, Trevin Murakami, Houshmand Mehr, Lucas Stillmaker, Stephen Le, Zhibin Xiao, Satyabrata Sarangi, Shifu Wu, and Mark Hildebrand. Thank you to my coauthors from other research groups, Philipp Gysel, Mohammad Motamedi, and Professor Soheil Ghiasi. Thank you to my colleagues in other research labs (Khadar, Stanley, Vipul, Scott, Volodymyr), from whom I received useful feedback.

I thank my colleagues at Intel in the Many Integrated Core Group with whom I had the pleasure of discussing my research and working with including, Brian Hickmann, Aaresh Powvalla, Tom Fletcher, Niravkumar Patel, Henry Slonsky, and Matthew Young.

I would also like to thank the Air Force Research Laboratory for providing the radar data sets and specifically LeRoy Gorham for his feedback.

I would also like to acknowledge the following sources of funding which have helped make this work possible: C2S2 Grant 2047.002.014, NSF Grant 1018972 and 0903549 and CAREER Award 0546907, SRC GRC Grant 1598, 1971, and 2321 and CSR Grant 1659, DoD and ARL/ARO Grant W911NF-13-1-0090, and SEM, the GAANN fellowship, the Frank and Carolan Walker Fellowship, the George S. and Marjorie Butler Fellowship, the UCD & Humanities Graduate Research Award, the ECE Graduate Program Fellowship, the Graduate Research Mentorship Fellowship, the ECE TA Program Support Fellowship, the Herbert Tryon Fellowship, the Laura Perrot Mahan Fellowship, the ECE Travel Grant Award, and the Dissertation Writing Fellowship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Floating-point (FP) representation is the most commonly used method for approximating real numbers in modern computers [1]. However, the large area and power requirements of FP hardware limit many architectures to the use of fixed-point arithmetic, for example, software-defined radio architectures [11], Blackfin microprocessors [12], picoChip [13], the Xscale core [14], and massively parallel processor chips such as AsAP [7, 15]. Small chip area is especially critical for many-core architectures, since increasing area per core has a dramatic effect on total chip area and can effectively reduce the number of cores that will fit on a chip die. There is also interest in adding embedded floating-point units (FPUs) in field-programmable gate arrays (FPGAs) [16], though most commercial vendors do not offer dedicated hard-block FPUs due to the large area overhead [17].

This dissertation presents hybrid FP implementations, which perform FP arithmetic on a small fixed-point processor using a combination of fixed-point software instructions and additional hardware [18, 19, 20]. Hybrid implementations offer alternative area-throughput trade-offs to full software or full hardware approaches (Figure 1.1). They provide higher throughput than full software kernels by including either custom FP-specific (CFP) instructions or unsigned, shift-carry, and leading zero detection (USL) support, which replace long sections of code thereby performing the same operation in fewer cycles. This dissertation demonstrates that hybrid implementations require less area than conventional full hardware modules by using the existing fixed-point hardware, such as the arithmetic logic unit (ALU) and multiply-accumulate (MAC) unit.

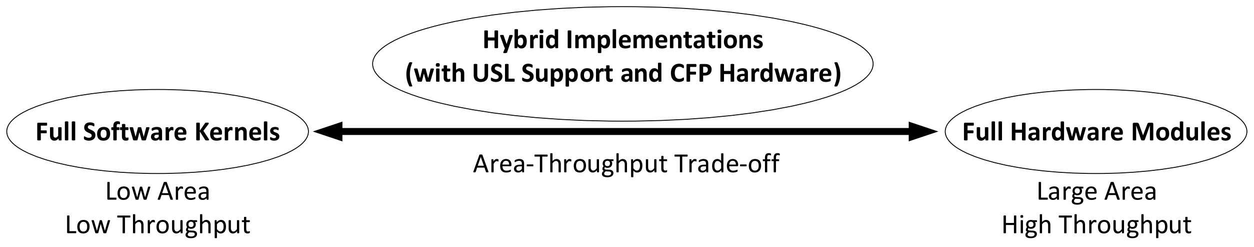


Figure 1.1: Hybrid implementations offer alternatives to pure-software and pure-hardware designs and enable a spectrum of designs with varying levels of chip area and throughput.

The hybrid implementations with USL support are added to a simple fixed-point processor to determine the area and throughput trade-offs provided by minimal architectural improvements to the instruction set architecture (ISA). These architectural improvements increase FP throughput without adding FP-specific hardware. On the other hand, the hybrid implementations with CFP add instructions that are specific to FP. The USL ISA modifications include adding unsigned operation support, leading zeros detection, and additional shift instructions. A set of hybrid implementations with USL support is created, which require less area than the hybrid implementations with CFP hardware, but offer less throughput.

In this dissertation, the design and implementation of 38 multiply-add, 8 addition/subtraction, 6 multiplication, 45 division, and 45 square root designs are presented. These designs include full software kernels, full hardware modules, hybrid implementations with USL support, and hybrid implementations with CFP hardware. Three different algorithms for division and three for square root are utilized.

Also presented are functionally equivalent FPUs formed from combinations of full software kernels, full hardware modules, hybrid implementations with USL support, and hybrid implementations with CFP hardware to perform unfused multiply-add, along with Newton-Raphson division and square root. The proposed software kernels, hardware modules, and hybrid implementations and FPUs (i.e., the combination of two or more FP software kernels, hardware modules, or hybrid implementations) are evaluated in terms of area, throughput, and instruction count when performing FP multiply-add, addition/subtraction, multiplication, division, and square root.

Additionally, this research dissertation presents a method for reducing overhead for a synthetic aperture radar (SAR) engine. SAR imaging uses pulses of microwave energy transmitted

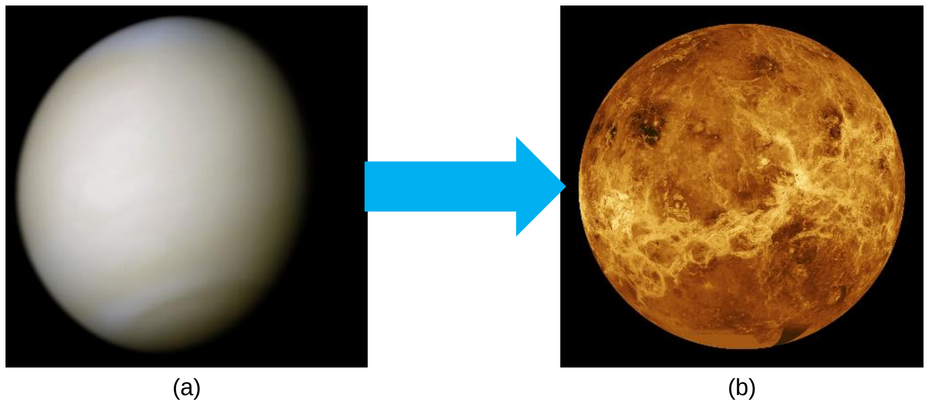


Figure 1.2: Images of Venus produced by NASA probes. (a) imaged by NASA's Mariner 10 and covered in sulfur dioxide clouds and only partially illuminated [21], (b) imaged by Magellan probe using SAR imaging without clouds and fully illuminated [22]. The color hues are simulated and are based on images recorded from the Soviet Venera 13 and 14 spacecraft.

from a series of locations towards a target and reflected back towards an antenna to provide a means for day, night, and all weather imaging while producing resolution that otherwise requires a large antenna aperture [4]. SAR imaging is used in many fields, including environmental monitoring, navigation, reconnaissance, and surface mapping. For example, Figure 1.2 (a) shows an image of Venus taken by NASA's Mariner 10 probe, but the surface is not visible due to the presence of sulfur dioxide clouds and the planet is not fully illuminated. About 25 years later, NASA mapped the surface by imaging through the clouds using SAR on its Magellan probe as shown in Figure 1.2 (b) .

The collection geometry for spotlight-mode SAR is depicted in Figure 1.3. For spotlight-mode SAR, an antenna is typically mounted to an aircraft that flies in a circular path. A sensor steers the antenna beam to continuously illuminate, or "spotlight" the terrain patch being imaged [4]. As data are acquired from more angles, the image resolution is improved.

The focus of this portion of the dissertation is to reduce the size of data words in a SAR backprojection image formation datapath to much smaller widths than the FP double-precision (DP) and single-precision (SP) commonly used in programmable processors as well as in custom hardware. Reducing data word widths directly reduces circuit area, which is easy to measure and thus it is the metric used in this work. Energy dissipation and computational latency are also directly reduced by word width reduction, but unfortunately also depend on factors such as radar data and architectural details and as a result are more difficult to compare.

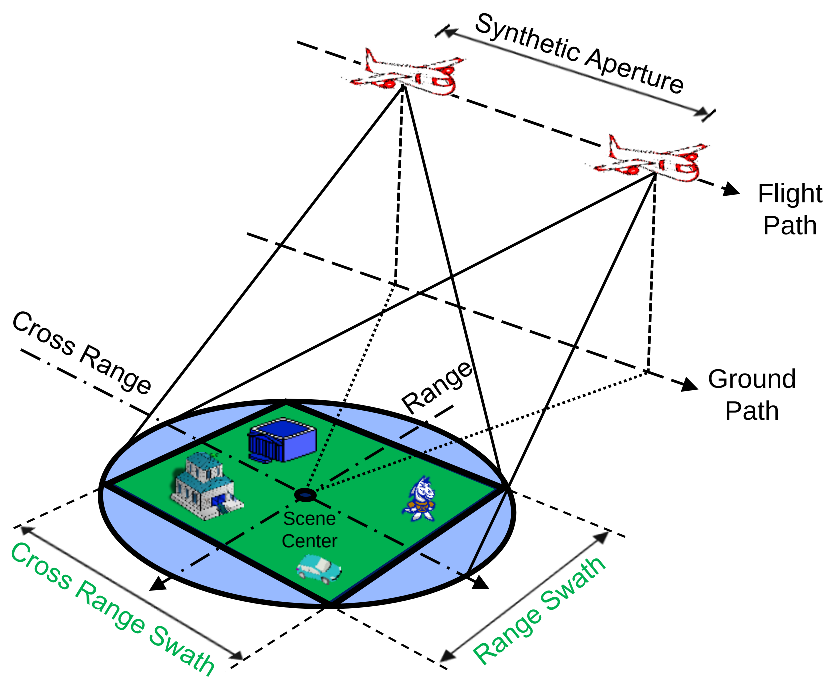


Figure 1.3: Spotlight-mode synthetic aperture radar collection geometry. Adapted from [4].

The circuit area of an application-specific IC (ASIC) is a useful metric to predict the energy dissipation per calculation (among similar designs) and fabrication cost per chip. In addition, comparing ASIC areas among similar designs can also provide a basis for predicting the resource requirements, energy/calculation, and performance of implementations on FPGAs.

The FP hardware requirements of a backprojection algorithm for an airborne spotlight-mode SAR system and the effect that reducing the FP word width has on final image quality are determined. Identifying the necessary FP word width for each of these blocks can improve future ASIC design and algorithm development. The image formation algorithm is first broken into seven functional blocks, then the FP word width for each of these blocks is reduced. SAR images are formed and then assessed against images created using DP-FP arithmetic. The potential width reduction and area savings are determined by observing which FP word widths maintain acceptable image quality.

Figure 1.4 plots the number of processors on a single die versus year. For each chip considered, a processor is defined as being capable of independent program execution. As seen from the graph, the trend over time has been an increase in the number of cores per die. Additionally, modern fabrication technologies now allow for the number of processors to exceed a thousand

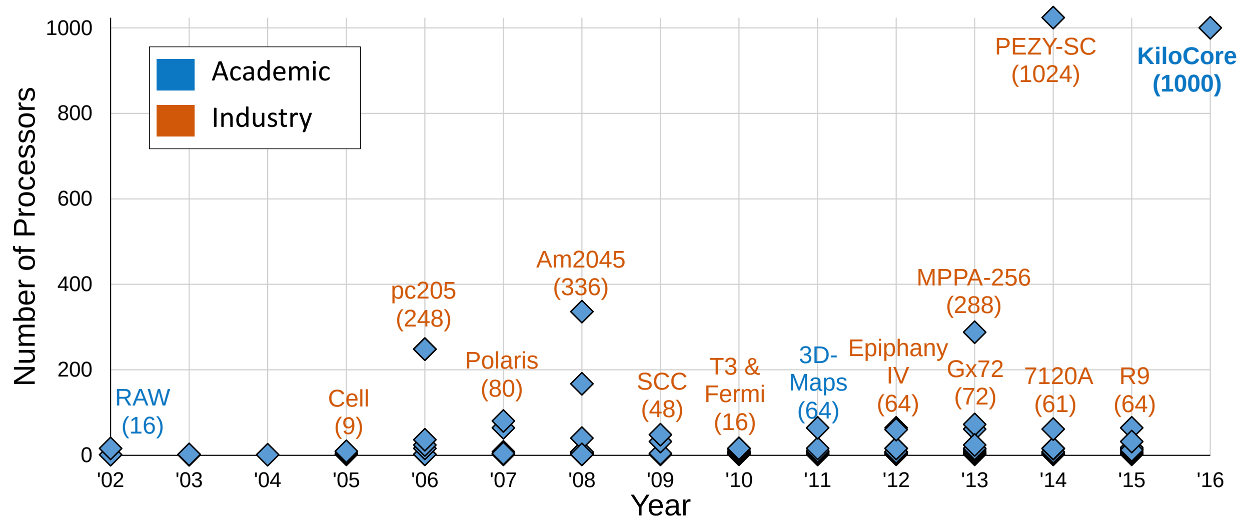


Figure 1.4: Number of processors on a single die versus year. Each processor is capable of independent program execution [6].

per die [23]. However, to enable effective computation in this 1000-processor era the processor architecture, the interconnect, memory interactions, and application design must all be innovative [23, 24].

While most processors contain memory caches as an effective means for storing frequently used data and instructions, maintaining cache coherency and limiting power dissipation are major difficulties when scaling up to 1000s of processors. Implementing simpler cores allows for lower energy per operation and more cores per die. Increasing the number of cores allows for the amount of parallel processing to increase by allowing more independent and concurrent instruction streams. This dissertation presents the design and chip details for two fine-grained many-core chips chips fabricated in 32 nm partially depleted silicon-on-insulator (PD-SOI) complementary metal oxide semiconductor (CMOS) technology, namely KiloCore and KiloCore2. Neither of these designs contain caches and instead utilize local memory, nearby processor memory, or on-chip memory blocks. KiloCore is one of the first chips fabricated containing at least 1000 processors. This dissertation covers the design process from register-transfer level (RTL) to graphic database system (GDS) and the steps involved, and the lessons learned from the tape-out of each of these chips. This section covers the physical design of the oscillators, macro blocks, and chip level portions of the design. Additionally, the steps/considerations made for chip finishing, chip packaging design, inter-processor timing, power planning, and power gating are detailed. The specifications for each chip are provided as well as measured results for KiloCore. Figures of the macro block and chip level layouts are provided. The testing process is also described. KiloCore2 is currently awaiting testing; however, specifications and layout images are provided for this chip.

In 2004, Phillip Colella identified seven numerical methods important for the next decade of scientific and engineering workloads [25], particularly high end simulation in the physical sciences. Inspired by Colella, Asanovic et al. published a list of thirteen common computational “dwarfs” or “motifs”, which represent patterns of computation and communication shared among many applications [26]. Sparse linear algebra, which has applications in general purpose computing, machine learning and graphics, is among these dwarfs. This dwarf involves operating on datasets which are dominated by zero values and spans a series of possible workloads; however, this dissertation focuses on sparse matrix dense vector multiplication. Cached-based architectures are ill-suited with the irregular data accesses of this workload [27], and contrasting with traditional architectures, KiloCore processors do not contain caches [9]. Similarly, Asanovic et al. stated the goal should be 1000s of cores per chip, each of which is most efficient in terms of throughput per watt and throughput per area.

For this section, the benefits of performing sparse matrix times dense vector multiplication (SpMV) on a fine-grained many-core platform are explored. Twenty-three implementations for performing SpMV using single-precision floating-point are created for a fine-grained many core platform with simple processors and FP support. These implementations are compared to two CPUs (Intel Core i7-3770 and Core i7-2630QM) and two GPUs (NVIDIA GeForce GT 620 and NVS 4200M). For implementations running on the fine-grained processors, the sparse matrix is streamed in, resulting in lower memory requirements than traditional implementations. These implementations for the fine-grained processors are also scalable with the length of the dense vector. The throughput per watt and throughput per area are determined for each implementation operating on five different sparse matrices from distinct problem types.

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows. The first part of Chapter 2 discusses related work on reducing the overhead for FP computation. The FP format, the algorithms used for performing FP operations, and the targeted architectures are then covered. Chapter 2 also discusses related work on SAR imaging and the data sets used for analyzing image quality when modifying FP word width. Background on the design of the KiloCore and KiloCore2 chips is then presented. The last part of this chapter covers related work on scientific kernels and the data sets. Chapters 3 and 4 present the work done on reducing FP hardware overhead. In Chapter 3, the software kernels, full hardware modules, and hybrid implementations are explained. The FP kernels, modules, and implementations are first compared against each other. FPUs formed from combinations of FP addition/subtraction and multiplication units are then evaluated. Chapter 4 presents the functional blocks for the SAR backprojection algorithm, and the methods used for reducing the FP word width and determining chip area. Finally the chapter concludes by evaluating the effect of FP word width reduction on image quality and area. Chapter 5 presents the design of two many-core chips fabricated in 32 nm PD-SOI CMOS, KiloCore and KiloCore2. In Chapter 6, various methods for performing SpMV are implemented for a many-core platform and considered against CPU and GPU SpMV implementations. Finally, Chapter 7 summarizes this dissertation and future work.

Chapter 2

Background

2.1 Related Work on Floating-Point Architectures

Several approaches have been explored for increasing FP throughput and maintaining low area overhead. Fused and cascade multiply-add FPUs improve accuracy and provide computational speedup [28, 29]; however, they introduce large area [30] and power overhead, which are undesirable for simple fixed-point processors. If blocks of data have similar magnitudes, block FP (BFP) can be useful for increasing signal-to-noise ratio (SNR) and dynamic range [31]. In block FP, several values share the same exponent as shown in Figure 2.1. However, if the data do not have similar magnitudes this can introduce roundoff errors. Micro-operations have been used to create a virtual FPU, which reuse existing fixed-point hardware to emulate a FP datapath for a very long instruction word (VLIW) processor [32]. Hardware prescaling and postscaling has also been used to reduce the required hardware for FP division and square root [33]. The hardware overhead can be reduced by shortening the exponent and mantissa widths for video coding [34], audio applications [35], and radar image formation [36]. Some speech recognition and image processing applications have been shown to not require the full mantissa width, and the specific requirements of these applications can be determined through accuracy analysis [37]. Additionally, approximation frameworks for convolutional neural networks can utilize reduced FP word widths [38]. Some chips with FPUs are not compliant to the IEEE-754 standard. An example is the CELL processor, which improves multimedia application performance by removing rounding, exceptions, and denormalized number support [39]. CFP instructions have also been explored for an FPGA to increase FP throughput with

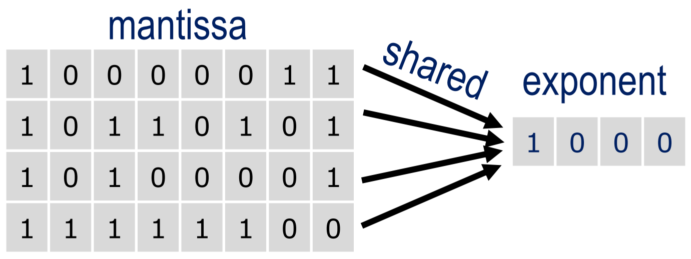


Figure 2.1: Example of Block FP where four values share the same exponent. This example uses a FP format with an 8-bit mantissa and 4-bit exponent.

lower area overhead than a full hardware FPU [40]. However, the authors did not consider modular FPUs, nor the throughput when performing the multiply-add operation, nor did they explore the area and throughput trade-offs of various division and square root algorithms. Additionally, the authors focused on adding FP support to a Nios II softcore processor implemented on an FPGA, rather than on a simple 16-bit datapath fixed-point processor.

2.2 Floating-Point Computation Background

This section provides background on the work done in Chapter 3 on Hybrid Floating-Point Implementations.

2.2.1 Floating-Point Format

The IEEE-754 single-precision format is used for all FP arithmetic, with values on the normalized value interval $\pm[2^{-126}, (2-2^{-23}) \times 2^{127}]$ [41]. An example of a 32-bit single-precision FP number is shown in Figure 2.2. The most significant bit (MSB) indicates whether the number is positive or negative. This is followed by an eight bit exponent, which is biased by 127. Lastly, the mantissa is normalized, so the leading one bit is implicitly stored. The IEEE-754 default rounding mode, round to nearest even is supported for all FP arithmetic in this dissertation. Additionally, round toward zero is supported. However, in order to reduce overhead and because they are often not needed, the following features are not supported: exception handling, NaN (Not a Number), ±infinity, denormalized values, and alternative rounding modes. Many applications, such as some multimedia and graphics processing, do not rely on all elements of the standard [39, 42]. Therefore,

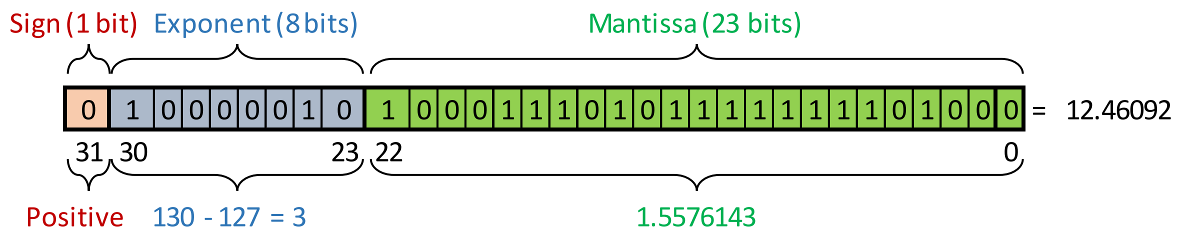


Figure 2.2: Example IEEE-754 single-precision number.

the FP implementations presented are targeted for similar workloads.

2.2.2 Floating-Point Arithmetic

2.2.2.1 Addition/Subtraction

This operation begins by determining the smaller magnitude operand, aligning the mantissas of the two operands based upon the difference in their exponents, and adding or subtracting the mantissas based on the desired operation and the signs of the operands. The initial exponent is set to the exponent of the larger magnitude operand. The result is then normalized and rounded. The sign bit is determined by the larger input operand.

An example FP addition is shown in Figure 2.3. Two values to be added are shown at the top in binary, hexadecimal, and decimal format. The final result is shown at the bottom in decimal format. Both values are positive so the result is positive. The smaller number's mantissa is shifted left by four bit positions for mantissa alignment, since this is the exponent difference. After adding the mantissas, no shift is needed for normalization and all of the rounding bits are zero, so the exponent and mantissa are not adjusted.

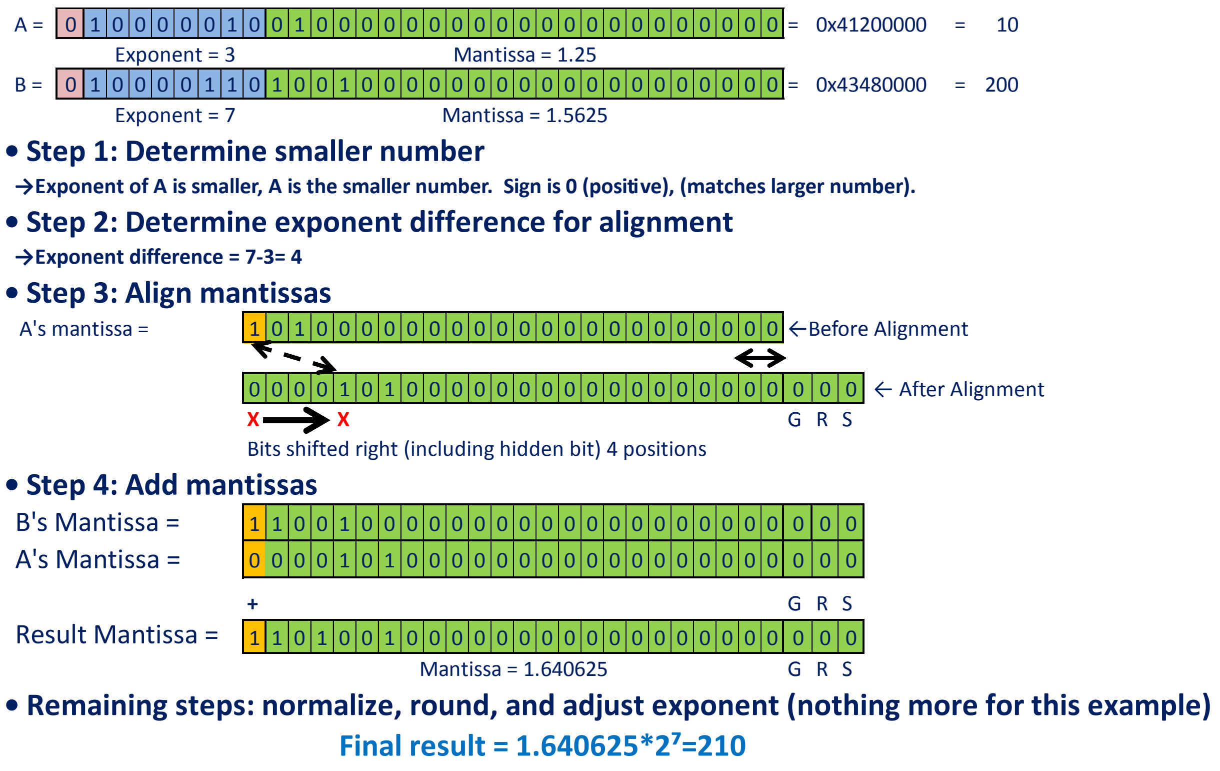
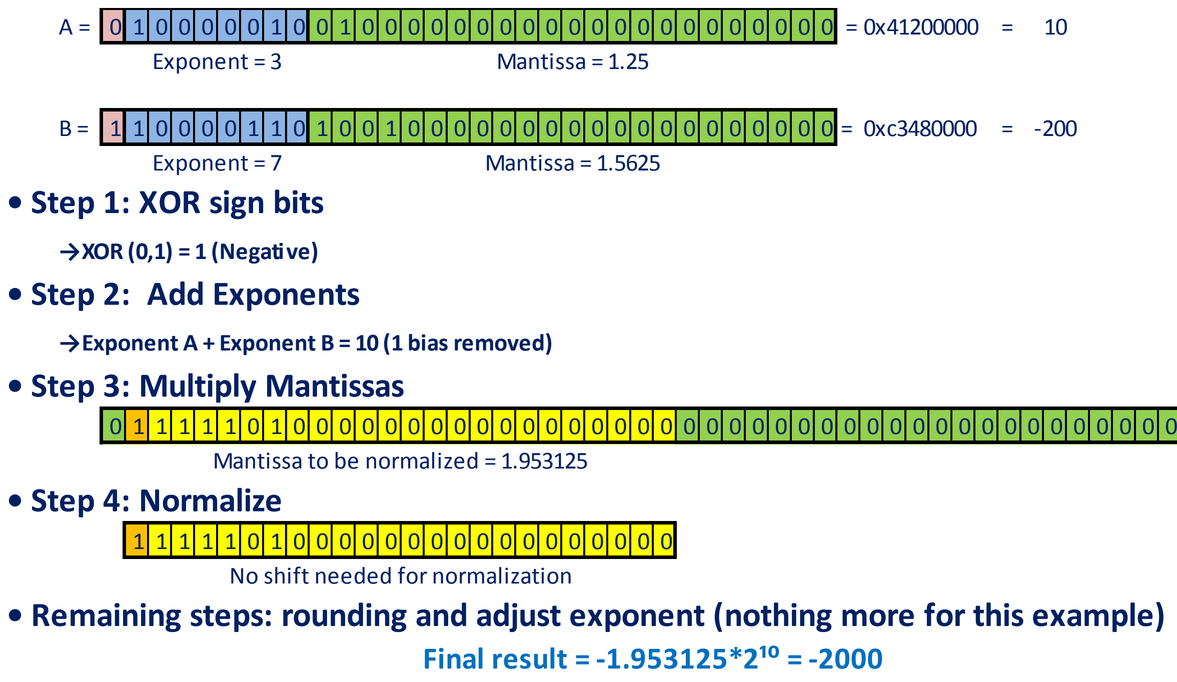

A = 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 = 0x41200000 = 10
Exponent = 3
Mantissa = 1.25
B = 0 1 0 0 0 0 1 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 = 0x43480000 = 200
Exponent = 7
Mantissa = 1.5625
• Step 1: Determine smaller number
→Exponent of A is smaller, A is the smaller number. Sign is 0 (positive), (matches larger number).
• Step 2: Determine exponent difference for alignment
→Exponent difference = 7-3= 4
• Step 3: Align mantissas
A's mantissa =
←Before Alignment
← After Alignment
G R S
Bits shifted right (including hidden bit) 4 positions
• Step 4: Add mantissas
B's Mantissa =
A's Mantissa =
+
Result Mantissa =
Mantissa = 1.640625
• Remaining steps: normalize, round, and adjust exponent (nothing more for this example)
Final result = 1.640625*2^7=210

Figure 2.3: Example showing the addition of two IEEE-754 single-precision FP values.

2.2.2.2 Multiplication

This operation begins by multiplying the mantissas together. The initial exponent is set by adding the operand exponents, the product is then normalized and rounded, and the sign bit is set by XORing the sign bit of both operands together.

A FP multiplication example is shown in Figure 2.4. Two values to be multiplied are shown at the top in binary, hexadecimal, and decimal format. The final result is shown at the bottom in decimal format. The XOR of the sign bits indicates that the result will be negative. Adding the exponents shows that the result exponent will have a value of 10. After multiplying the mantissas, no shift is needed for normalization and all rounding bits are zero, therefore the exponent and mantissa are not adjusted.



A = 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 = 0x41200000 = 10
Exponent = 3
Mantissa = 1.25
B = 1 1 0 0 0 0 1 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 = 0xc3480000 = -200
Exponent = 7
Mantissa = 1.5625

- **Step 1: XOR sign bits**
  - **→XOR (0,1) = 1 (Negative)**
- **Step 2: Add Exponents**
  - **→Exponent A + Exponent B = 10 (1 bias removed)**
- **Step 3: Multiply Mantissas**

0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Mantissa to be normalized = 1.953125

- **Step 4: Normalize**

1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
No shift needed for normalization

- **Remaining steps: rounding and adjust exponent (nothing more for this example)**

Final result = -1.953125*2^{10} = -2000

Figure 2.4: Example showing the multiplication of two IEEE-754 single-precision FP values.

2.2.2.3 Multiply-Add

The multiply-add operation performs $a + b \times c$. The unfused multiply-add first calculates $b \times c$, rounds the result, adds the rounded product to the addend a, then performs a second rounding. The fused multiply-add (FMA) rounds once, after the product is added to the addend.

2.2.2.4 Division

Three algorithms are implemented for division: long-division [43], non-restoring [44], and Newton-Raphson [1]. Division is typically an infrequent operation [45, 46], so little area should be allocated for this operation. The long-division and non-restoring algorithms are chosen for their simplicity and low area impact, whereas the Newton-Raphson algorithm is selected for its potentially high throughput [47]. Given that division takes longer to compute than other operations and can be complex, the Cray T3E Fortran Optimization guide states, “The best strategy for division is to avoid it whenever possible” [48].

The long-division algorithm first compares the magnitude of the divisor and the dividend. If the divisor is less than or equal to the dividend, then it is subtracted from the dividend to form a partial remainder, and a 1 is right-shifted in as the next bit of the quotient. Otherwise, they are not subtracted and a 0 is shifted in for the next bit of the quotient [43]. The partial remainder is then left-shifted by one bit, and set as the new dividend. This process continues until all of the quotient bits are determined. The result is then normalized and rounded. The result exponent is calculated by subtracting the input exponents and adding back the bias.

The non-restoring division algorithm is similar to the restoring algorithm except that it avoids the restoring step for each loop iteration to improve performance [49]. The divisor's mantissa is first subtracted from the dividend's mantissa. A loop is executed that first checks if the result is negative or positive, then left-shifts the quotient and the result. If the result is negative, the dividend is added to the result. If the result is positive, the least significant bit (LSB) of the quotient is set to 1 and the dividend is subtracted from the result [50]. This loop iterates until all bits are determined for the quotient [44]. The final step then restores the divisor if the result was negative. The result is then normalized and rounded. The result exponent is calculated in the same manner as long-division.

For the Newton-Raphson division algorithm, the reciprocal of the divisor is determined iteratively and then multiplied by the dividend [1]. The divisor and dividend are first scaled down to a small interval. A linear approximation is then used to estimate the reciprocal and minimize the maximum relative error of the final result [51]. This estimation is then improved iteratively. Once this reciprocal is determined, it is multiplied by the scaled dividend to obtain the result, which is then refined by computing residuals at a higher precision [43].

These division algorithms calculate the result's sign by XORing the sign bits of both operands.

2.2.2.5 Square Root

Three algorithms are used for performing the square root operation: digit-by-digit, non-restoring [52], and Newton-Raphson [1]. Similar to division, square root is typically an infrequent operation [45, 46], therefore little area should be allocated. The digit-by-digit and non-restoring algorithms are chosen for their low area impact, while the Newton-Raphson method is chosen for providing high throughput since the algorithm converges quadratically rather than linearly [53].

The digit-by-digit square root algorithm first determines the result exponent. If the unbiased exponent is odd, one is subtracted to make it even and the radicand mantissa is left-shifted to account for the change without a loss of precision. The exponent is then right-shifted by one bit. Solving for the root mantissa then begins by setting the MSB of the root to one, squaring the root, and subtracting it from the radicand. If the result is negative, the bit set to one is changed to zero, otherwise it is left as a one. The next MSB of the root is then set to 1, and the process is continued until all of the root bits are determined. The squaring step is unnecessary for the first iteration of this loop. The result is then normalized and rounded.

The non-restoring square root algorithm involves a loop where each iteration calculates one digit of the square root exactly and the digits are based upon whether the partial remainder is negative or positive [52]. The result exponent is determined by dividing the original exponent by two and adding 63, which is half the bias rounded down. The LSB of the original exponent is then added to this sum.

The Newton-Raphson square root algorithm finds the reciprocal of the square root first by using an algorithm similar to Newton-Raphson division [1]: scaling the input, applying the linear approximation [54], and iterating to improve the approximation. The result is determined by multiplying the reciprocal approximation by the original input, and corrected via the Tuckerman test [55].

2.3 Background on Many-Core Chip Design

Costs for modern CMOS processors, which are several million dollars for fabrication and tens of millions of dollars for design, are expected to continue to rise. Therefore, chips that are not tailored to a specific application are increasingly attractive. ASICs can provide the highest performance and energy efficiency for an application, but lack flexibility. Programmable digital signal processors (DSPs) are easily programmable but provide much lower performance and energy efficiency [15]. FPGAs, which contain programmable logic blocks with reconfigurable interconnect, lie in the middle of these two. Simple programmable processors aim to provide both high energy efficiency and performance, while offering the flexibility of being reprogrammable. Previous examples of such platforms that have been fabricated include the Asynchronous Array of Simple Processors (AsAP) chips, AsAP [56] and AsAP2 [7]. The many-core processing arrays presented in this work also offer both high performance and energy efficiency, and are reconfigurable and reprogrammable.

Caches, which store frequently used instructions and data, are present in most modern processors. However, they pose several issues when processors scale into the 100s or 1000s, primarily with cache-coherency and power dissipation [57, 58]. KiloCore and KiloCore2 do not contain caches, but instead store data and instructions in small memories present in each core. Neighboring processors, on-chip memory, or off-chip memory can also be used if the data or instruction requirements of a particular application are large. In addition to not containing caches, processors contain their own local oscillator rather than a phase-locked loop (PLL), and are clocked using a globally asynchronous locally synchronous (GALS) clocking scheme. As the number of processors per die continues to increase, it is becoming increasingly important to shut down inactive silicon. Therefore, processors may be halted when they have no work to do and thus dissipate only leakage power. KiloCore2, another processor design covered in this dissertation, includes dynamic voltage frequency scaling (DVFS) circuitry to power gate processors to further reduce power dissipation.

2.4 Targeted Architectures for Hybrid FP Implementations

Several methods for performing FP operations on a fixed-point datapath are evaluated on the asynchronous array of simple processors architecture (AsAP2), shown in Figure 2.5. However, the work presented in Chapter 3 applies to any fixed-point architecture. AsAP2 is an example of a fine-grained many-core system with a fixed-point datapath [59], and features 164 simple independently-clocked homogeneous programmable processors. Each processor occupies 0.17 mm^2 in 65 nm CMOS technology, and can operate up to a maximum clock frequency of 1.2 GHz at 1.3 V [7]. Processors support 63 general-purpose instructions, contain a 128x35-bit instruction memory and a 128x16-bit data memory, and implement a 16-bit fixed-point signed-only datapath including a MAC unit with a 40-bit accumulator. The platform is capable of executing a wide range of applications including audio and video processing [60, 61], an 802.11a baseband receiver [62], the advanced encryption standard engine [63, 64], as well as ultrasound image processing [65].

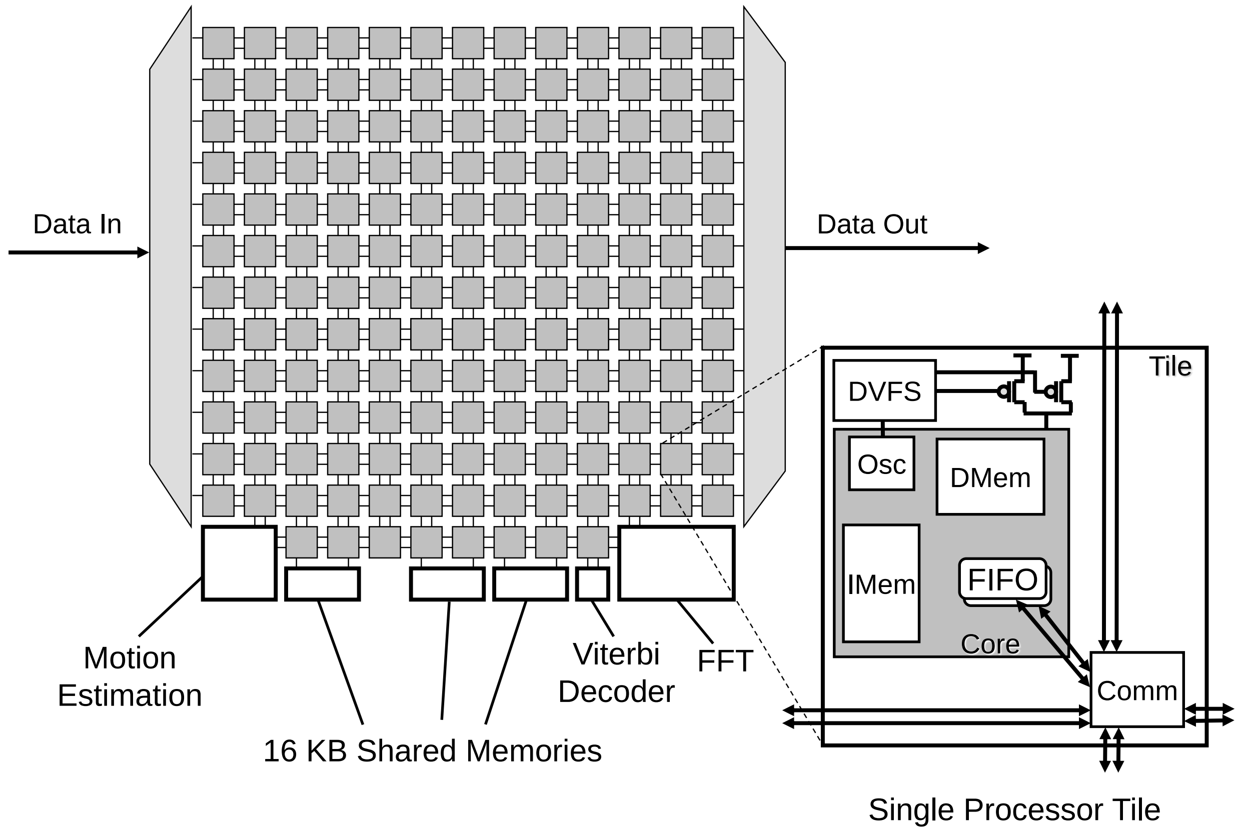


Figure 2.5: Block diagram of fine-grained many-core AsAP2 targeted architecture [7].

2.5 Related Work on Synthetic Aperture Radar Imaging

The SAR backprojection algorithm is a widely used, compute intensive method for forming images, and is known for its inherently parallel nature [66, 67]. With the shift towards many-core processing-arrays and interest in energy-efficiency, algorithms have been studied to parallelize SAR backprojection algorithms for a GPU [68], an Intel Xeon Phi many-core accelerator [5], and a TI DSP [69]. Additionally, FPGAs have been considered as image processing platforms for real-time SAR processing [70]. Previously, SAR data compression followed by transmission to remote computation nodes has been necessary [71, 72]. However, performing computation on an embedded system can remove the need for a raw-data link if performed in an energy-efficient fashion.

FP arithmetic is typically chosen for performing backprojection computations due to the dynamic range and precision needs of the raw data [5, 73]. Previous work has been done on optimal design trade-offs of general FP hardware [19, 74]. However, little exploration has been done into reducing FP mantissa and exponent width and observing the effect this has on SAR image quality and chip area.

2.6 Synthetic Aperture Radar Data Sets

Three publicly available data sets released by the Air Force Research Laboratory (AFRL) are utilized for image formation [75, 76]. The phase history data for all three data sets are in the X-band region with a circular flight path and collected via spotlight passes.

2.6.1 Volumetric SAR Data Set

This data set is formed from imaging stationary civilian vehicles and calibration targets [75]. For each azimuth angle, there are 117 pulses on average, each with 424 frequency samples. Images formed from this data set have a scene extent of 100 m $\times$ 100 m and a 501 $\times$ 501 pixel image.

2.6.2 Point Target Data Set

This data set consists of synthetically generated data for three point targets [77]. The targets were simulated with 128 pulses and 512 frequency samples per pulse. Images formed from this data set have a scene extent of 10 m $\times$ 10 m and a 501 $\times$ 501 pixel image.

2.6.3 Ground Moving Target Indicator Data Set

The SAR-based Ground Moving Target Indicator (GMTI) motion compensated radar data set includes data from imaging a moving vehicle in an urban environment [76]. The data includes 8000 pulses and 384 frequency samples per pulse. Images formed from this data set have a scene extent of 200 m $\times$ 200 m and a 1001 $\times$ 1001 pixel image.

2.7 Related Work on Scientific Kernels

SpMV is one of the most common kernels present in data mining, signal processing, and image processing. The basic linear algebra subroutines (BLAS), which are used for performing common linear algebra operations, include SpMV. The underlying infrastructure of complex graph analysis tools such as the Knowledge Discovery Toolbox [78] and Pegasus [79], utilizes computational kernels such as SpMV as well. To accelerate these routines in hardware, FPGAS have been used, which found that the bottlenecks lie occur when transferring data between host and the FPGA memory [80]. To improve both power and area efficiency, an ASIC was fabricated in 40 nm CMOS which performs sparse BLAS operations using the compressed sparse column (CSC) format and accumulates partial products for multiple rows [81]. In order to speed up the performance of SpMV operations, alternative data formats have been considered; for example, the Sliced COO method [82], a unified format for processors with wide "single instruction, multiple data" (SIMD) units [83], and CSR5 [84], a format targeted towards CPUs, GPUs and the Xeon Phi processor. SpMV performance modeling and exploration of various data formats has also been performed for the Cell SPE [85]. These traditional cached-based architectures are ill-suited with the irregular data accesses of SpMV [27].

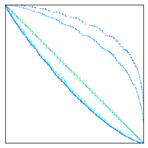
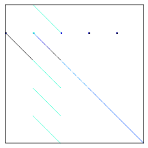
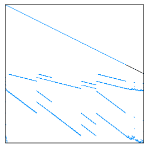
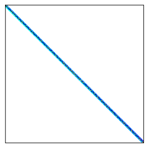
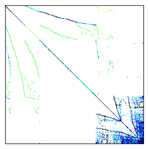
2.8 Sparse Matrix Data Sets

Sparse matrices are obtained from the SuiteSparse Matrix Collection, a constantly updated collection of sparse matrices from a wide range of domains, including computational fluid dynamics, semiconductor device simulations, computer graphics/vision, circuit simulation, economic modeling, and thermodynamics [86]. The sparse real unsymmetrical matrices used to evaluate SpMV performance are shown in Table 2.1. The sparsity pattern figures are created using *cspy* [87]. All matrix and vector data is in single-precision 32-bit IEEE-754 format. The median N value for the real unsymmetrical matrices in the entire database is 2339, while the average density is 0.01. This work considers matrices with an N value less than and greater than the median N value of the database matrices.

2.9 Matrix Libraries

To ensure a fair comparison of performance evaluation results, well known and freely available matrix libraries were used, specifically Eigen [88] and cuSPARSE from NVIDIA [89]. Eigen is a general C++ library for linear algebra algorithms and is used on the CPUs, while cuSPARSE

Table 2.1: Sparse Matrices Used for Evaluating SpMV Performance.

Sparsity Pattern	Name	Problem Description	N	NNZ
	HB-gre_1107	Directed Weighted Graph	1107	5664
	Bai-tols2000	Computational Fluid Dynamics	2000	5184
	Hamrle-Hamrle2	Circuit Simulation	5952	22162
	Averous-epb1	Thermal	14734	95053
	Rommes-descriptor_xingo6u	Eigenvalue/Model Reduction	20738	73916

N = Number of rows or columns. Matrices are square.
NNZ = Number of nonzero elements.

provides a set of basic linear algebra subroutines for performing specifically sparse matrix operations used on the GPUs.

2.9.1 Eigen

Eigen is a C++ template library for linear algebra. Eigen is also used in many projects including Google's TensorFlow open source software library for machine learning [90], and Pteros, a C++ library for molecular modeling [91]. Eigen claims to offer comparable and sometimes faster speed than the most optimized BLAS implementations, including the Intel MKL (Math Kernel Library) [88]

2.9.2 cuSPARSE

The cuSPARSE library is implemented on top of NVIDIA's Compute Unified Device Architecture (CUDA) and can be called from C and C++ [89]. CUDA allows general purpose processing on GPUs, and the cuSPARSE library routines are used to perform the SpMV operation using a GPU. Additionally, NVIDIA claims cuSPARSE provides higher performance than Intel MKL when performing sparse matrix operations [92].

Chapter 3

Hybrid Hardware/Software FP Implementations

When area cannot be increased, software implementations are the only option for performing FP arithmetic. Dedicated hardware designs are ideal when the goal is maximum throughput. When area is constrained, hybrid designs are optimal because they increase throughput and require less area than dedicated FP hardware. They provide a method for satisfying an area constraint that dedicated hardware would violate. Hybrid implementations w/ USL support increase throughput and reduce area overhead by adding functionality to existing hardware to simplify multi-word operations. The hybrid implementations w/ CFP exceed the performance of the USL support designs by adding custom hardware which performs specific steps of a FP operation. These steps would otherwise require many fixed-point instructions.

The full software implementations require many operations on large (multi-word) data values. Multi-word operations require carrying between words, or summing and carrying between partial products. The programmer must avoid using the bit that is treated as signed (bit 16 for the target platform), and must handle carry flags and partial product summation in software. Therefore, signed hardware cannot operate on completely utilized 16-bit words. Words must be partitioned into 15-bits each at most.

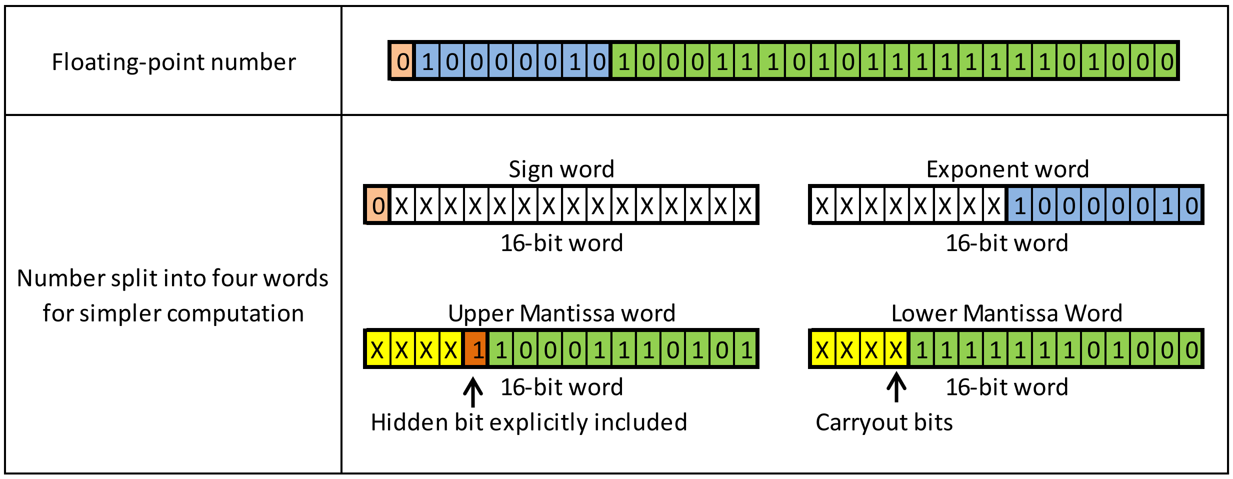

Floating-point number
0 1 0 0 0 0 0 1 0 1 0 0 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 0 0 0
Number split into four words for simpler computation
Sign word
0 X X X X X X X X X X X X X X X
16-bit word
Exponent word
X X X X X X X X 1 0 0 0 0 0 1 0
16-bit word
Upper Mantissa word
X X X X 1 1 0 0 0 1 1 1 0 1 0 1
16-bit word
Hidden bit explicitly included
Lower Mantissa Word
X X X X 1 1 1 1 1 1 1 0 1 0 0 0
16-bit word
Carryout bits

Figure 3.1: FP values split into four words to simplify software computation. The hidden bit is explicitly included in the upper mantissa word. The yellow boxes indicate bit locations available for carryout bits that may arise during intermediate computations.

3.1 Full Software Kernels

The full software kernels are coded in AsAP instructions and form a software library consisting of addition/subtraction, multiplication, division, and square root. They are referred to "full software" because they utilize only general purpose fixed-point instructions and no custom FP instructions. Since the word size of the platform is 16-bits, each 32-bit FP value is received on-chip as two words. To simplify software computation, these two words are split into four words as shown in Figure 3.1 to store the following: the sign bit, exponent, high mantissa bits and low mantissa bits.

The total area for each software kernel is determined by the number of processors required to implement each FP operation. Since these kernels use only the platform's existing fixed-point datapath, they do not add additional area.

The programs for these kernels are large due to the lack of unsigned ALU instructions and the number of fixed-point instructions required for emulating FP hardware. Computation time for software FP consists primarily of operand comparisons, mantissa alignment, addition, normalization, and rounding. Each full software kernel is discussed below.

3.1.1 Addition/Subtraction Kernel (Full SW Add/Sub)

Since 222 instructions are required for this kernel and each processing core has an instruction memory that can only store 128 instructions, two processors are needed for sufficient instruction memory. The first processor sorts the operands and aligns the mantissas. The second processor adds the mantissas, normalizes and rounds.

3.1.2 Multiplication Kernel (Full SW Mult)

Most of the instruction overhead for this kernel is from performing mantissa multiplication and rounding. The partial products of the multiplication are created and added using the MAC, and aligned using the shifter.

3.1.3 Division Kernel Version 1 (Full SW Div Ver. 1)

This kernel uses the long-division algorithm [43]. The loop to determine the quotient requires the greatest number of instructions, and involves several shift and subtraction operations.

3.1.4 Division Kernel Version 2 (Full SW Div Ver. 2)

This kernel uses the Newton-Raphson algorithm [1]. The kernel begins with zero input detection and handling, followed by exponent calculation. The input is then prepared for later calculations. The initial estimate of the reciprocal is calculated, followed by Newton-Raphson iterations. The first input is then multiplied by the reciprocal of the second, and then the result is normalized and rounded. Lastly, the LSB is corrected.

3.1.5 Square Root Kernel Version 1 (Full SW Sqrt Ver. 1)

This kernel uses the digit-by-digit method. Most of the overhead involves squaring each value being tested.

3.1.6 Square Root Kernel Version 2 (Full SW Sqrt Ver. 2)

This kernel uses the Newton-Raphson method, similar to *Full SW Div Ver. 2*, except the first input is multiplied by the reciprocal of the square root. Most of the instruction overhead is from handling multi-word values.

3.2 Full Hardware Modules

Full hardware modules offer the highest throughput, but require the largest area overhead of the FP designs implemented. These modules are referred to as "full hardware" because all arithmetic is performed on dedicated FP hardware, without using any fixed-point instructions for computation. Since the target platform has a 16-bit datapath, the FP values are first loaded into a set of FP registers, one word at a time. Each value is stored as two 16-bit words as shown in Figure 3.2. An entire FP operation is carried out by a single FP instruction and the results are read from the FP registers, 16-bits at a time. The instructions used in each module are shown in Table 3.1.

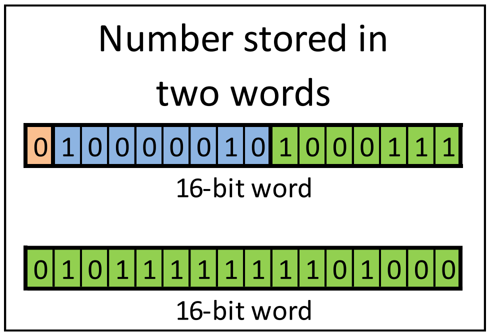


Figure 3.2: FP values split into two words for hardware computation. The upper word holds the sign bit, the exponent, and the seven MSBs of the mantissa. The lower word holds the 16 LSBs of the mantissa.

For comparison purposes, a separate version of each full hardware module is created where the word size and datapath are 32-bits. The source operands and the destination are specified in a single instruction. Most of the 32-bit full hardware modules require less area than the 16-bit datapath modules; however, this comparison doesn't consider the total core area for a 32-bit datapath and word size. The full hardware modules are discussed below.

3.2.1 Fused Multiply-Add Module (Full HW FMA)

The full hardware FMA module uses the *FMA* instruction, with a two-cycle execution latency. The design of the module is shown in Figure 3.3 and matches that of a traditional single-path

Table 3.1: Instructions Used by Each FP Design.

	FP Design	Additional Instructions Used
Full Software Modules	*Full SW Add/Sub*	*None*
	Full SW Mult	*None*
	Full SW Div Ver. 1	*None*
	Full SW Div Ver. 2	*None*
	Full SW Sqrt Ver. 1	*None*
	Full SW Sqrt Ver. 2	*None*
Full Hardware Modules	*Full HW Add/Sub*	*FPAdd, FPSub*
	Full HW Add/Sub (32-bit I/O)	*FPAdd32, FPSub32*
	Full HW Mult	*FPMult*
	Full HW Mult (32-bit I/O)	*FPMult32*
	Full HW Div	*FPDiv*
	Full HW Div (32-bit I/O)	*FPDiv32*
	Full HW Sqrt	*FPSqrt*
	Full HW Sqrt (32-bit I/O)	*FPSqrt32*
	Full HW FMA	*FMA*
	Full HW FMA (32-bit I/O)	*FMA32*
Hybrid Implementations with USL	*Hybrid Add/Sub w/ USL*	*ADDU, ADDUC, LZD, SHLC, SHRC, SUBU, SUBUC*
	Hybrid Mult w/ USL	*ADDU, ADDUC, MACCUL, MACUL, MACUH, MULTUL, SHLC*
	Hybrid Div w/ USL Ver. 1	*ADDU, ADDUC, SHLC, SUBU, SUBUC*
	Hybrid Div w/ USL Ver. 2	*ACCSHU, ADDU, ADDUC, MACCUL, MACUH, MACUL, SHRC, SHLC, SUBU, SUBUC*
	Hybrid Sqrt w/ USL Ver. 1	*ACCSHU, ADDU, ADDUC, MACCUL, MACUH, MACUL, SHRC, SHLC, SUBU, SUBUC*
	Hybrid Sqrt w/ USL Ver. 2	*ACCSHU, ADDU, ADDUC, MACCUL, MACUL, SHRC, SHLC, SUBU, SUBUC*
Hybrid Implementations with CFP	*Hybrid Add/Sub w/ CFP Ver. 1*	*BShiftL, FPAdd_SatAlign, FPAdd_Round, LZD*
	Hybrid Add/Sub w/ CFP Ver. 2	*BShiftL, FPAdd_AlignSmall, FPAdd_Round, LZD*
	Hybrid Add/Sub w/ CFP Ver. 3	*BShiftL, FPAdd_Align, FPAdd_Compare, FPAdd_Round, LZD*
	Hybrid Add/Sub w/ CFP Ver. 4	*Shift_LZA, FPAdd_Align, FPAdd_Round*
	Hybrid Mult w/ CFP Ver. 1	*FPMult_NormRndCarry*
	Hybrid Mult w/ CFP Ver. 2	*FPMult_NormRnd*
	Hybrid Div w/ CFP Ver. 1	*FPDiv_LoopExpAdj*
	Hybrid Sqrt w/ CFP Ver. 1	*FPSqrt_Loop*

FMA architecture, similar to the FMA in the IBM RS/6000 [1, 93]. The addend is complemented if effective subtraction is performed, and right shifted by the exponent difference. The multiplier uses radix-4 Booth encoding with reduced sign extension, limiting the widths of the partial products to 28 and 29 bits. The partial products are then compressed using a Wallace tree into carry-save format. A 3:2 carry save adder then adds these values and the lower 48 bits of the shifted addend. An end-around carry adder with a carry lookahead adder computes the sum. In parallel, a leading-zeros anticipator (LZA) determines the number of leading zeros for the result, to within 1 place [94, 95]. The result is complemented if the addend is larger than the product. The result is normalized using the LZA count, followed by a possible 1 bit correction and rounding.

Full HW FMA (32-bit I/O) is created for a 32-bit datapath and word size, and uses *FMA32,* with a two-cycle execution latency. This instruction uses three source operands.

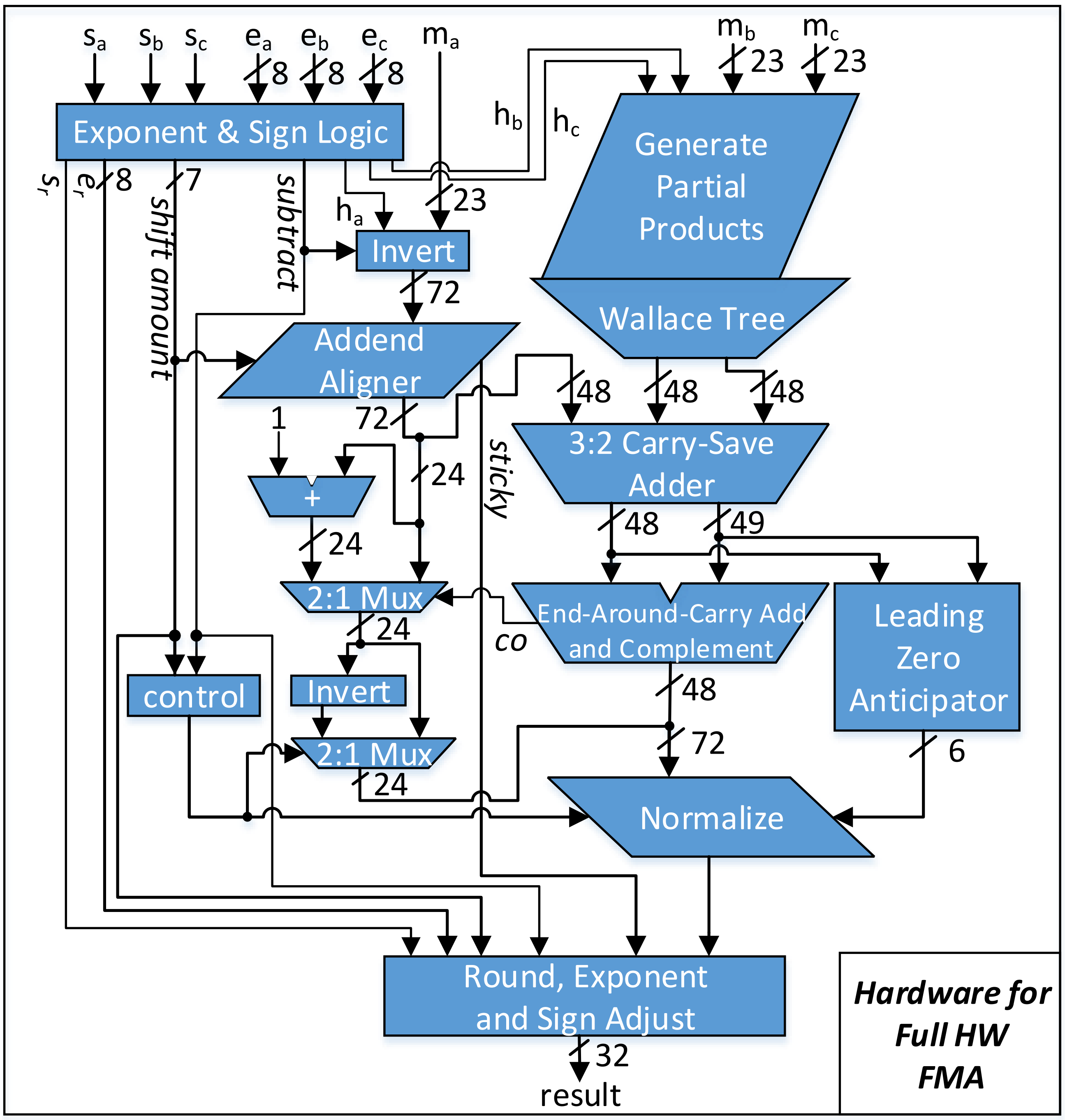


Figure 3.3: Block diagram of the single-precision FMA module. The sign bit of input operands a, b, and c are designated by s_a, s_b, and s_c. The exponents are labeled e_a, e_b, and e_c. The hidden bits of a, b, and c are designated by h_a, h_b, and h_c, respectively, and set to 1 if the corresponding exponent is nonzero. The mantissa bits are labeled as m_a, m_b, and m_c.

3.2.2 Addition/Subtraction Module (Full HW Add/Sub)

The full hardware addition/subtraction module uses the *FPAdd* and *FPSub* instructions with a two-cycle execution latency each to perform addition and subtraction, respectively.

A separate version of this module is created for a 32-bit datapath and word size. *Full HW Add/Sub (32-bit I/O)* uses the *FPAdd32* and *FPSub32* instructions to perform addition and subtraction, each of which has a single-cycle execution latency. If operands are read from a processor's local memory, then a single instruction can perform addition/subtraction.

3.2.3 Multiplication Module (Full HW Mult)

This module uses the *FPMult* instruction with a single-cycle execution latency to perform multiplication. A separate version of this module is created for a 32-bit datapath and word size. *Full HW Mult (32-bit I/O)* uses the *FPMult32* instruction to perform multiplies with a single-cycle execution latency. Assuming operands are read from a processor's local memory, then a single instruction can perform multiplication. The datapath for this module is shown in Figure 3.4.

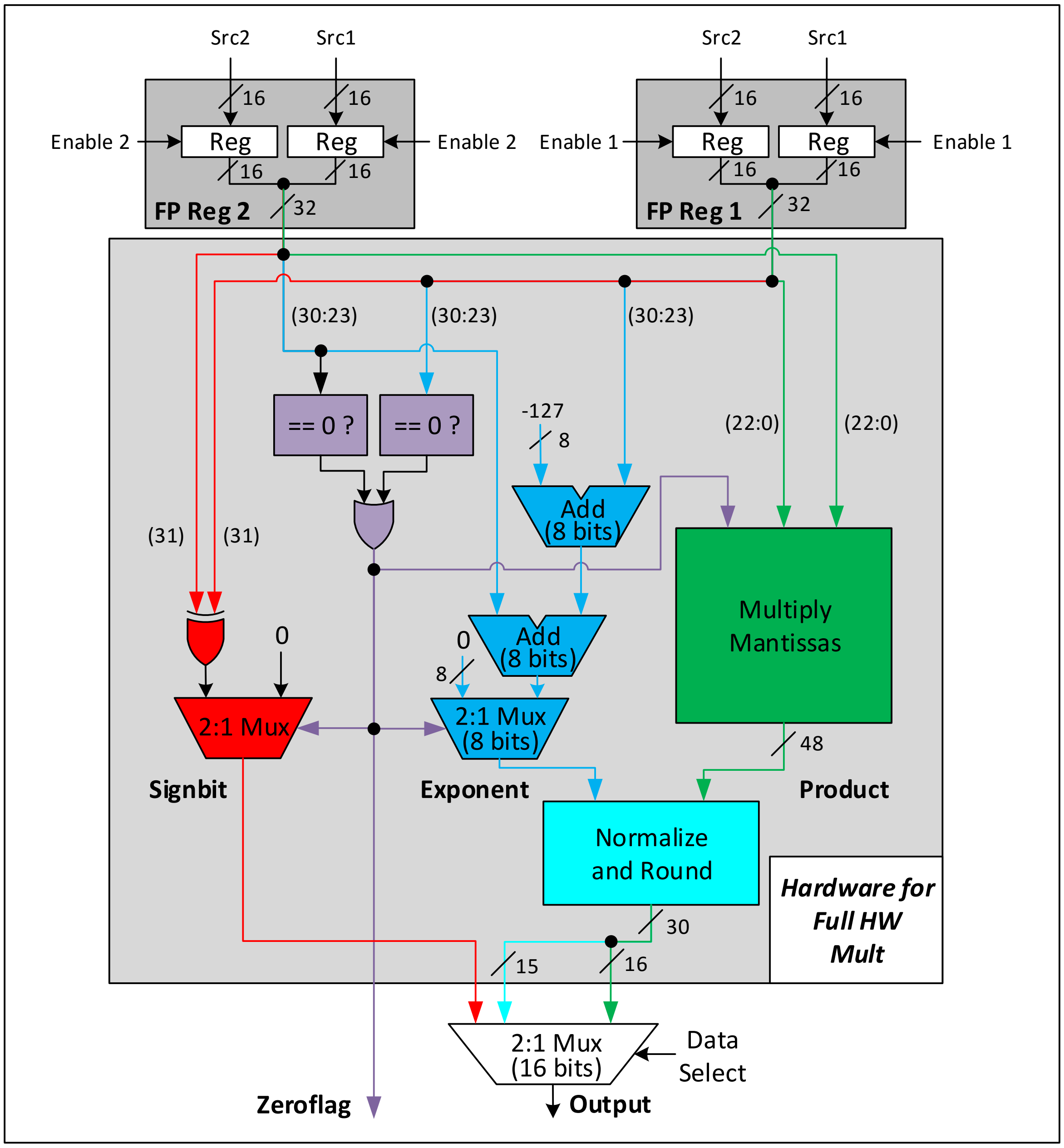


Figure 3.4: Full Hardware Mult Datapath.

3.2.4 Division Module (Full HW Div)

The full hardware division module uses the *FPDiv* instruction with a 30-cycle execution latency to perform FP divides. The non-restoring division algorithm is used for performing division in full hardware [44].

A separate version of this module is created for a 32-bit datapath and word size. *Full HW Div (32-bit I/O)* uses the *FPDiv32* instruction to perform division with a 30-cycle execution latency. Assuming operands are read from a processor's local memory, then a single instruction can perform division.

3.2.5 Square Root Module (Full HW Sqrt)

The full hardware square root module uses the *FPSqrt* instruction with a 26-cycle execution latency to perform FP square root operations. The non-restoring square root algorithm is used for calculating the square root in full hardware [52].

A separate version of this module is created for a 32-bit datapath and word size. *Full HW Sqrt (32-bit I/O)* uses the *FPSqrt32* instruction to perform square root operations with a 26-cycle execution latency. A single instruction can perform square root operations.

3.3 Proposed Hybrid Implementations with Unsigned, Shift-Carry, and Leading Zero Detection Support

As mentioned in Section 2.4, the example architecture supports only signed operations. To determine the throughput and area achievable by increasing the instruction set, USL support is added to the target platform's ISA. Several ISA modifications are implemented, including adding unsigned operation support, leading zeros detection, and additional shift-carry instructions. These extra shift instructions have the ability to set a carry flag if data are shifted out. Table 3.1 indicates the instructions utilized in each hybrid implementation. Each of these instructions has a single-cycle execution latency, except for the MAC instructions, which require two cycles. Similar to the full software kernels, each value is split across four 16-bit words as shown in Figure 3.5. The following instructions are implemented:

3.3.0.1 SUBU

Unsigned subtraction.

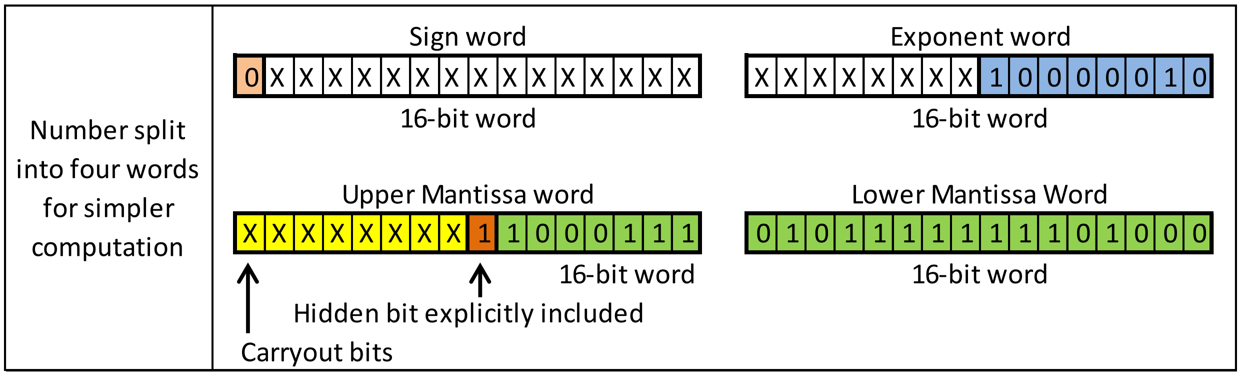

Number split into four words for simpler computation
Sign word
0 X X X X X X X X X X X X X X X
16-bit word
Exponent word
X X X X X X X X 1 0 0 0 0 0 1 0
16-bit word
Upper Mantissa word
X X X X X X X X 1 1 0 0 0 1 1 1
16-bit word
Lower Mantissa Word
0 1 0 1 1 1 1 1 1 1 1 0 1 0 0 0
16-bit word
Hidden bit explicitly included
Carryout bits

Figure 3.5: FP values split into four words to simplify software computation with USL instructions. The hidden bit is explicitly included in the upper mantissa word. The yellow boxes indicate bit locations available for carryout bits that may arise during intermediate computations. Since USL instructions are available, the 16 LSBs of the mantissa fit into the last data word without needing bits for carryout.

3.3.0.2 SUBUC

Unsigned subtraction instruction with borrow if the carry flag is asserted.

3.3.0.3 ADDU

Unsigned addition.

3.3.0.4 ADDUC

Unsigned addition with carry-in.

3.3.0.5 SHLC

Shift left one bit and shift in a 1 bit at the LSB if the carry flag is asserted.

3.3.0.6 SHRC

Shift right and shift in a 1 bit at the MSB if the carry flag is asserted.

3.3.0.7 LZD

Returns number of leading zeros.

3.3.0.8 MULTUL

Unsigned multiply that returns the 16 LSBs of the result. The accumulator is not overwritten.

3.3.0.9 MACCUL

Unsigned multiply that returns the 16 LSBs of the result. The accumulator is overwritten with the result.

3.3.0.10 MACUL

Unsigned multiply-accumulate that returns the lower 16 LSBs of the result.

3.3.0.11 MACUH

Unsigned multiply-accumulate that returns the 16 MSBs of the result.

3.3.0.12 ACCSHU

Unsigned right shift for the accumulator and returns the 16 LSBs of the result.

3.3.1 Addition/Subtraction Hybrid Implementation with USL Support (Hybrid Add/Sub w/ USL)

Addition and subtraction with the hybrid implementation with USL support are performed similar to the *Full SW Add/Sub* kernel, except for the following: *SUBU* is used to calculate the larger operand, the exponent difference, and subtract the mantissas; *SUBUC* aids with mantissa subtraction; *ADDU* and *ADDUC* are used to add mantissas and for rounding; the shift instructions *SHLC* and *SHRC* shift the mantissa for normalization; and *LZD* counts leading zero bits for normalization.

3.3.2 Multiplication Hybrid Implementation with USL Support (Hybrid Mult w/ USL)

Multiplication with the hybrid implementation with USL support is performed similar to the *Full SW Mult* kernel, except for the following differences: *MULTUL*, *MACCUL*, *MACUL*, *MACUH*, and *ADDU* help perform mantissa multiplication; *SHLC* shifts for normalization; and *ADDU* and *ADDUC* are used for rounding up.

3.3.3 Division Hybrid Implementation with USL Support Version 1 (Hybrid Div w/ USL Ver. 1)

This implementation uses the long-division algorithm [43]. Unsigned addition/subtraction and added shift instructions reduce the instruction count for exponent calculation, divisor and dividend mantissa subtraction, rounding and normalization.

3.3.4 Division Hybrid Implementation with USL Support Version 2 (Hybrid Div w/ USL Ver. 2)

This implementation uses the Newton-Raphson division algorithm [1]. The unsigned addition/subtraction, multiply-accumulate, and additional shift instructions reduce the instruction count for calculating the exponent and initial estimate, executing the Newton-Raphson iterations, multiplying the input by the reciprocal, rounding, and correcting the LSB.

3.3.5 Square Root Hybrid Implementation with USL Support Version 1 (Hybrid Sqrt w/ USL Ver. 1)

This implementation uses the digit-by-digit method. Unsigned addition/subtraction instructions decrease the instruction count for exponent calculation, root incrementing, radicand and squared root subtraction, and rounding. Unsigned multiply-accumulate reduces the instruction count for squaring the root being tested, and the additional shift instructions assist with setting the next radicand bit and alignment.

3.3.6 Square Root Hybrid Implementation with USL Support Version 2 (Hybrid Sqrt w/ USL Ver. 2)

This implementation uses the Newton-Raphson square root algorithm [1]. Unsigned addition/subtraction and multiply-accumulate instructions ease rounding and the correction of the LSB, calculating the exponent, determining the initial value, and performing the Newton-Raphson iterations. The additional shift instructions help with preparing the input data for later calculations and the Newton-Raphson iterations.

3.4 Proposed Hybrid Implementations with Custom FP-Specific Hardware

Hybrid implementations with CFP hardware are composed of fixed-point software and CFP instructions operating together on FP workloads. They increase throughput by reducing the bottlenecks of full software kernels and require less area than full hardware modules.

CFP instructions perform operations on the data stored in a set of FP registers. Similar to the full hardware modules, each value is stored as two 16-bit words as shown in Figure 3.2. Table 3.1 indicates the instructions utilized in each implementation, where each instruction has a single-cycle execution latency. Depending on the specific area and throughput constraints, different implementations can be combined into various FPU configurations. Seven implementations are described next.

3.4.1 Addition/Subtraction Hybrid Implementation with CFP Hardware Version 1 (Hybrid Add/Sub w/ CFP Ver. 1)

This implementation performs FP addition/subtraction using fixed-point and CFP instructions. Fixed-point instructions are used to determine the larger and smaller exponents. The four custom FP instructions described next perform the rest of the FP operation.

3.4.1.1 FPAdd_SatAlign

After the operands are sorted and the exponent difference is calculated in software, the FP registers are loaded with the original FP operands. Using the software-calculated exponent difference, this instruction saturates the alignment amount, then aligns and adds the mantissas. The hidden bits are inserted and the sticky bit is determined during alignment. When effective subtraction is performed, the smaller magnitude operand's mantissa is inverted and a one is added. The unnormalized result is stored in a FP register and the 16 MSBs are returned.

3.4.1.2 LZD

Following mantissa addition, LZD counts the leading zeros of the result. This value is used for shifting during normalization. *LZD* is also one of the USL support instructions described in Section 3.3.

3.4.1.3 BShiftL

This instruction uses the shift amount determined by *LZD* and the sum stored in the FP registers by *FPAdd_SatAlign. BShiftL* shifts left for normalization, adjusts the exponent, and stores the 27 LSBs of the result in a FP register. This instruction can also be used by non-FP general purpose workloads for large shifts.

3.4.1.4 FPAdd_Round

Following normalization, *FPAdd_Round* performs rounding and exponent adjustment. The final result is written to a FP register and the 16 MSBs are output.

3.4.2 Addition/Subtraction Hybrid Implementation with CFP Hardware Version 2 (Hybrid Add/Sub w/ CFP Ver. 2)

The second version of hybrid addition/subtraction with CFP hardware performs operand sorting, exponent difference calculation and saturation with fixed-point software instructions. This implementation utilizes *FPAdd_AlignSmall,* which relegates the sticky bit calculation and hidden bit insertion to software.

3.4.2.1 FPAdd_AlignSmall

This instruction aligns and adds the mantissas using the software calculated shift amount. Similar to *Hybrid Add/Sub w/ CFP Ver. 1* and *Hybrid Add/Sub w/ CFP Ver.* 3, the rest of the FP operation is performed using the *LZD*, *BShiftL*, and *FPAdd_Round* instructions.

3.4.3 Addition/Subtraction Hybrid Implementation with CFP Hardware Version 3 (Hybrid Add/Sub w/ CFP Ver. 3)

Algorithm 1 displays the pseudocode for this implementation; variables are italicized, comments are in green font, CFP instructions are bolded and in blue font, and all other lines represent operations carried out by fixed-point software. After the input operands are loaded, *FPAdd_Compare* sorts the operands and calculates the saturated shift amount and stores this value in *ExpDiff*. *FPAdd_Align* reads the variable *ExpDiff* to perform the mantissa alignment, possibly complements one of the mantissas, and then adds them. The result is stored in *FPReg1* and the 16 MSBs are stored in *FPreg3*. *LZD* stores the leading zeros count of *FPReg3* in *UpperZeros*. If all bits were zero, then *LZD* counts the leading zeros in the LSBs of *FPReg1*. *BShiftL* then normalizes after adding the leading zeros counts together. *FPAdd_Round* then rounds the normalized result and outputs the 16 MSBs. *FPAdd_Compare* and *FPAdd_Align* are described next.

Algorithm 1 Pseudocode of Hybrid Add/Sub w/ CFP Ver. 3

```
while true do
    $FPReg1_{[31:16]} \leftarrow Input$                    \\ Load Operands
    $FPReg1_{[15:0]} \leftarrow Input$
    $FPReg2_{[31:16]} \leftarrow Input$
    $FPReg2_{[15:0]} \leftarrow Input$
    FPAdd_Compare $ExpDiff$                                \\ Sort Operands
    FPAdd_Align $ExpDiff$                                  \\ Align & Add
    $LowerZeros \leftarrow 0$                              \\ Initialize 0s count
    LZD $UpperZeros$                                       \\ Count 0's in $FPReg3$
    $Temp \leftarrow UpperZeros - 16$
    if $Temp == 0$ then                                    \\ Check if all 0's
        $FPReg3 \leftarrow FPReg1_{[31:16]}$
        LZD $LowerZeros$                                   \\ Count 0's in $FPReg3$
    end if
    \\ Correct count and add counts together
    $UpperZeros \leftarrow UpperZeros - 4$
    $ShiftAmount \leftarrow UpperZeros + LowerZeros$
    BShiftL $ShiftAmount$                                  \\ Normalize
    FPAdd_Round $Output$                                   \\ $Output \leftarrow FPReg2_{[31:16]}$
    $Output \leftarrow FPReg2_{[15:0]}$
end while
```

3.4.3.1 FPAdd_Compare

This instruction sorts both operands after they are loaded into the FP registers. The sorted operands are then rewritten into the FP registers. *FPAdd_Compare* also saturates the shift amount since exponent differences greater than 25 result in identical mantissa alignments.

3.4.3.2 FPAdd_Align

This instruction is similar to *FPAdd_SatAlign*, except that it doesn't perform saturation for the alignment shift amount since this is handled by *FPAdd_Compare*. This instruction reads the sorted operands from the FP registers, then aligns and adds them using the shift amount. Similar to *Hybrid Add/Sub w/ CFP Ver. 1*, the rest of the FP operation is performed using the *LZD*, *BShiftL*, and *FPAdd_Round* instructions.

3.4.4 Addition/Subtraction Hybrid Implementation with CFP Hardware Version 4 (Hybrid Add/Sub w/ CFP Ver. 4)

This implementation sorts the operands, calculates the hidden bit, sticky bit, and saturated exponent difference using fixed-point instructions. *FPAdd_Align* aligns mantissas, potentially complements one of them, and adds them together. *Shift_LZA* replaces *BShiftL* and *LZD,* and is described next.

3.4.4.1 Shift_LZA

An LZA forms an indicator string to anticipate the leading zeros in parallel with the addition [1]. The leading zeros count is then used for normalization shifting, followed by a possible 1 bit correction. The rest of the FP operation is performed using *FPAdd_Round.*

3.4.5 Multiplication Hybrid Implementation with CFP Hardware Version 1 (Hybrid Mult w/ CFP Ver. 1)

This version of hybrid multiplication with CFP hardware performs mantissa multiplication, and exponent and sign bit calculation using fixed-point software instructions. *FPMult_NormRndCarry* performs the normalization and rounding steps and sets a carry flag that is used to adjust the exponent if necessary. This instruction is described next.

3.4.5.1 FPMult_NormRndCarry

Following mantissa multiplication and exponent calculation, the product is loaded into a FP register. The normalized and rounded mantissa is written back into a FP register, the 16 MSBs are returned, and the carry flag is set if a carry out occurs from rounding. If the carry flag is set, the exponent is incremented in software. Figure 3.6 shows the hardware for implementing this hybrid implementation in the execution stage of the target platform.

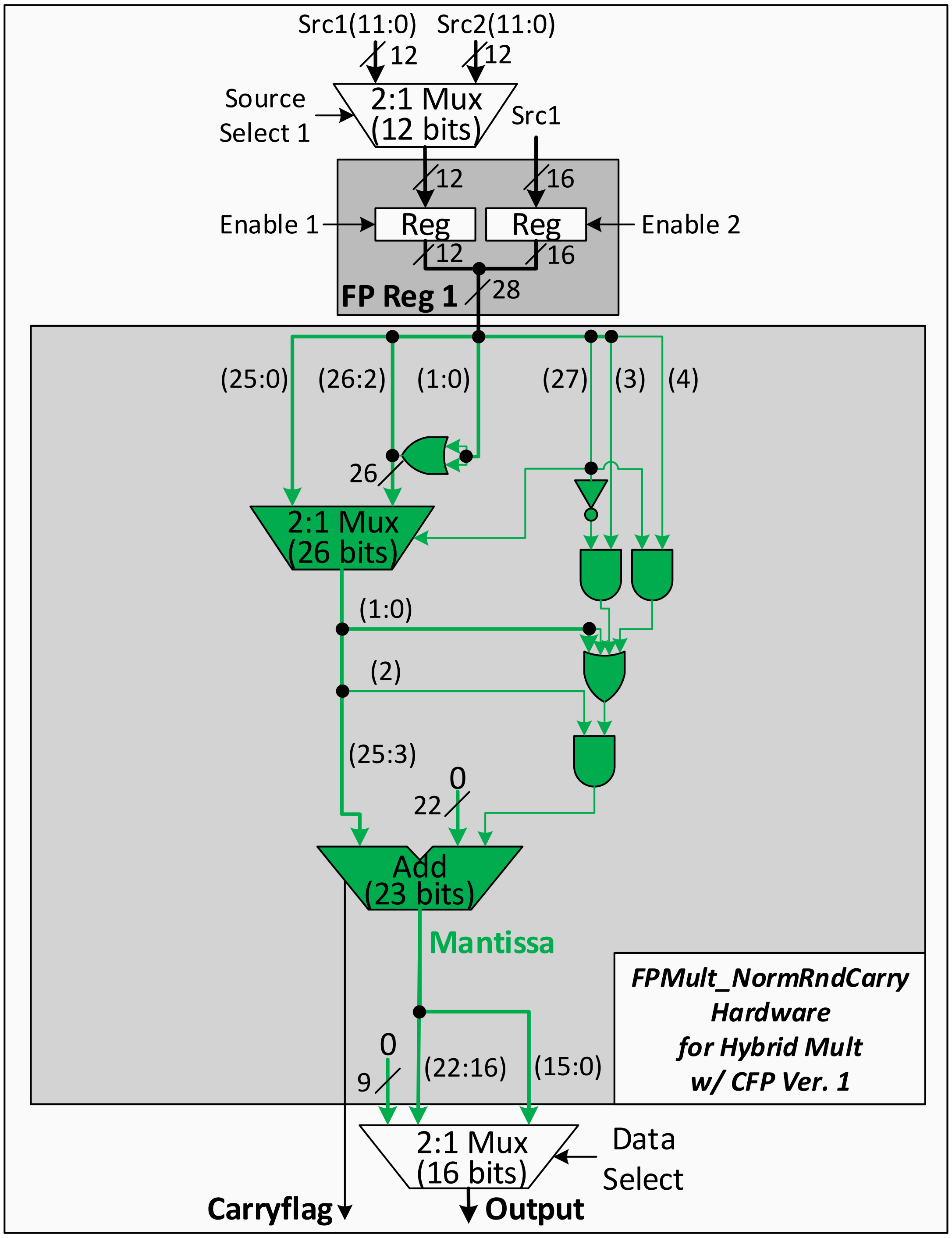


Figure 3.6: (a) The hardware to implement the *FPMult_NormRndCarry* instruction for the *Hybrid Mult w/ CFP Ver. 1* implementation. FP Reg 1 is loaded with the product of the mantissa multiplication. The rounded result and carry bit are produced. If the carry flag is set, the exponent is incremented in software.

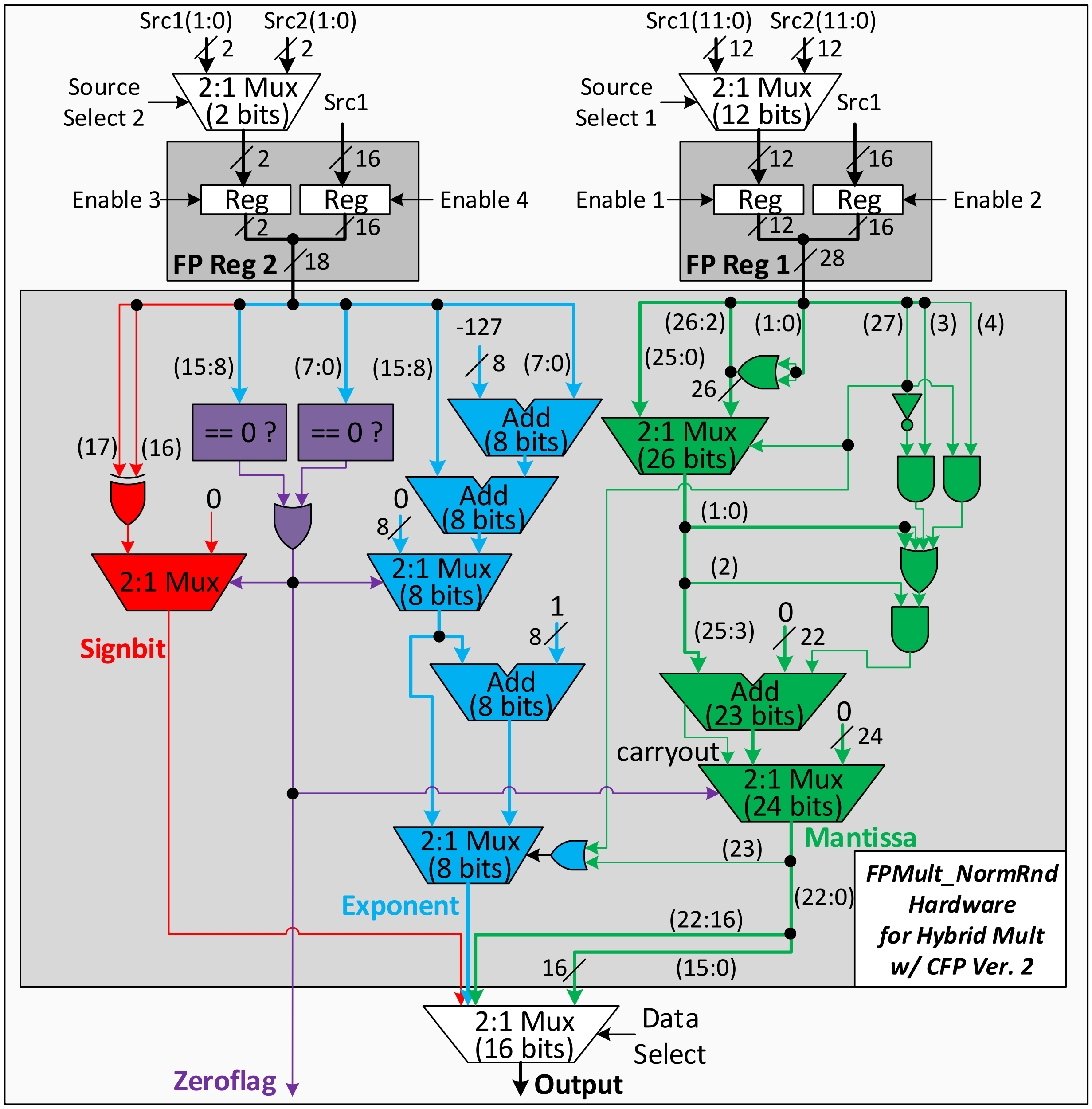


Figure 3.7: The hardware to implement the *FPMult_NormRnd* instruction for the *Hybrid Mult w/ CFP Ver. 2* implementation. FP Reg 1 is loaded with the product of the mantissa multiplication and FP Reg 2 is loaded with the sign bits and exponents of both operands. The sign, exponent, and rounded result are then produced, as well as a zero flag.

3.4.6 Multiplication Hybrid Implementation with CFP Hardware Version 2 (Hybrid Mult w/ CFP Ver. 2)

This implementation performs mantissa multiplication in software using fixed-point instructions. The *FPMult_NormRnd* custom instruction calculates the new exponent, and performs normalization and rounding. This instruction is described next.

3.4.6.1 FPMult_NormRnd

Following mantissa multiplication in software, the product is loaded into an FP register. The other FP register is loaded with the exponent and sign bits. The new sign bit, exponent, and normalized and rounded product are then calculated. The result is written back to the FP registers and selectable via a 16-bit mux. Figure 3.7 shows the hardware for adding this hybrid implementation into the execution stage of the target platform.

3.4.7 Division Hybrid Implementation with CFP Hardware Version 1 (Hybrid Div w/ CFP Ver. 1)

The non-restoring division algorithm is used for performing FP division with this implementation [44]. The exponent and sign bit of the result are first determined in software. The custom FP instruction described next performs the rest of the operation.

3.4.7.1 FPDiv_LoopExpAdj

After both inputs and the partially computed exponent are loaded into the FP registers, this instruction performs the division loop to calculate the final mantissa. The exponent is then adjusted in hardware following normalization and rounding.

3.4.8 Square Root Hybrid Implementation with CFP Hardware Version 1 (Hybrid Sqrt w/ CFP Ver. 1)

The non-restoring square root algorithm is used for calculating the FP square root with this implementation [52]. The exponent of the result is first determined in software. The custom FP instruction described next performs the rest of the operation.

3.4.8.1 FPSqrt_Loop

After loading the input into the FP register, this instruction performs the square root loop to calculate the final mantissa.

3.5 Results and Comparisons

Each implementation is synthesized with a 65 nm CMOS standard cell library using Synopsys DC Compiler with a 1.3 V operating voltage and 25°C operating temperature and clock frequencies of 600, 800, 1000, and 1200 MHz. The FP designs are synthesized using various clock frequencies and plotted in Figure 3.8 to explore how results vary with the operating condition. The remaining results utilize a 1200 MHz clock frequency to match the platform processor's clock frequency.

For accuracy and performance analysis, FPgen [96], a test-suite for verifying FP datapaths is used to include test cases unlikely to be covered by pure random test generation. This testing is supplemented by using millions of pseudorandomly generated FP values on the normalized value interval $\pm[2^{-126}, (2-2^{-23}) \times 2^{127}]$. Cycles per FP operation (FLOP) data was gathered for each implementation by performing FP operations on these datasets.

With the exception of the full software kernels, each design adds circuitry to the platform processor. The area for this circuitry is referred to as "additional area". The area for each individual design is calculated by scaling the relative area increase determined from synthesis for the circuitry added to the datapath. Additional circuitry is added to the platform processor core by modifying the original RTL and ensuring that the original timing constraints are not violated. To satisfy the timing constraints, certain instructions have a multi-cycle execution latency, as mentioned in Sections 3.2–3.3.

3.5.1 Individual FP Designs Compared

Figure 3.8 plots additional area versus delay for each design using four different target clock frequencies ranging from 600–1200 MHz. Additional area is plotted versus cycles per FLOP times the clock period in nanoseconds. The designs are plotted on separate graphs according to operation type. Since the FMA supports both addition/subtraction and multiplication operations, it is plotted in both (a) and (b). Designs for both a 16-bit and 32-bit word size and datapath are plotted. The 32-bit I/O designs are smaller than their counterparts because they do not consider the additional area required for a processor with a 32-bit word size and datapath.

As expected, designs providing higher throughput generally require greater area. Regardless of clock period, the dedicated FMA requires the most area. Having a split multiplier and addition/subtraction design requires less area than an FMA due to extra circuitry present such as a wider alignment shifter, adder, and normalization shifter, as well as the LZA and end-around carry adder present in the FMA. A large area savings is not observed for most designs when using a longer clock period. Additionally, the target platform utilizes a 1.2 GHz clock frequency and a separate clock is not available for the FP circuitry; therefore, the results for the rest of this dissertation consider a 1.2 GHz clock frequency.

Table 3.2 lists the throughput per core, instruction count, and area for each design. Figure 3.9 plots the throughput and area for each design. Each implementation is plotted on a separate graph according to operation type. Full software kernels are found on the left side of the plots, along the y-axis. Full hardware modules are located in the bottom right hand of the plots. The hybrid implementations (with USL support and with CFP hardware) are found in the middle of the plots.

The optimal design subject to an area constraint is determined by selecting an implementation that uses less area than the constraint, and requires the fewest average cycles per FLOP. As an example, consider an area constraint, A_{max}, equal to 10% of the target platform processor area. For this example, more area is allocated for addition/subtraction and multiplication hardware because division and square root are less frequent operations [45, 46]. As shown in Figure 3.9(a), an area constraint is first set for the addition/subtraction design, $A_{70\%\ max}$ equal to 70% of the maximum area constraint. The optimal implementation that requires the least cycles per FLOP while not exceeding the area constraint is denoted by the green arrow as *Hybrid Add/Sub w/ CFP Ver.* 3. Using the remaining area, the area constraint $A_{mult\ max}$ is set in (b), and the optimal design for multiplication is *Hybrid Mult w/ CFP Ver. 2.* There remains available area, $A_{div\ max}$, for improving division throughput in (c) using *Hybrid Div w/ USL Ver. 1.* Finally, the optimal design for square root is determined in (d) to be a software kernel, *Full SW Sqrt Ver. 1.*

The full hardware designs require the most area and achieve the highest throughput;

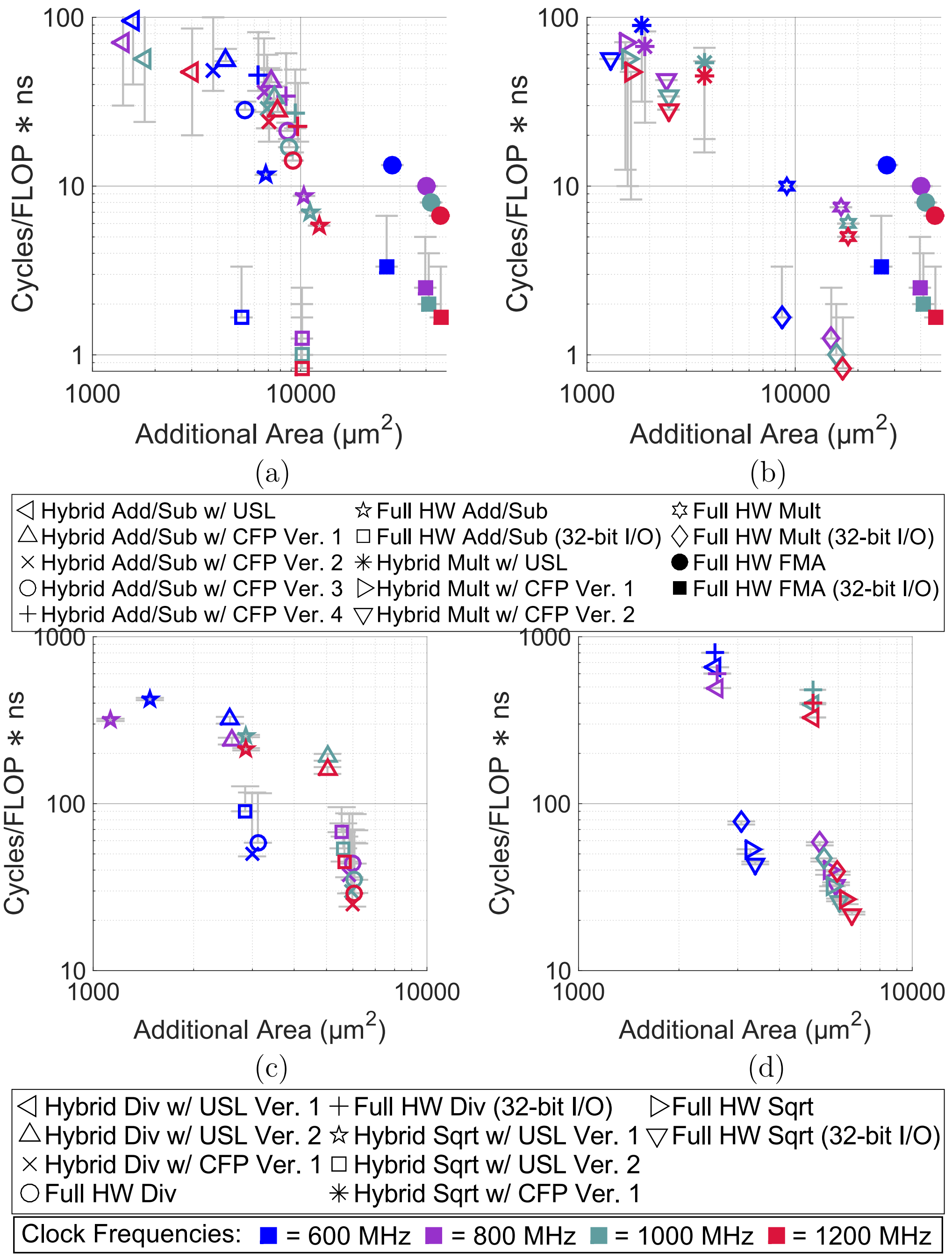


Figure 3.8: Result of exploring different cycle times for different FP designs. The markers denote the average cycles per FLOP times the clock period, and the interval bar endpoints for each symbol denote the corresponding minimum and maximum. Cycles per FLOP are scaled by the number of cores required. Results are obtained from synthesis in 65 nm CMOS at 1.3 V and 600–1200 MHz. (a) Addition/subtraction designs. (b) Multiplication designs. (c) Division designs. (d) Square root designs.

Table 3.2: Throughput, Instruction Count, and Area for FMA Designs.

	FP Design	Instruction Count (Static)	Throughput Per Core (MFLOPS)	Average Speedup (Per Core)	Cycles/FLOP (Per Core)	Additional Area (µm^2)	Area Increase (%)
Addition/Subtraction Designs	*Full SW Add/Sub*	222	9.70	1.00x	124	0	0
	Full HW Add/Sub	6	171	17.6x	7	13 621	8.10
	Full HW Add/Sub (32-bit I/O)	1	1200	123x	1	10 402	6.19
	Hybrid Add/Sub w/ USL	127	21.1	2.18x	57	3136	1.87
	Hybrid Add/Sub w/ CFP Ver. 1	40	35.8	3.69x	34	8740	5.20
	Hybrid Add/Sub w/ CFP Ver. 2	78	41.5	4.28x	29	7603	4.52
	Hybrid Add/Sub w/ CFP Ver. 3	**17**	**70.6**	**7.28x**	**17**	10 821	**6.44**
	Hybrid Add/Sub w/ CFP Ver. 4	31	44.1	4.55x	28	12 532	7.46
	Full HW FMA	8	150	15.5x	8	55 121	32.8
	Full HW FMA (32-bit I/O)	2	600	61.9x	2	51 771	30.8
Multiplication Designs	*Full SW Mult*	66	17.4	1.00x	69	0	0
	Full HW Mult	6	200	11.5x	6	18 189	10.8
	Full HW Mult (32-bit I/O)	1	1200	69.0x	1	17 258	10.3
	Hybrid Mult w/ USL	54	22.4	1.29x	54	3665	2.18
	Hybrid Mult w/ CFP Ver. 1	52	21.2	1.22x	57	1659	0.99
	Hybrid Mult w/ CFP Ver. 2	**34**	**35.3**	**2.03x**	**34**	2596	**1.54**
	Full HW FMA	8	150	8.62x	8	55 121	32.8
	Full HW FMA (32-bit I/O)	2	600	34.5x	2	51 771	30.8
Division Designs	*Full SW Div Ver. 1*	84	1.54	1.00x	777	0	0
	Full SW Div Ver. 2	1032	0.70	0.45x	1719	0	0
	Full HW Div	8	34.3	22.3x	35	6230	3.71
	Full HW Div (32-bit I/O)	3	40.0	26.0x	30	5985	3.56
	Hybrid Div w/ USL Ver. 1	**63**	**4.73**	**3.07x**	**254**	2957	**1.76**
	Hybrid Div w/ USL Ver. 2	125	6.24	4.05x	193	5138	3.06
	Hybrid Div w/ CFP Ver. 1	28	22.2	14.4x	54	5706	3.39
Square Root Designs	***Full SW Sqrt Ver. 1***	**114**	**0.80**	**1.00x**	**1500**	**0**	**0**
	Full SW Sqrt Ver. 2	1482	0.46	0.58x	2610	0	0
	Full HW Sqrt	8	37.5	46.9x	32	6553	3.90
	Full HW Sqrt (32-bit I/O)	3	46.2	57.8x	26	6772	4.03
	Hybrid Sqrt w/ USL Ver. 1	60	2.49	3.11x	481	5138	3.06
	Hybrid Sqrt w/ USL Ver. 2	214	3.05	3.81x	394	5138	3.06
	Hybrid Sqrt w/ CFP Ver. 1	20	25.5	31.9x	47	6148	3.66
FMA Designs	*Full HW FMA*	9	133	22.9x	9	55 121	32.8
	Full HW FMA (32-bit I/O)	2	600	103x	2	51 771	30.8

The optimal implementations subject to an area constraint from Figure 3.9 are denoted in bold font.
FMA results reported separately for addition/subtraction, multiplication, and fused multiply-add operations.

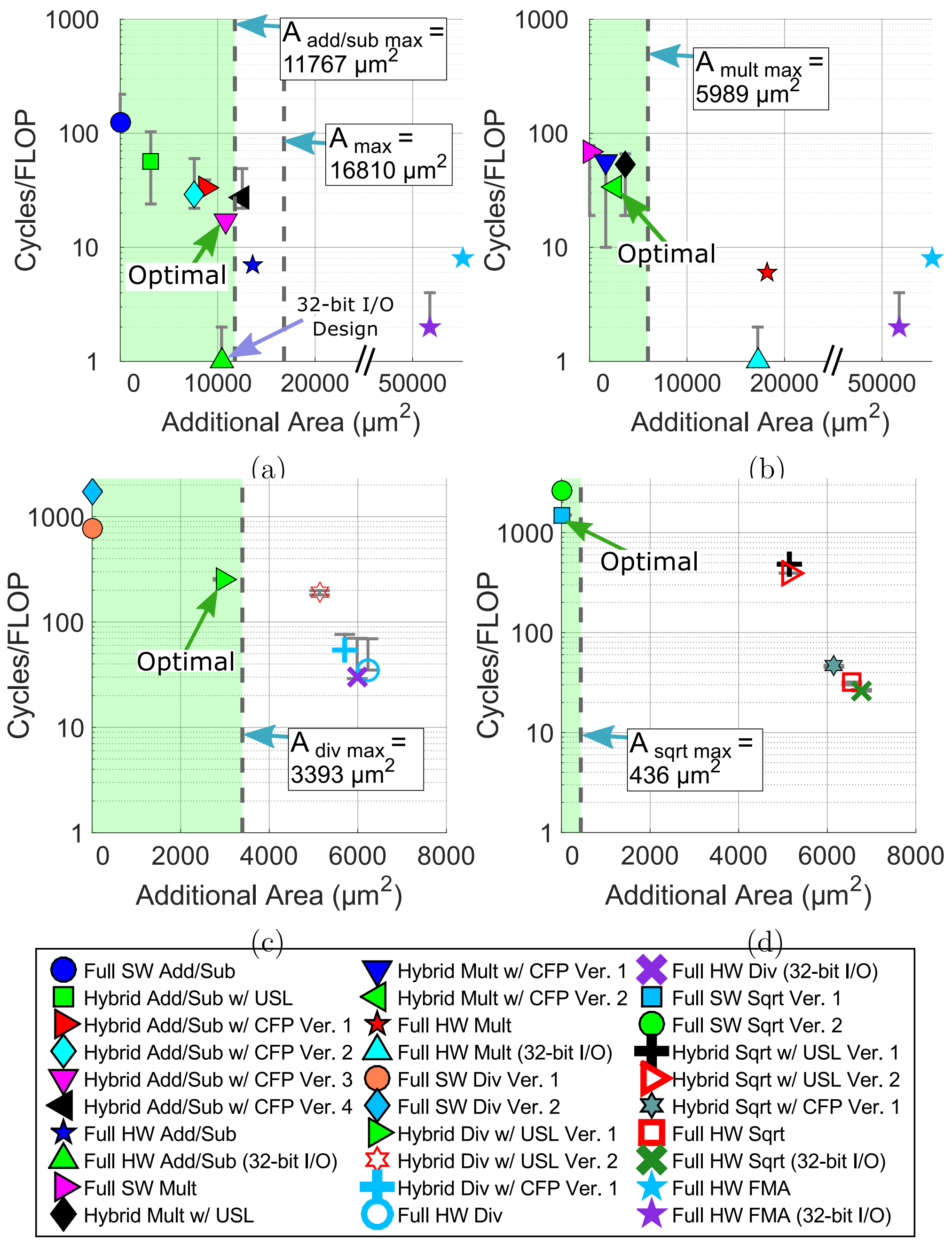


Figure 3.9: Additional area versus cycles per FLOP for each FP design and determining the optimal designs from area constraints. The 30 markers in the legend denote the average cycles per FLOP, and the endpoints of the interval bars for each symbol denote the corresponding minimum and maximum. Cycles per FLOP are scaled by the number of cores required. Area constraints are indicated by the vertical dashed lines. The optimal design has the fewest average cycles per FLOP and an area below the area constraint. For this example, the area available for additional hardware, A_{max}, is equal to 10% of the processor area. (a) The optimal adder/subtractor is first determined using an area constraint of 70% of the maximum area constraint, (b-c) the optimal multiplication, division, and square root designs are determined using the remaining available area. Designs satisfying the area constraint appear in the green regions. Results are obtained from synthesis in 65 nm CMOS at 1.3 V and 1.2 GHz.

however, none of these implementations meet the area constraint, and the FMA is the largest implementation, increasing processor area by 32.8%. Except for multiplication, the hybrid implementations with USL support require the least area to improve throughput. They can also be used for general purpose workloads because the USL instructions are non-FP specific. For the full software kernels, division and square root require less cycles per FLOP using the long-division and digit-by-digit algorithms, respectfully. However, the division and square root hybrid implementations with USL support require slightly less cycles per FLOP when using the Newton-Raphson algorithm.

3.5.2 Comparison when Combining FP Designs

To compare the throughput and area when combining multiple designs, the FP designs discussed in Sections 3.1–3.4 are combined into 38 functionally-equivalent FPU implementations consisting of an addition/subtraction and multiplication unit. These designs are evaluated for performing unfused multiply-add, and Newton-Raphson division and square root. These Newton-Raphson and FMA implementations of divide and square root are mapped in a pipelined fashion and loops are unrolled to potentially provide high throughput [97]. These implementations are compared against full software, full hardware, and hybrid designs using the long-division, digit-by-digit, non-restoring, or Newton-Raphson algorithm.

Figure 3.10 plots the cycles per FLOP for the unfused multiply-add operation versus additional area. Just as in Section 3.5.1, the optimal design subject to an area constraint can be easily determined. The optimal multiply-add design is first determined, followed by the optimal division and square root designs. For this example, the area constraint, A_{max}, is equal to 10% of the target platform processor area; however, 80% of the area constraint, $A_{80\%\ max}$, is allocated for the addition/subtraction and multiplication hardware. Based upon the cycles per FLOP for performing the multiply-add operation, and denoted by the green arrow, *Hybrid Add/Sub w/ CFP Ver. 3* is the optimal addition/subtraction design and *Hybrid Mult w/ CFP Ver. 2* is the optimal multiplication design. Despite offering reduced latency and higher throughput, the additional area overhead for the FMA does not meet the area constraint.

Figure 3.11 plots the cycles per FLOP for the division operation versus additional area. The combinations of addition/subtraction and multiplication designs from Figure 3.10 are used to perform Newton-Raphson division. Using one of the FPU implementations from Figure 3.10

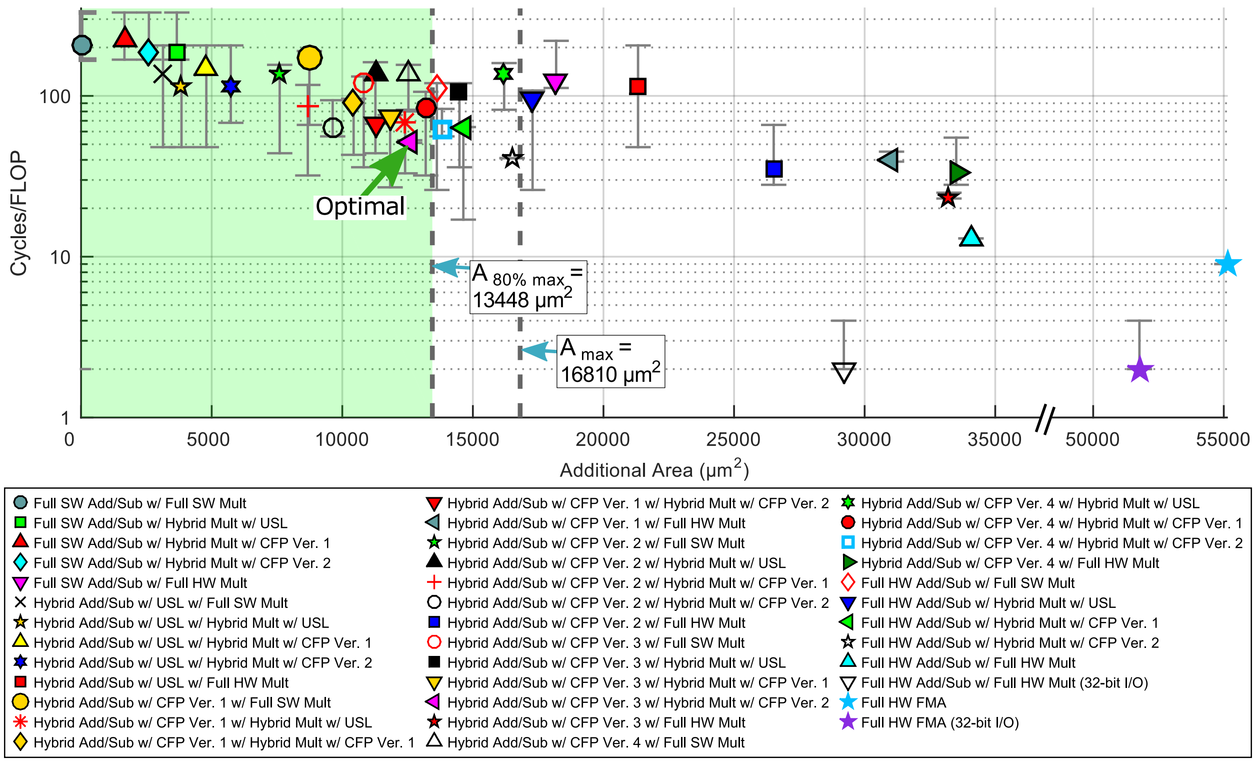


Figure 3.10: Multiply-add (A+B×C) area versus cycles per FLOP for all FPU implementations and determining the optimal implementation from an area constraint. Design-point symbols are placed at the average cycles/FLOP point with interval bars showing the range over all possible values. Cycles per FLOP are scaled by the number of cores required. Area constraints are indicated by the vertical dashed lines. The optimal design has the fewest average cycles per FLOP and an area below the area constraint. For this example, the area available for additional hardware, A_{max}, equals 10% of the processor area. The optimal adder/subtractor and multiplier are first determined using 80% of the maximum area constraint, $A_{80\%\ max}$. Using the remaining available area, the optimal designs for division and square root are determined in Figure 3.11 and Figure 3.12, respectively. Designs satisfying the area constraint appear in the green highlighted region. Results are obtained from synthesis in 65 nm CMOS at 1.3 V and 1.2 GHz.

to implement division does not require any additional area other than that already incurred for the addition/subtraction and multiplication designs. Using the remaining area from choosing a design in Figure 3.10, the optimal design for improving division throughput is determined. The division implementation using the optimal FPU from Figure 3.10 is denoted by the blue arrow; however, it does not improve division throughput versus full software. Subject to the constraint $A_{div\ max}$, the optimal division design is *Hybrid Div w/ USL Ver. 1*, which uses the long-division algorithm. Scaled by core count, none of the addition/subtraction and multiplication combinations using Newton-Raphson division increase throughput.

Figure 3.12 plots the cycles per FLOP for the square root operation versus additional area.

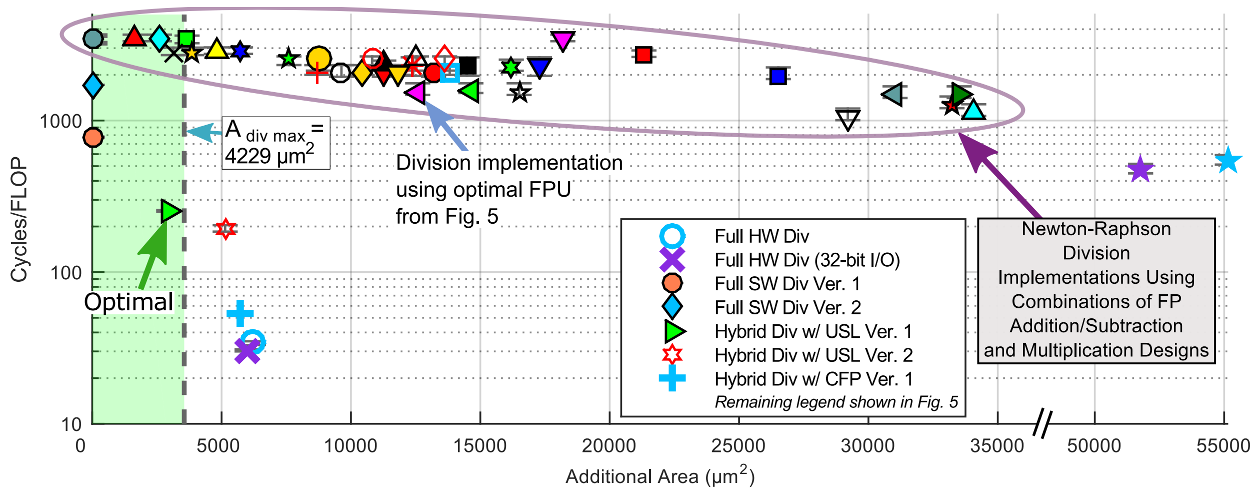


Figure 3.11: Division area versus cycles per FLOP for all implementations and determining the optimal implementation from an area constraint. Four methods are evaluated for performing division; the Newton-Raphson method, division in software, division in hardware, and hybrid division. Design-point symbols are placed at the average cycles/FLOP point with interval bars showing the range over all possible values. Cycles per FLOP are scaled by the number of cores required. The optimal division implementation is determined with the remaining available area, $A_{div\ max}$. Designs satisfying the area constraint appear in the green highlighted region. Using the remaining available area, the optimal design for square root is determined in Figure 3.12. The remaining legend is shown in Figure 3.10. Results from synthesis in 65 nm CMOS at 1.3 V and 1.2 GHz.

The combinations of addition/subtraction and multiplication designs from Figure 3.10 are used to perform Newton-Raphson square root. Using one of the FPU implementations from Figure 3.10 to implement square root does not require any additional area other than that already incurred for the addition/subtraction and multiplication designs. $A_{sqrt\ max}$ is the area left over from choosing an addition/subtraction and multiplication design in Figure 3.10, and a division design in Figure 3.11, and is used to determine an optimal square root design. The square root implementation using the optimal FPU from Figure 3.10 is denoted by the blue arrow; however, it achieves lower throughput than the software implementation. Subject to the area constraint, the optimal square root design is *Full SW Sqrt Ver. 1*, which implements the digit-by-digit algorithm. Contrary to division, some Newton-Raphson square root implementations using combinations of addition/subtraction and multiplication designs improve throughput over the full software implementation.

The FPU implementations are also evaluated for performing two scientific kernel benchmarks. Figure 3.13 plots the cycles per FLOP for a radix-2 complex butterfly computation in (a), and a 2x2 matrix multiplication in (b). These benchmarks are two examples of kernels in

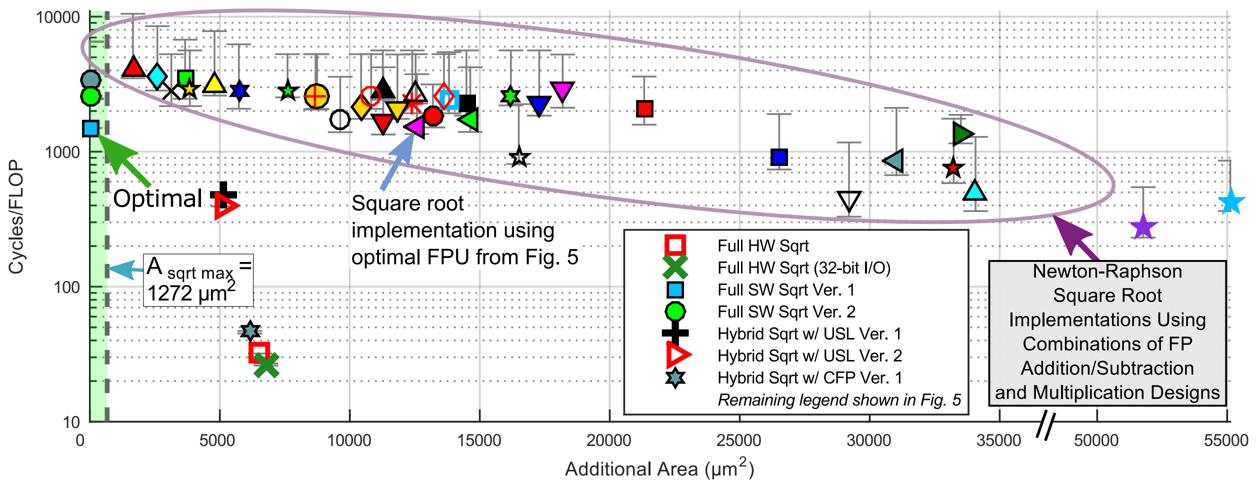


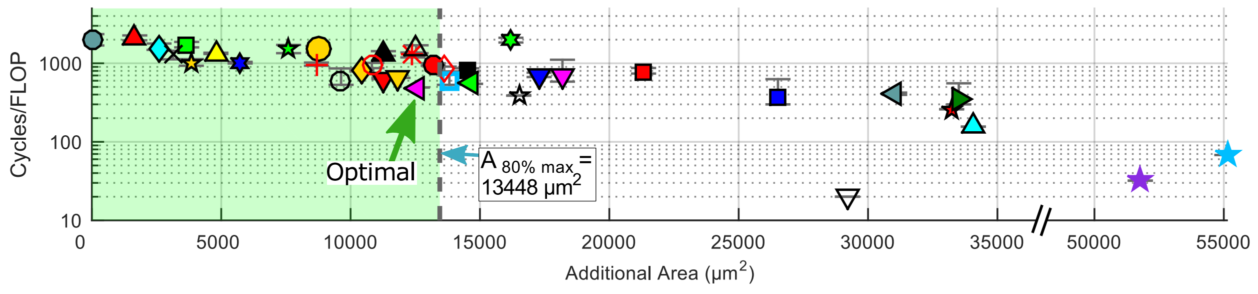
Figure 3.12: Square root area versus cycles per FLOP for all implementations and determining the optimal implementation from an area constraint. Four methods are evaluated for performing square root; the Newton-Raphson method, square root in software, square root in hardware, and hybrid square root. Design-point symbols are placed at the average cycles/FLOP point with interval bars showing the range over all possible values. Cycles per FLOP are scaled by the number of cores required. The optimal square root implementation is determined with the remaining available area, $A_{sqrt\ max}$. Designs satisfying the area constraint appear in the green highlighted region. Remaining legend shown in Figure 3.10. Results from synthesis in 65 nm CMOS at 1.3 V and 1.2 GHz.

many scientific workloads [85]. They are implemented using the minimum number of cores and the addition/subtraction and multiplication designs from Figure 3.10, and subject to the same area constraint. Similar trade-offs between the designs are seen in Figure 3.10 and Fig. 3.13. As denoted by the green arrow, *Hybrid Add/Sub w/ CFP Ver. 3* and *Hybrid Mult w/ CFP Ver. 2* remain the optimal addition/subtraction and multiplication designs, respectively. The full hardware designs provide the highest throughput but do not meet the area constraint. The policy of using cycles per FLOP and additional area for each FP design from Figure 3.9–3.12 can be employed to roughly estimate the performance and area requirements for computing other benchmarks.

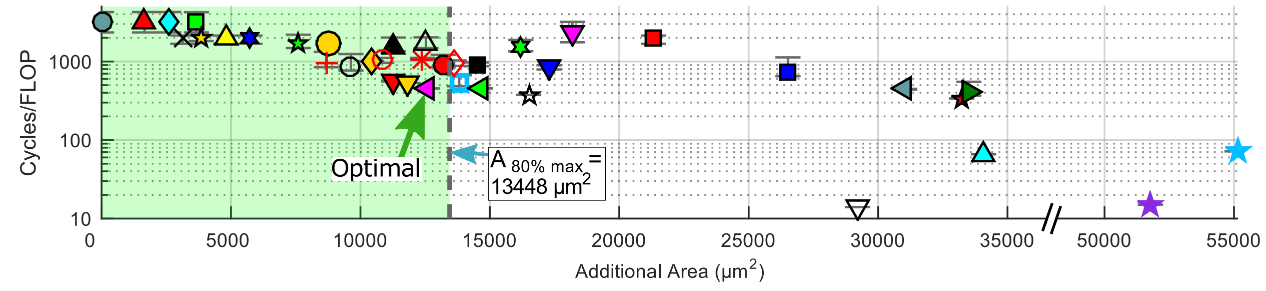
3.6 Advantages of Hybrid Approaches

The hybrid implementations with USL support provide unsigned hardware which allows efficient handling of multi-word values, improving Newton-Raphson throughput. The long-division and digit-by-digit methods see much less benefit, as they depend more on shifts.

Multiple hybrid designs w/ CFP are implemented to explore the benefits of different design approaches. Each version differs in terms of which steps or the proportion of the FP op-



(a) 2x2 matrix multiplication



(b) Radix-2 complex butterfly computation

Figure 3.13: Benchmark results for two scientific application kernels. (a) The benchmark results for calculating a 2x2 matrix multiplication, (b) the benchmark results for computing a radix-2 complex butterfly operation. Design-point symbols are placed at the average cycles/FLOP point with interval bars showing the range over all possible values. Cycles per FLOP are scaled by the number of cores required. Area constraints are indicated by the vertical dashed lines. The optimal design has the fewest average cycles per FLOP and an area below the area constraint. The optimal adder/subtractor and multiplier are determined using the same area constraint as Figure 3.10, $A_{80\%\ max}$. Designs satisfying the area constraint appear in the green highlighted region. The legend is shown in Figure 3.10. Results from synthesis in 65 nm CMOS at 1.3 V and 1.2 GHz.

eration that is performed in software. Which steps justify hardware support is based upon the throughput increase and area overhead. *Hybrid Add/Sub w/ CFP Ver. 3* increases addition/subtraction throughput the most by supporting operand comparison in hardware. Otherwise, sorting the operands requires many instructions to compare the exponents and the multi-word mantissa. *Hybrid Add/Sub w/ CFP Ver. 4* includes an LZA which increases throughput, but requires more area and improves throughput less than supporting operand sorting in hardware. For multiplication, *Hybrid Mult w/ CFP Ver. 2* increases throughput the most by adding more hardware support than *Ver. 1.* This implementation reduces the executed instruction count by calculating the sign bit, exponent, and determining if the result is zero in hardware. This additional circuitry increases area and is shown in Figure 3.7. The division and square root implementations use less area than dedicated FP hardware by performing sign bit and exponent calculation in software; the throughput of these operations is increased by performing the rest of the operations in hardware.

3.7 Related Work and Comparison

Since this work presents single-precision FP implementations, results are compared with other work that increases single-precision FP throughput with less area overhead than a dedicated hardware design, and do not compare with implementations using BFP or a reduced FP word width. Table 3.3 summarizes a comparison with other methods for improving FP throughput. The results include designs with a 16-bit and 32-bit word size and datapath, all implementing single-precision FP. Not every work reports area data; therefore, to make a consistent comparison, the area overhead of each design is evaluated against the area of the dedicated full hardware design reported in that respective work. This dissertation and two of the papers in the table explore alternatives to an FMA [32, 33], while one compares against an Altera FPU without divide [40]. This dissertation reports the area overhead for supporting each FP operation individually. Other work does not publish area for specific FP operations; therefore, area is recorded under the operation categories for which cycle counts are reported. Except for the FMA design, the implementations for multiply-add perform an unfused operation.

The work by Gilani et al. [32] and Viitanen et al. [33] did not explore modular designs. Hockert et al. [40] explored modular designs with varying amounts of hardware support, but did not evaluate the overhead for supporting individual FP operations.

My work presents a wider range of area overheads for improving FP throughput, allowing more versatility across a large range of area constraints. My designs also offer the lowest cycles per FLOP for both the divide and square root operations while requiring less area than an FMA. Comparing my 32-bit I/O designs to other work that reduces FP area overhead compared to dedicated FP hardware, my implementations achieve the lowest cycles per FLOP for all operation types.

Table 3.3: Comparison of Methods for Improving FP Throughput with Less Overhead.

	Clock Freq (MHz)	Process Node (nm)	Area Overhead Compared to Full HW Design* (%)					Cycles/FLOP				
			Multiply-Add	Add /Sub	Multiply	Divide	Square Root	Multiply-Add	Add /Sub	Multiply	Divide	Square Root
Gilani et al. [32]	1000	65	19.2†			N/A	N/A	6	5	5	N/A	N/A
Hockert and Compton. [40]	500	65	N/A	22.8–80.0†				N/A	77–118	109	218–238	289–313
Viitanen et al. [33]	300	110	N/A	N/A	N/A	13.3†		N/A	N/A	N/A	49	43
This Work‡ (16-bit I/O)	**1200**	65	**4.71**–61.8	**5.69**–24.7	**3.01**–33.0	**5.36**–11.3	**9.32**–61.8	13–186	7–57	6–57	**35**–254	**32**–1373
This Work‡ (32-bit I/O)	**1200**	65	56.4	20.1	33.3	**11.6**–56.4	**13.1**–56.4	**2**	**1**	**1**	**30**–1053	**26**–447

Results reported for alternatives to baseline implementations. Cycles per FLOP are scaled by the number of cores required.
This dissertation provides more options for improving FP throughput and a wider range of area overheads than previous work.
* Area overhead relative to FMA area reported in [32], [33], and this dissertation. Area overhead relative to Altera FPU without divide in [40].
† Total area to perform these FP operations reported.
‡ Results from synthesis in 65 nm CMOS at 1.3 V and 1.2 GHz.

Chapter 4

Area Efficient Synthetic Aperture Radar (SAR) Image Formation

SAR uses one antenna to time multiplex [98]. An antenna transmits a single pulse (known as a chirp with a series of increasing frequencies), and receives the reflected complex signal (which has both an amplitude and phase). Using the received signals while transitioning from point A to B, it is possible to form an image that would have required a stationary antenna of length D = (distance from A to B). A SAR image is produced from the radar reflectivity of a target scene. Using pulse echo timing, it is possible to perform ranging [99]. Each scatterer being imaged is projected onto the current radar line of sight in a range bin according to the distance from the antenna phase center [3]. This work uses a tomographic method for image reconstruction. Munson et al. presented a tomographic formulation of spotlight SAR [2], which covers the mathematical development of the projection-slice theorem used in tomographic image reconstruction.

4.1 Backprojection Algorithm Functional Blocks

A previously developed SAR signal model was used for the backprojection imaging algorithm [100]. Gorham et al. used this model to generate a backprojection imaging algorithm in MATLAB [77]. The functional blocks of the backprojection implementation are shown in Figure 4.1. For the data sets considered, the range to the motion compensation point was always near 10,000 m, and each coordinate of the antenna phase center ranged between 0–7000 m, conditions which are

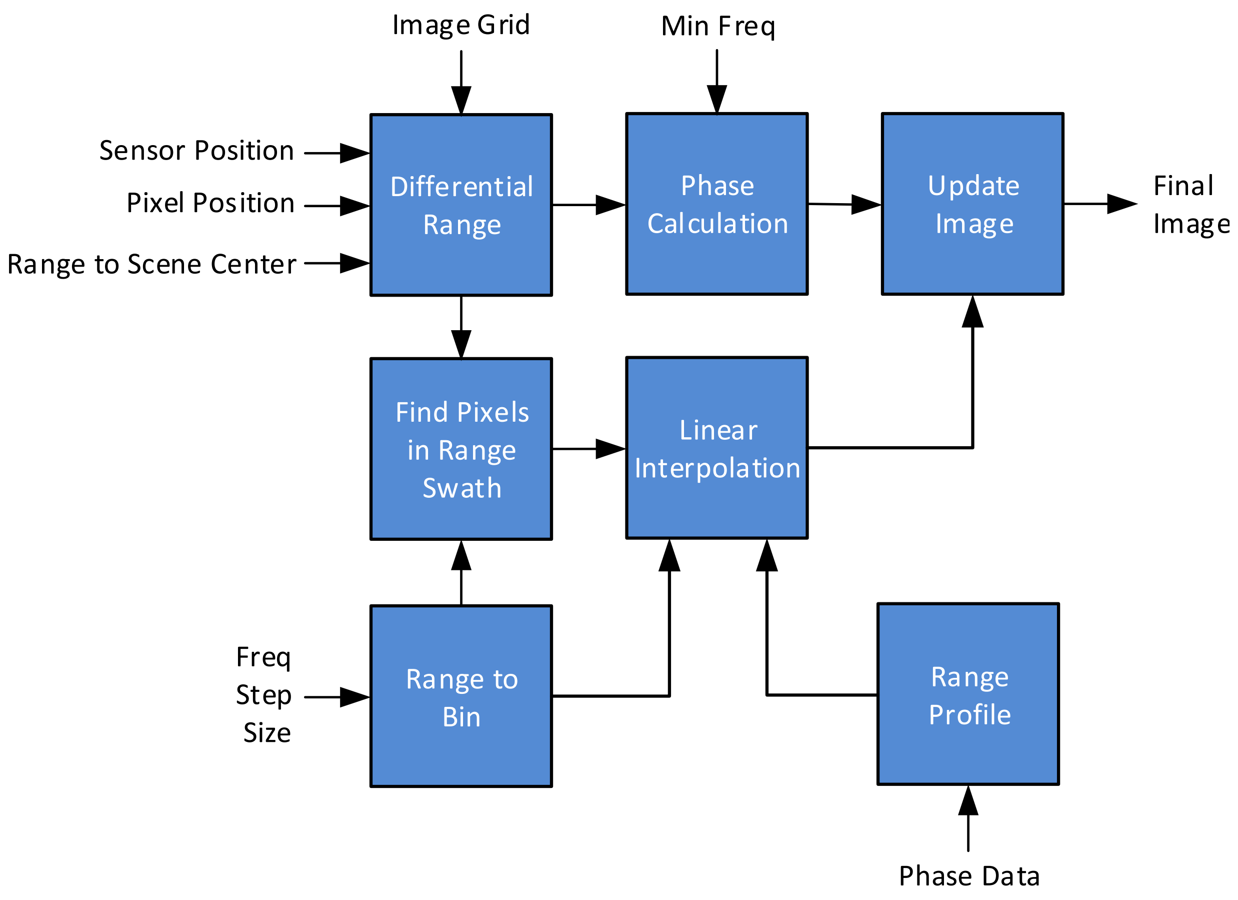


Figure 4.1: Block diagram of the seven major computation blocks of the SAR backprojection engine.

common with many airborne SAR systems [101]. Each functional block of the backprojection algorithm is described next.

4.1.1 Range Profile

The range profile is a one-dimensional profile of the scene. A single pulse, known as a chirp, with a series of increasing frequencies is transmitted. The reflected signal, or phase history data, is inversely Fourier transformed to form a range profile. The range profile is computed by zero padding the phase history data and computing the inverse fast Fourier transform (IFFT). The size of the IFFT was chosen as 4096 points to maintain computational efficiency of the IFFT, reduce the occurrence of artifacts, and produce a smoother image. The range profile data are then used during linear interpolation, along with the differential range and the range to each bin, to determine an interpolated value for the range profile.

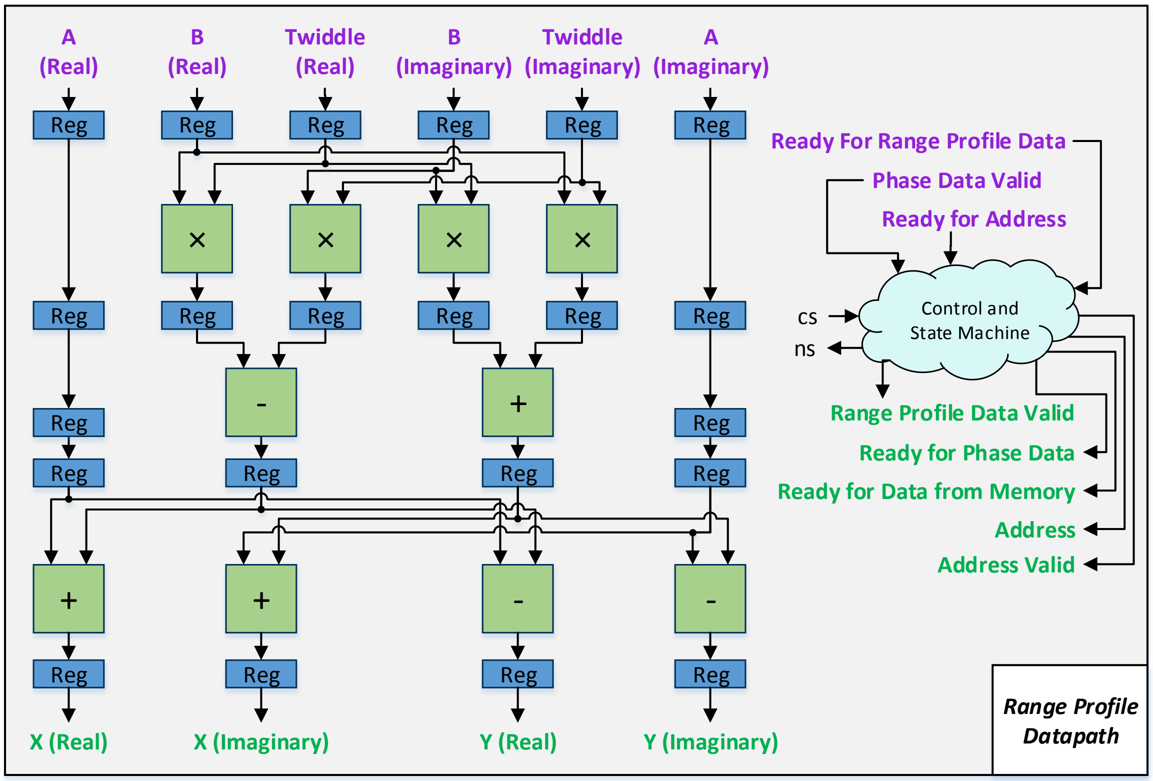


Figure 4.2: Datapath for one radix-2 butterfly for *Range Profile* functional block.

The areas for storing the phase history data, the computed transform, and twiddle factors (i.e. complex multipliers of the form W_N^r [102]) in memory are considered separately from the computation area. The computation area is calculated for performing one radix-2 butterfly at a time. As with the other functional blocks, the number of computation units can be replicated depending on the desired throughput; however, the area relationship between using different FP word widths remains the same. Figure 4.2 shows the datapath for one radix-2 butterfly.

The computational area is provided in Table 4.1. The area for data storage for the DP-FP implementation was $3\,060\,000\,\mu\mathrm{m}^2$. The minimum area for the data storage to obtain a structural similarity index metric (SSIM) greater than or equal to 0.5 was $680\,000\,\mu\mathrm{m}^2$. The minimum area to obtain an SSIM $\geq$ 0.9–0.99 was identical, and required $1\,020\,000\,\mu\mathrm{m}^2$. The memory area is estimated from a large on-chip shared memory fabricated in 65 nm CMOS [7], for a many-core processor array [59].

4.1.2 Range to Bin

The range to bin data are used for finding pixels that fall within the range swath and for linear interpolation. The values of these bins are evenly spaced along the scene range. This module uses a multiplication, addition/subtraction, and division unit, and is implemented using the non-restoring algorithm [49].

The datapath for the *Range to Bin* functional block is shown in Figure 4.3. The input frequency step size, Δf, is used to determine the maximum alias-free extent of the image, *maxWr*. This value is divided by the number of points in the IFFT and multiplied by the *nextindex* value to determine each value of the range to bin value. Shown at the top of the figure, the high and low index are saved so that the range to bin value can be incremented or decremented. This is especially useful for the *Linear Interpolation* block, and saves time when finding bounds for the differential range values. The *Linear Interpolation* block returns signals as to whether the differential range was within, above, or below the bounds so that the range value can be incremented or decremented appropriately. The minimum and maximum values of the range swath are sent to the *Find Pixels in Range Swath* block and the *Linear Interpolation* block. The *Range to Bin Table* block in Figure 4.3 is a parameterized module containing the start value, end value, increment amount, and the speed of light. The table is parameterized in order to provide these constants with the desired exponent and mantissa width, up to DP-FP. The enable signals for the registers are not shown, but are set in the control block.

4.1.3 Differential Range

The differential range is computed using the three-dimensional position of each pixel, the position of the sensor at each pulse, and the range to the scene center. The differential range is the difference between the distance from the antenna phase center to the scatterer, and the distance from antenna phase center to the origin. This block uses five addition/subtraction, three multiplication, and one square root unit(s), and is implemented using the non-restoring algorithm [49]. The differential range is computed to determine phase correction for each scatterer located at each pixel location. It is also passed to the *Linear Interpolation* block. Given the range to each frequency bin and the range profile data, the *Linear Interpolation* block interpolates the range profile for each

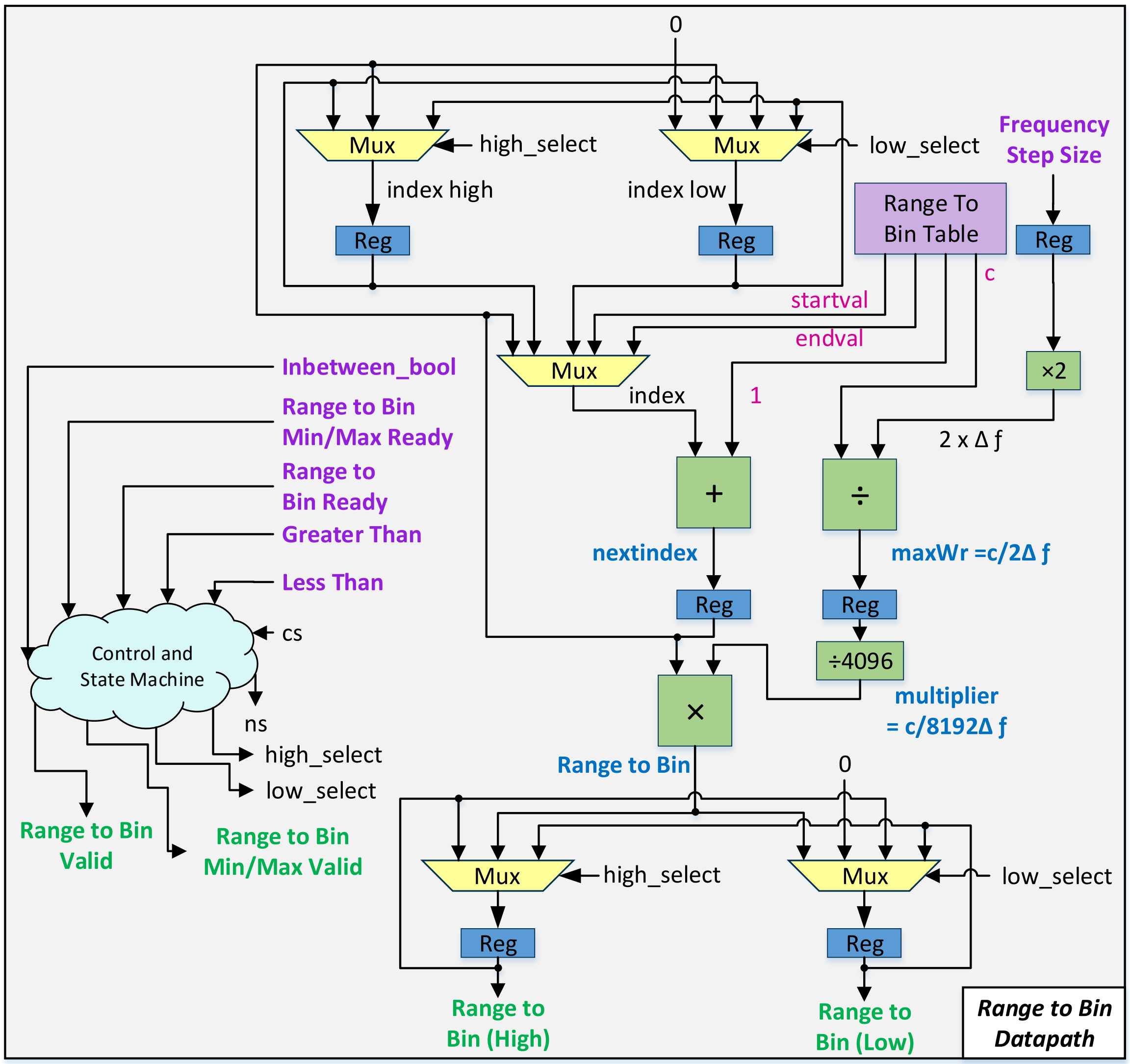


Figure 4.3: Datapath for *Range to Bin* functional block.

differential range value.

The datapath for the *Differential Range* block is shown in Figure 4.4. The pixel position and antenna position are input into the differential range block and the distance between the two are calculated. This is done by subtracting each x, y, and z point, and then squaring these values. After that the square root of their sum is taken. The resultant value is the distance from the antenna to the scatterer. Next the range to the scene center is subtracted to get the differential range.

4.1.4 Phase Calculation

The linear interpolation block can interpolate values for the range profile, but the phase for each differential range value must also be computed. Determining the phase correction for the receiver output relies on the minimum frequency of the received samples and the differential range [77]. This calculation involves a complex exponential function which can be transformed into equivalent trigonometric functions. Figure 4.5 shows the datapath for the *Phase Calculation* block. This block consists of four major components: range reduction, generation of the powers of the variable x, cosine calculation, and sine calculation.

First, inputs to the *Phase Calculation* block are range reduced into a smaller interval (i.e., $[-\pi/4, \pi/4]$) [103], then the sine and cosine are computed using polynomial approximations for values within this interval [104]. In order to pipeline the computation of the polynomial approximations for sine and cosine, this module uses 29 multiplication units and 14 addition/subtraction units. The phase correction is then applied to the data following linear interpolation.

4.1.5 Find Pixels in Range Swath

Figure 4.6 shows the datapath for this functional block. Using the minimum and maximum values of the range swath and the differential range values, this module uses a comparison unit to determine which pixels are within the alias free region, also known as the range swath. The range swath is determined by the frequency step size, Δf. The *Range to Bin* functional block determines the bounds of the range swath. If the differential range is within the bounds of the range swath, it is output, otherwise a zero is output. The data is sent to the *Linear Interpolation* block for interpolating the range profile. Interpolation is performed on the pixels whose differential range values lie within the range swath.

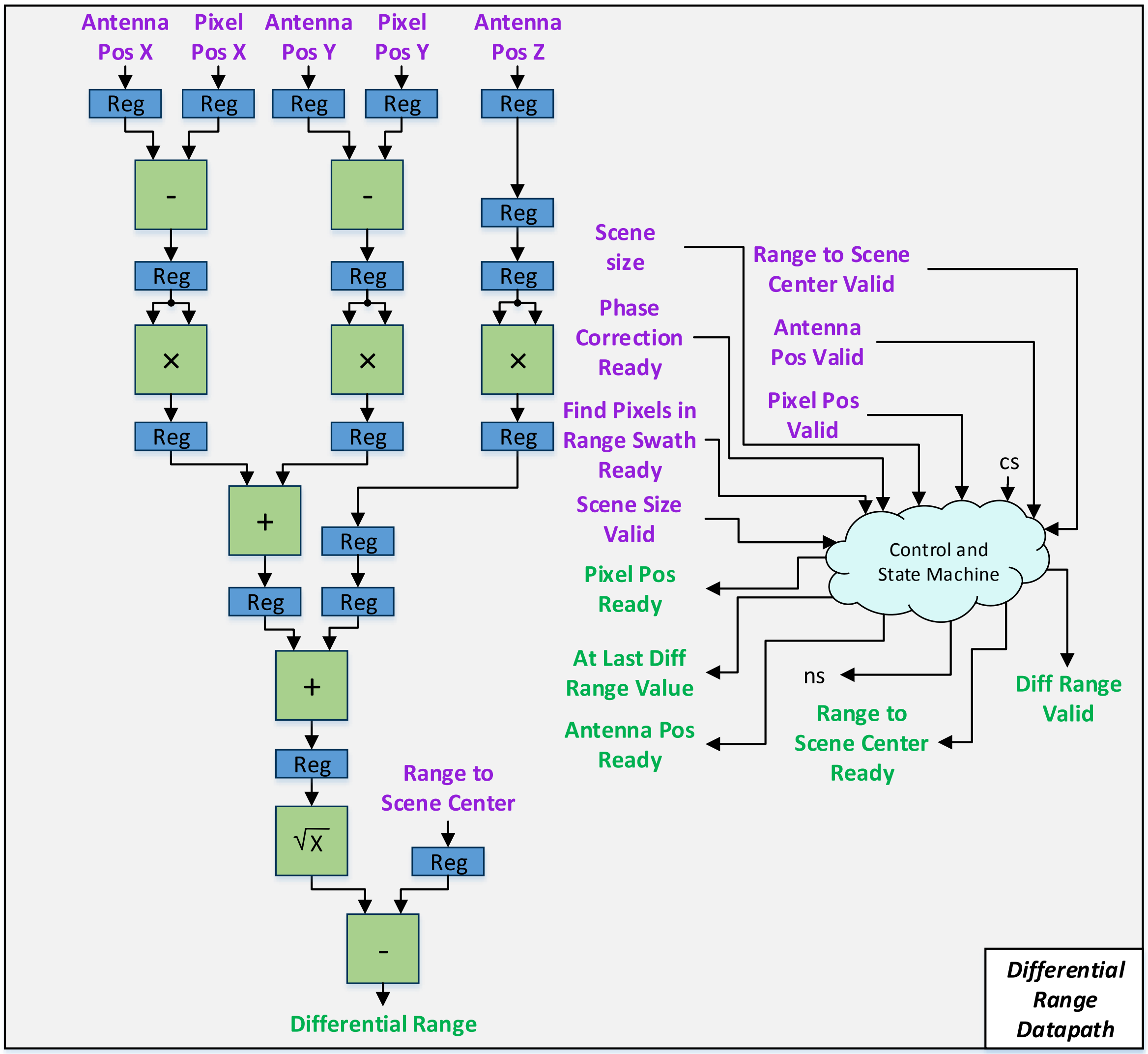


Figure 4.4: Datapath for *Differential Range* functional block.

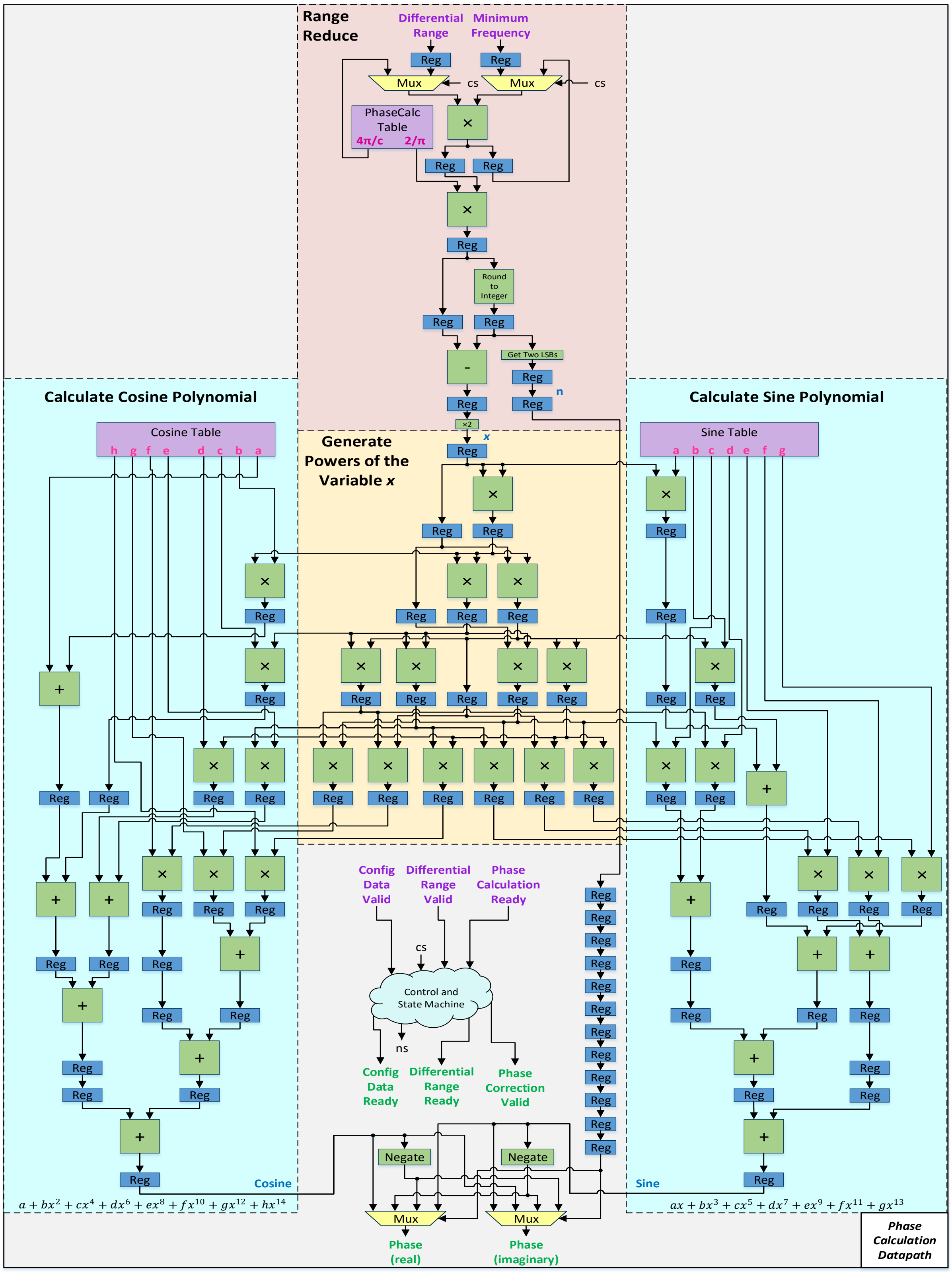


Figure 4.5: Datapath for *Phase Calculation* functional block with major components highlighted.

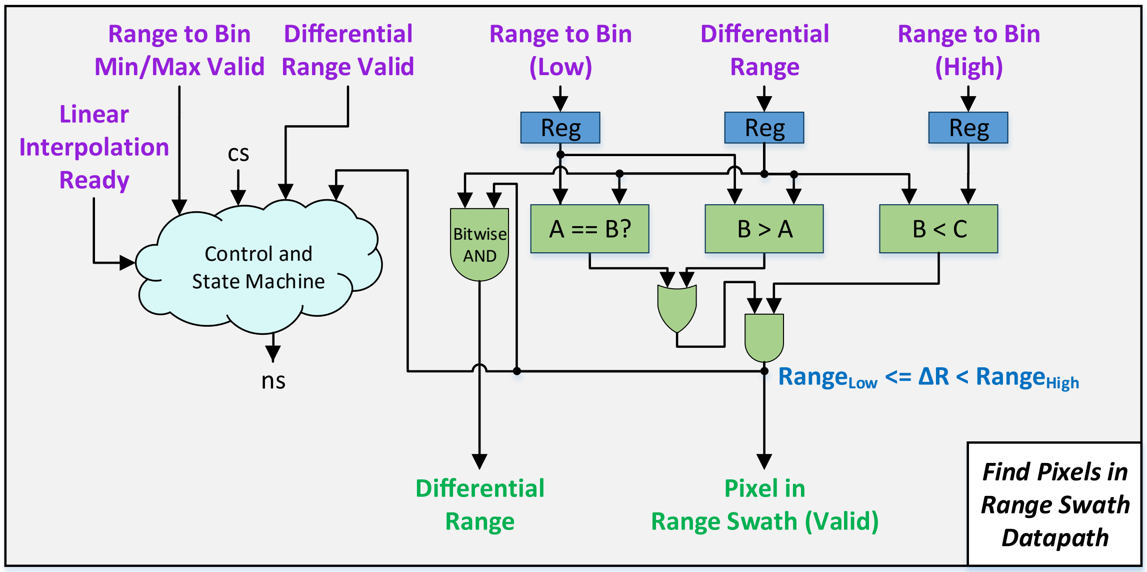

Range to Bin Min/Max Valid
Differential Range Valid
Range to Bin (Low)
Differential Range
Range to Bin (High)
Linear Interpolation Ready
cs
Reg
Reg
Reg
Control and State Machine
Bitwise AND
A == B?
B > A
B < C
ns
Range_Low <= ΔR < Range_High
Differential Range
Pixel in Range Swath (Valid)
Find Pixels in Range Swath Datapath

Figure 4.6: Datapath for *Find Pixels in Range Swath* functional block.

4.1.6 Linear Interpolation

Following calculation of the range profile, the range to each bin in the range profile, and the pixels in the range swath, a linear interpolation operation is performed since the values of the differential range do not exactly line up with the discrete range to bin values [77]. This linear interpolation determines the response from each pixel location.

If the differential range is not within the lower and upper range to bin value, then a request for the next range to bin values is sent to the *Range to Bin* functional block, depending if the bounds need to be changed. If the value is within the range, then the linear interpolation proceeds and the output is sent to the *Image Update* block, where phase correction is applied.

Figure 4.7 shows the datapath for the *Linear Interpolation* functional block. This module uses one division, five addition/subtraction, and four multiplication modules. This number of functional blocks allows simultaneous processing of the real and imaginary data.

4.1.7 Image Update

As the aircraft acquires phase history data along its flight path, a new image can continuously be formed from each pulse. These images are added together to resolve the target scene. After

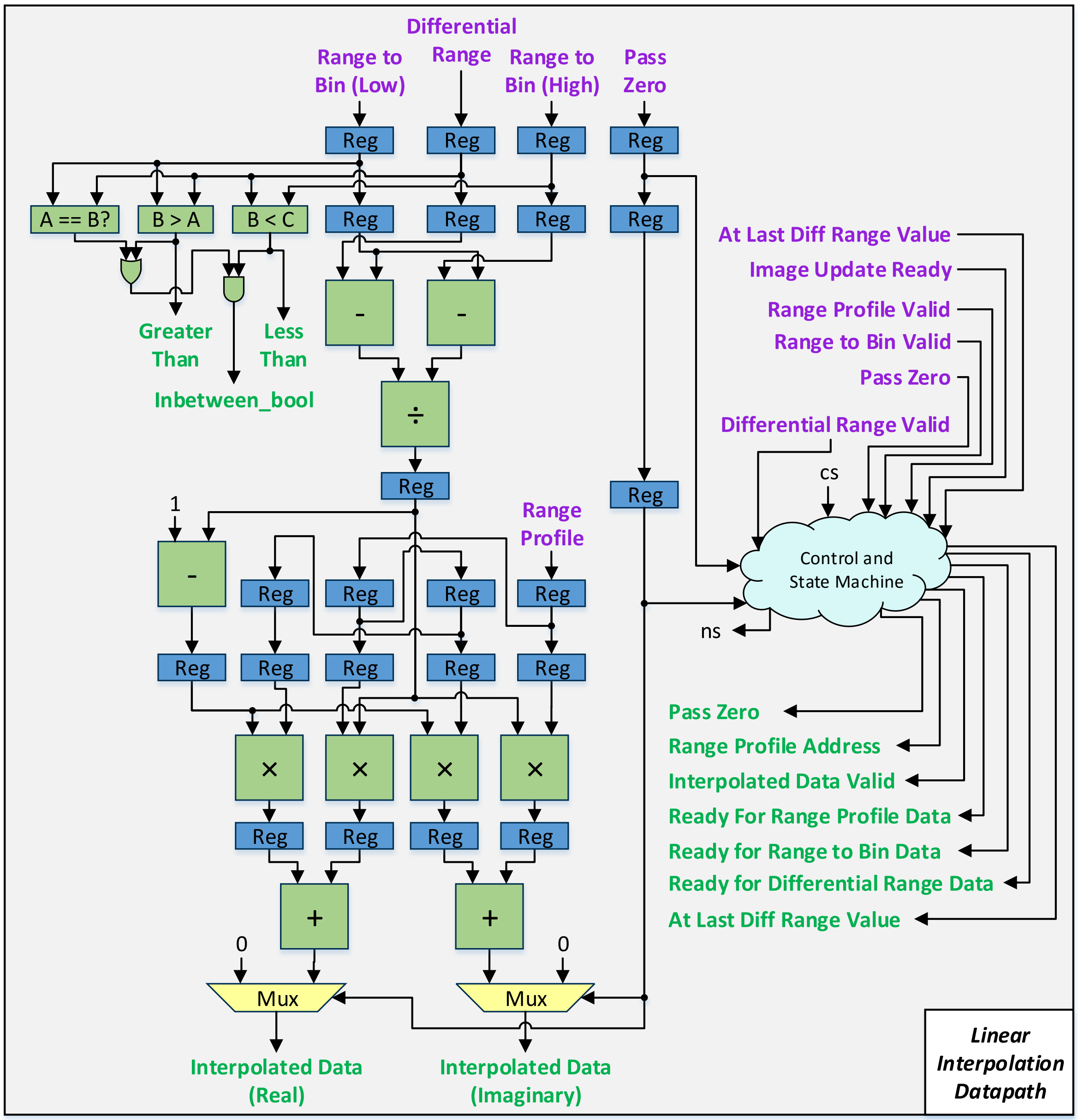


Figure 4.7: Datapath for *Linear Interpolation* functional block.

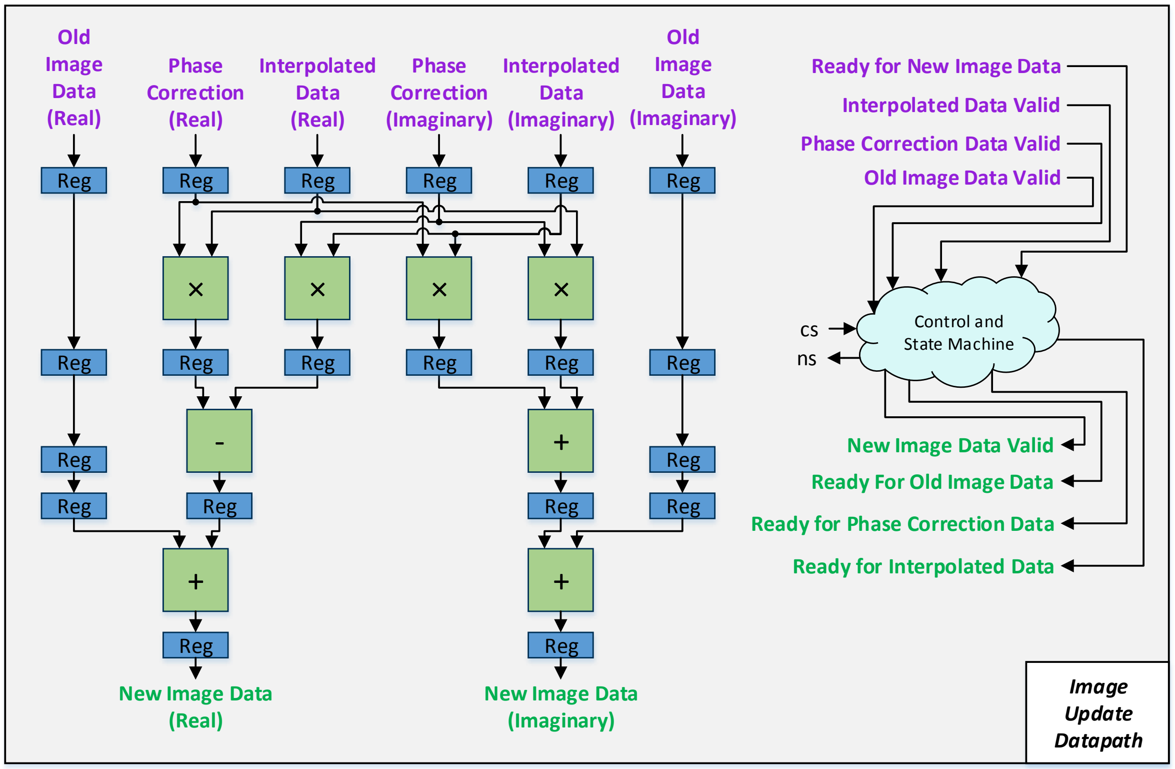


Figure 4.8: Datapath for *Image Update* functional block.

phase correction, the current image data are added to the old image data from the previous pulse to form the new image data. The final image response is the summation of the image response for every pulse.

Following the linear interpolation step, phase correction is applied to the data using a complex number multiplication and then the image responses are summed for each pulse to create the final image data. This module utilizes four multiplication modules and four addition/subtraction modules. The datapath for this functional block is shown in Figure 4.8.

4.2 Methods for Reducing Floating-Point Word Width and Determining Area for Backprojection

Each functional block of the backprojection algorithm is written in either C++ or MATLAB. Each program is parameterized to support computations at any exponent and mantissa width up to

DP-FP. An example of a 64-bit DP-FP number is shown in Figure 4.9. Modifying the exponent and mantissa width allows control over the available dynamic range and precision for each FP format. The IEEE-754 default rounding mode, round-to-nearest, is used for all computations [41]. The dependency on dynamic range is evaluated by modifying the exponent width to between 1 and 11 bits. The effects of precision on image quality are explored by modifying the mantissa width to be between 1 and 52 bits after the decimal point. Before modifying the mantissa width, the minimum exponent width needed to accommodate the dynamic range is determined.

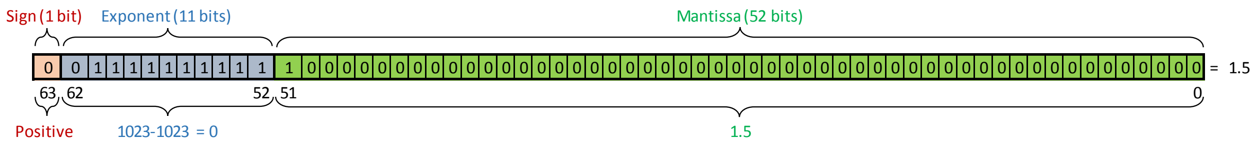


Figure 4.9: Example IEEE-754 double-precision number.

To obtain gold-standard images to measure against, each functional block is first configured to perform computations using DP-FP arithmetic. The backprojection algorithm is then performed on all data sets to form each gold-standard image. The peak signal-to-noise ratio (PSNR) and SSIM index are used to quantify the quality of images formed using each FP word width configuration [105], and each new image is measured against the gold-standard image. PSNR measures the ratio between the maximum signal power and the noise corrupting the image; however, this metric does not map well with the human visual system. SSIM is used in addition to PSNR because it is based on the notion that human visual perception is adapted for extracting structural information about an image.

Figure 4.10 plots PSNR and SSIM versus exponent width for each functional block. The *Range Profile* and *Differential Range* functional blocks require the largest exponent bit-widths due to the dynamic range of the data input to these blocks. The *Find Pixels in Range Swath, Linear Interpolation*, and *Image Update* functional blocks require the smallest exponent widths.

To determine the area requirements at each exponent and mantissa width, each functional unit is written in Verilog RTL and synthesized in 65 nm CMOS at 1.3 V and a clock frequency of 1.2 GHz. Each module is parameterized to support any FP word width, without having to perform a redesign for each exponent and mantissa width. While it is also possible to reduce multiplier area and delay through the use of a Dadda or Wallace multiplier, choosing the optimal design is dependent

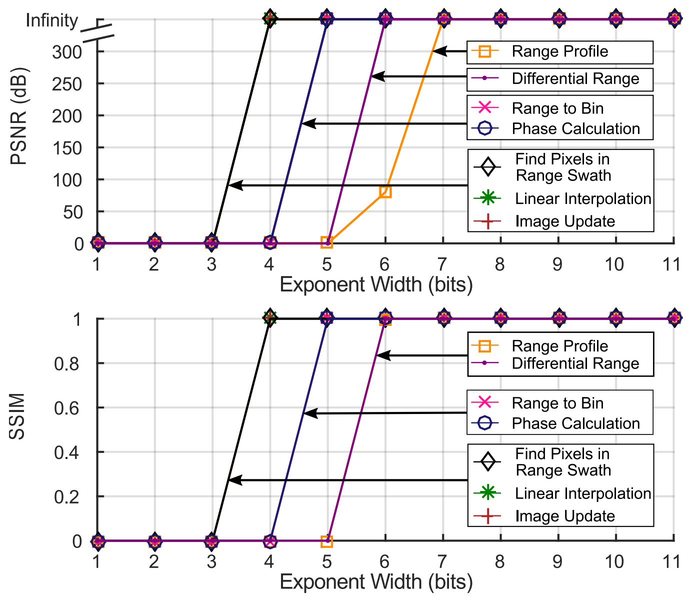


Figure 4.10: PSNR (upper plot) and SSIM (lower plot) of resulting images versus the exponent widths of the seven functional blocks used to compute those images. The values of SSIM and PSNR are used for determining the minimum exponent widths for each functional block. Data are determined by the worst-case PSNR and SSIM across the three data sets. The mantissa width is kept at 52 bits.

on multiplier width [106]. However, Design Compiler will select the appropriate architectures for the adder [107] and multiplier [108] based on the design constraints. Each functional block is pipelined. Since each functional block has a different pipeline depth, this results in different numbers of pixels that can be operated on simultaneously. Therefore, area percentages are made relative to the same functional block. To achieve a throughput requirement, the computational units can simply be replicated and the area relationship between using different FP word widths will remain the same.

4.3 Comparison of Image Quality with Reduced Floating-Point Word Widths

For each image quality comparison, the mantissa and exponent width for one of the seven functional blocks is reduced from DP-FP. DP arithmetic is then utilized for the remaining six functional blocks. Each data set is processed to form an image. The images formed when modifying the FP word width of each functional block are then evaluated against the gold-standard image. Images are saved as gray level (8-bit) jpegs using lossless compression before comparison.

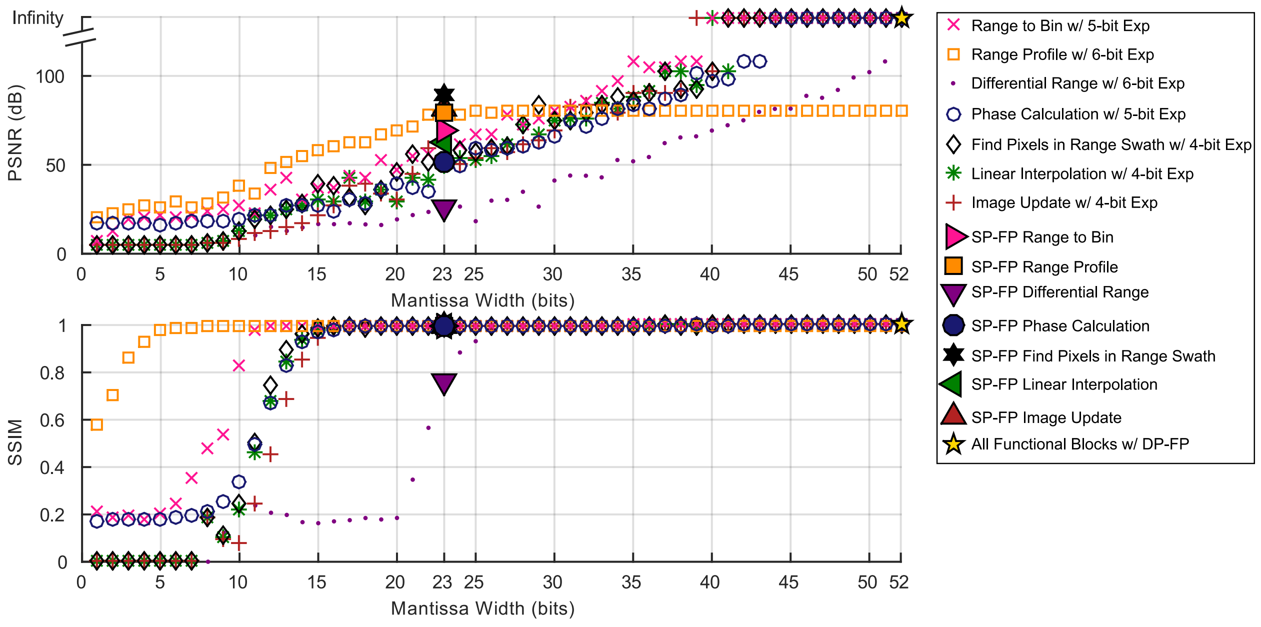


Figure 4.11: PSNR (upper plot) and SSIM (lower plot) of resulting images versus the mantissa widths of the seven functional blocks used to compute those images. Images are measured against images formed using double-precision floating-point (DP-FP) and single-precision floating-point (SP-FP) arithmetic. Data are determined by the worst-case PSNR and SSIM across the three data sets. The exponent width is chosen to satisfy the dynamic range requirement of all data sets as shown in Figure 4.10. All functional blocks using DP arithmetic are denoted by the gold star symbol. Functional blocks using SP-FP arithmetic have a mantissa width of 23 bits.

Figure 4.11 plots the worst-case PSNR and SSIM for images constructed using the three data sets while varying the mantissa width. PSNR and SSIM versus mantissa width are plotted in the upper and lower figure, respectively. All mantissa widths between 1 and 52 bits are traversed. The exponent width of each functional block is chosen to satisfy the dynamic range requirements of all data sets as shown in Figure 4.10. The results when using SP-FP and DP-FP arithmetic are also denoted.

Although PSNR is a commonly used image quality metric, SSIM proves to be more useful for determining human perceived image quality. Among the data sets utilized, images appearing identical can differ in PSNR value by as much as 80 dB. Alternatively, no visibly detectable differences are found between the gold-standard images and images formed using a reduced FP word width when the SSIM is ~0.95 or higher. Therefore, images produced with an SSIM ≥ 0.99 are considered indiscernible from the gold-standard images. After achieving a value ≥ 0.99, the SSIM asymptotically approaches a value of 1 without visibly improving image quality, therefore the additional FP word width and hardware are unnecessary.

The results shown in Table 4.1 demonstrate that the mantissa width requirements for each functional block range between 6–27 bits to form an image with an SSIM $\geq$ 0.99. These reductions in widths amount to average area savings of 75.5%. The largest area savings are obtained by reducing mantissa width, rather than exponent width.

The range profile functional block has the largest potential area savings. It is possible to reduce the FP word width for this block from DP-FP to a format using a 6-bit exponent and 6-bit mantissa and obtain a resulting SSIM value of 0.99. This reduction amounts to an area savings of 91.2%. Conversely, the differential range block required the largest mantissa width. To achieve an SSIM of 0.99, a mantissa width of 27 bits is required; reducing area by 48.4%.

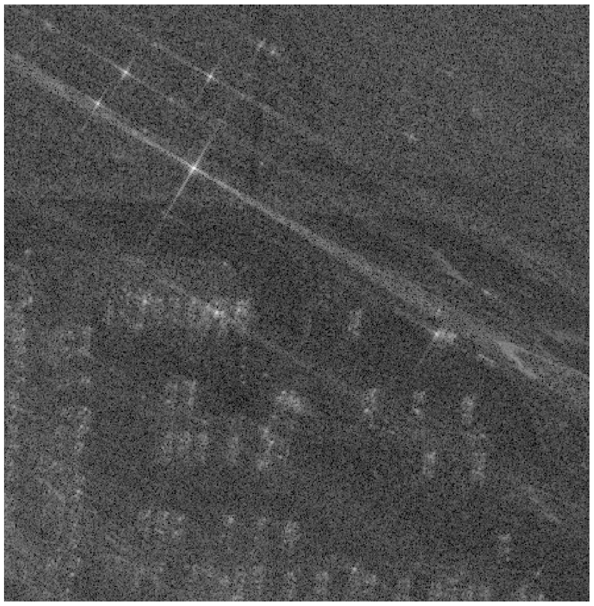
The image quality produced when reducing the exponent and mantissa width of all functional blocks simultaneously is also considered. Figure 4.12 shows four images formed using the volumetric data set. Each image is examined against Figure 4.12d, which is the gold-standard image created by using DP-FP arithmetic for each functional block. For Figure 4.12a–4.12c each functional block is configured to use the exponent and mantissa widths shown in Table 4.1 to achieve an SSIM of 0.5, 0.9, and 0.99, respectively. For Figure 4.12a, although each functional block is configured to achieve an SSIM of 0.5, the image produced when using these reduced widths together forms a visibly degraded image with an SSIM of 0.42. However, for Figure 4.12b and Figure 4.12c each functional block is configured to achieve an SSIM of 0.9 and 0.99, respectively, and the image quality is not visibly different from Figure 4.12d. Similar results are observed for the other data sets in which using the settings for achieving SSIM values $\geq$ 0.9 maintained image quality.

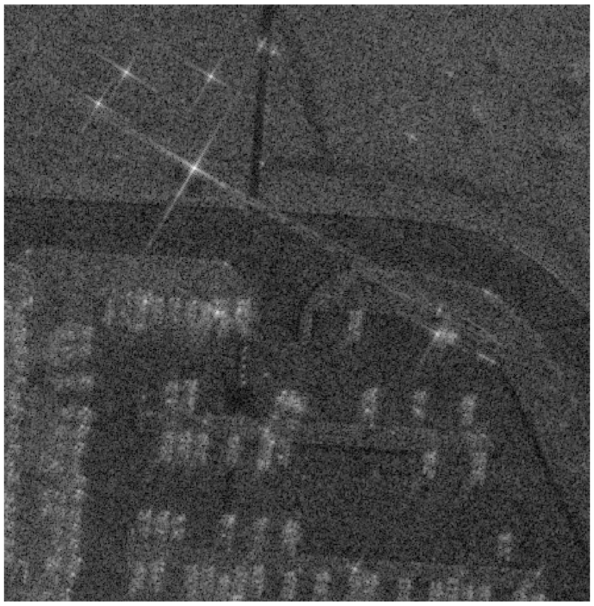
Table 4.1: Area and Floating-Point Word Widths Required for Various SSIM Values for Each Functional Block Measured Against Single-Precision and Double-Precision Arithmetic.

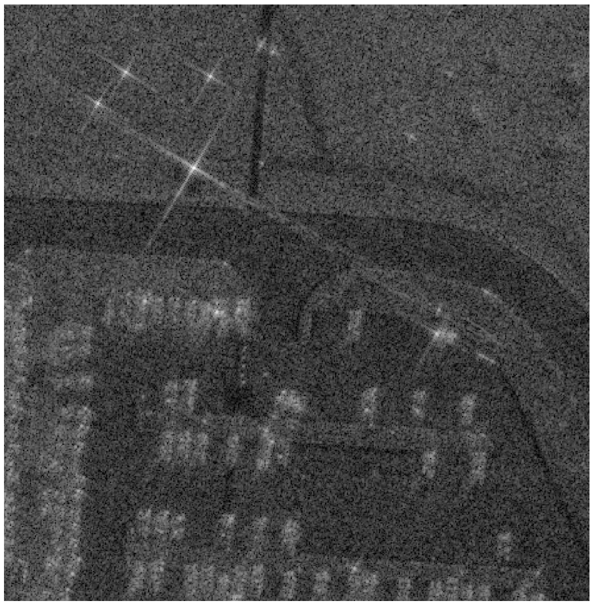
	Functional Block						
	Range to Bin	Range Profile	Differential Range	Phase Calculation	Find Pixels in Range Swath	Linear Interpolation	Image Update
Area w/ Single Precision Arithmetic (µm^2)	23365	102409	92708	552409	2469	108609	95212
Area w/ Double Precision Arithmetic (µm^2)	66040	250694	214185	1754742	5171	273014	257678
Exponent Width used for All Optimized Designs*	5	6	6	5	4	4	4
Minimum Mantissa Width for SSIM $\geq$ 0.5	9	1	22	12	11	12	13
Minimum Mantissa Width for SSIM $\geq$ 0.9	11	4	25	14	14	14	15
Minimum Mantissa Width for SSIM $\geq$ 0.99	12	6	27	17	16	17	17
Minimum Area (µm^2) for SSIM $\geq$ 0.5 (% Area of DP-FP)	8185 (12.4%)	4415 (1.8%)	82255 (38.4%)	221803 (12.6%)	1126 (21.8%)	47390 (17.4%)	43128 (16.7%)
Minimum Area (µm^2) for SSIM $\geq$ 0.9 (% Area of DP-FP)	10270 **(15.6%)**	16422 **(6.6%)**	93361 **(43.6%)**	249950 **(14.2%)**	1299 **(25.1%)**	51998 **(19.0%)**	52058 **(20.2%)**
Minimum Area (µm^2) for SSIM $\geq$ 0.99 (% Area of DP-FP)	10716 **(16.2%)**	22080 **(8.8%)**	110425 **(51.6%)**	354374 **(20.2%)**	1438 **(27.8%)**	66287 **(24.3%)**	57973 **(22.5%)**

Results based on synthesis in 65 nm CMOS with a supply voltage of 1.3 V at 1.2 GHz and evaluated against the images formed when using DP-FP arithmetic. SSIM values are shown for reducing mantissa width of only the given block while all other blocks have 52-bits (DP-FP).

* These values are the smallest exponent word widths which satisfy the dynamic range requirement as shown in Figure 4.10.


(a) Individual block SSIM = 0.5
Combined SSIM = 0.42


(b) Individual block SSIM = 0.9
Combined SSIM = 0.93


(c) Individual block SSIM = 0.99
Combined SSIM = 0.98

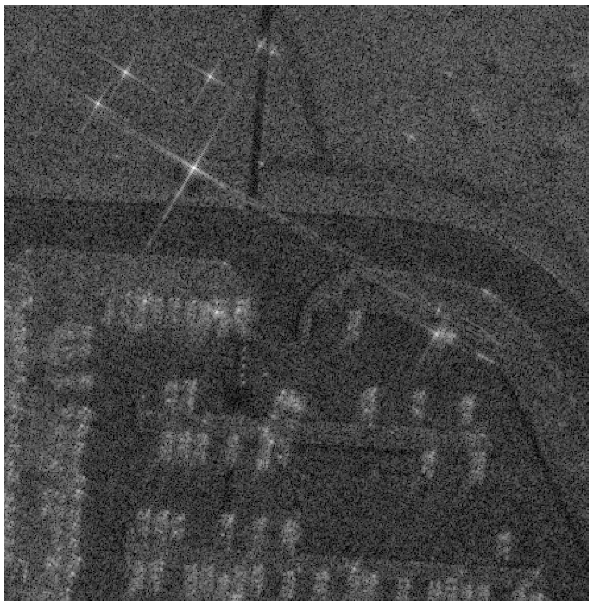

(d) Gold standard using DP-FP
Combined SSIM = 1

Figure 4.12: Images formed using the backprojection algorithm and the volumetric data set. An integration angle of 4° centered at 60° azimuth is used. Each functional block is connected together and configured to achieve a specific SSIM value. (a) Image formed using widths for each functional block to provide at least SSIM = 0.5. The SSIM of the image degraded to 0.42. For (b) and (c), configuring each functional block to provide a SSIM = 0.9 and SSIM = 0.99 does not visibly degrade the final image quality. (d) Gold-standard image created using DP-FP arithmetic.

Chapter 5

Design of Many-Core Platforms

In this chapter details of the physical design of many-core platforms in 32 nm PD-SOI CMOS are presented. Section 5.1 presents the first chip designed, KiloCore, and Section 5.2 presents the second chip designed, KiloCore2. For KiloCore, the testing process and measured results are provided, while KiloCore2 is awaiting testing. The results of this work are summarized in Chapter 7.

5.1 KiloCore

This section discusses the KiloCore chip, which was taped-out on March 4th, 2014 using 32 nm PD-SOI CMOS technology [8]. KiloCore was designed by eight researchers, including myself. I worked primarily on the physical design of the chip, developing synthesis constraints and synthesizing designs, and power and timing analysis. The chip features 1000 cores arranged in a 31 x 32 processor array configuration with an additional row of 8 cores at the bottom of the array [10], and has 621 million transistors. KiloCore also features twelve 64 KB shared static random-access memory (SRAM) blocks on the chip which are placed on the periphery. The nominal voltage is 0.9 V but the core voltage can range from 0.7 V–1.05 V [109].

5.1.1 Design Process

The physical design flow process was worked on primarily by Aaron Stillmaker (lead physical designer) and myself, and started with the RTL description of the circuit or macro block (e.g., oscillator, processor, independent memory). The architect was Brent Bohnenstiehl, who wrote a majority of the RTL as well. The functionality of the circuit was simulated using Cadence NC-Verilog. Once the design passed functional verification, I used Synopsys Design Compiler to convert the RTL into gates from the 32 nm standard cell library. Synthesis takes into account the timing and area constraints defined for the circuit to select the most appropriate gates. I developed the design constraints list and wrote a separate synthesis script for each design as there were different timing paths and constraints considered for each macro block. Synthesis produced a gate netlist that was then used for placement and routing of the design in Cadence Encounter. With Encounter, the floor planning, power gridding, placement, clock tree synthesis, routing, and optimization were performed by the lead physical designer. Similar to synthesis, a separate layout script was written for each macro block. Once a GDS file was produced by Encounter, one of the researchers would run design rule check (DRC) and layout versus schematic (LVS) using Mentor Graphics Caliber using the exported GDS and gate netlist. DRC ensured that the chip layout did not violate any of the foundry's rules for their 32 nm fabrication process and LVS verified that the physical chip layout matched the gate level netlist.

Once the macro block passed DRC and LVS, it was instantiated and placed into an array in Encounter by the lead physical designer, where inter-processor and inter-memory communication was routed. When DRC violations or LVS mismatches occurred, the issues were remedied by myself or one of the other researchers, and resolved in Encounter before reexporting another GDS and netlist file. Bin Liu then performed RC extraction on the macro blocks and ran FastSPICE simulations on them using Cadence Ultrasim to verify correct functionality. After all of the macro blocks had been placed and routed and verified, the final elements of the chip were added, namely input/output (I/O) drivers, electrostatic discharge (ESD) clamps to protect the chip during fabrication, crack stop and guard ring structures to prevent crack propagation during the wafer dicing process, and large decoupling capacitors (i.e., deep trench capacitors) for reducing supply noise on the power rails. I added the crack stop and guard ring structures using Virtuoso. Timing analysis was performed by me on the whole chip using Synopsys PrimeTime. The design was also verified by Bin by performing FastSPICE simulations using Ultrasim. The full chip design was run through Calibre to perform DRC and LVS one last time.

5.1.1.1 Oscillators

The oscillators were designed by Bin Liu by handpicking gates from the 32 nm standard cell library. Three different designs of the ring oscillator were made for KiloCore, two for the processor and one for the router. The independent memories use the same oscillator as the processor. For the processors, a major ring and microWatt ring oscillator were designed. The major ring oscillator offers a wider range of clock frequency operation, whereas the microWatt oscillator is intended for low power operation. Each oscillator design also contained some amount of control logic. I synthesized the oscillator control logic using Design Compiler. I blackboxed the major and microWatt ring during synthesis so the combinational loops they contain were not optimized out from the final gate netlist. After obtaining a netlist for the control logic and the major and microWatt ring oscillators, these were placed and routed in the processor tile by the lead physical designer. To ensure no wire routing from the processor would interfere with the oscillator, I determined how to fence off the oscillator during place and routing in Encounter. Additionally, the oscillators were set to "don't touch" so that Encounter wouldn't attempt optimization on the manually selected gates.

Based on SPICE simulations, the major ring oscillator had an expected working range of 1.54–3.47 GHz, while the microWatt ring oscillator had an expected working range from 43–910 MHz. The ring oscillators for the routers use the same design as the microWatt ring oscillators except with a different selection of standard cells in the delay lines; thereby providing 500 MHz, 1 GHz, 1.8 GHz, and 2.8 GHz according to SPICE simulation results. The SPICE simulations utilized only an HDL description of the oscillator, as it was originally assumed that wire load wouldn't have a significant effect on oscillator performance. This assumption was incorrect and led to undersized gates with poor drive strength, thereby limiting the performance of the fabricated chip to an average maximum clock frequency of 1.78 GHz for the processors, 1.77 GHz for the independent memories, and 1.49 GHz for the packet routers at 1.1 V. Therefore, the oscillator design process was modified for KiloCore2 by performing RC extraction before running Ultrasim FastSPICE simulations in order to consider the wireload. This process is discussed in Section 5.2.1.1.

5.1.1.2 Macro Blocks

As mentioned earlier, the full chip contains a set of macro blocks that were synthesized and placed and routed separately before being instantiated in the full chip layout. This allows easier control over the placement and sizing of these blocks and also makes the place and routing tasks manageable for the Encounter Digital Implementation (EDI) tool. These macro blocks include a single processor and an independent 64 KB memory block.

To avoid issues with the combinational blocks in the oscillators, I blackboxed the oscillators when synthesizing the macro blocks. The hand selected gates for the oscillators, and the synthesis produced netlists for the oscillator control logic, and the macro blocks were input to Encounter for placement and routing. Following place and route, a GDS and netlist were produced. Bin extracted an RC netlist and verified the DRC and LVS using Calibre. The extracted RC model was used for FastSPICE simulations to verify the functionality of the macro blocks. Once theses blocks were instantiated in the full chip layout, wire routing was performed.

5.1.1.3 Chip Level

The chip level of the design consists of the entire processor array, the independent memory array, and I/O drivers. Additionally, the foundry required several macro blocks of their own for testing their process during fabrication and these were included in the chip level layout. When synthesizing at the chip level, I blackboxed the macro blocks are blackboxed since they were placed and routed separately. The synthesis produced gate netlist was then imported into Encounter along with the already placed and routed macro blocks. Wire routing was then performed between the macro blocks and the chip level circuitry.

5.1.1.4 Chip Finishing

After a GDS was exported from Encounter by the lead physical designer, I added the crack stop, logo, and guard ring chip finishing features using Cadence Virtuoso. A picture of the logo I designed is shown in Figure 5.1. The logo features the name of the chip, the research group acronym, and the first initial of each contributing researcher's surname. I exported a final GDS from Virtuoso and sent it to the foundry for chip fabrication.

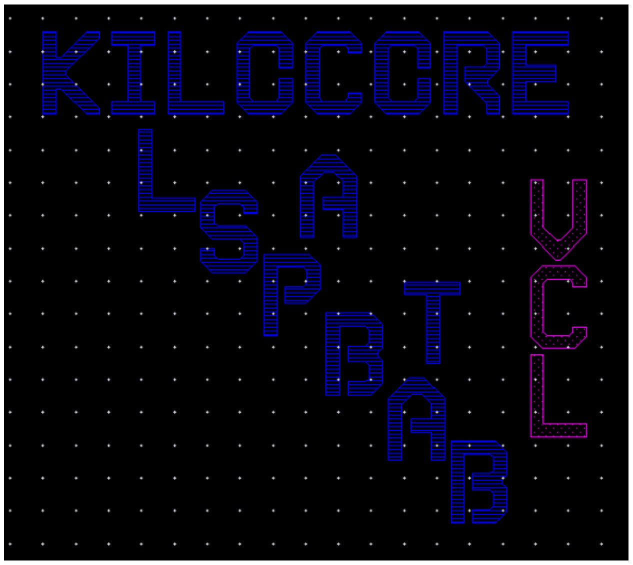


Figure 5.1: KiloCore logo.

5.1.1.5 Chip Package

Given that there was a lack of funding for a custom package, KiloCore uses an existing package design for a much smaller chip. The pads on the chip package only cover the center of the chip. This leads to a voltage drop across the on-die power rails when distributing power to the cores on the periphery. The package has 138 VDD pins, which are rated for supplying a maximum current of 200 mA; therefore, it can safely supply 24.8 W at 0.9 V and 30.4 W at 1.1 V. There are 29 VDD_{180} pins for powering the I/O drivers at 1.8 V. I calculated the capacitance of the PCB traces and the number of I/O drivers needed when estimating the number of pins required for powering the I/O drivers. I estimated that 19 pins were necessary to power both the LVDS and single-ended I/O drivers, therefore this package is capable of powering the I/O drivers. Additional funding was provided for custom chip packaging for KiloCore2 and therefore the power and I/O requirements for that chip were calculated to design a custom package as discussed in Section 5.2.1.6.

5.1.1.6 Inter-processor Timing

When routing all 1000 processors the EDI tool could not handle automatic routing between the cores and the process never finished executing. As a result, routing for inter-processor communication was not optimized with the EDI tool. As there were 34 days between full access to the design libraries and tape-out, there wasn't sufficient time in the schedule to perform static timing analysis before tape-out. Following tape-out, I performed static timing analysis using Synopsys PrimeTime on a 4x4 array with 2 independent memories as shown in Figure 5.2. The inter-processor communication timing was evaluated by analyzing links between a processor and its four nearest neighbors, as well as long distance communication between cores. Router to router communication was also analyzed. In addition to using Primetime, I used EDI to analyze timing paths visually. For many internal paths, EDI inverted the original starting point or endpoint, and subsequently changed the signal name to reflect the change, thereby making many start points and endpoints difficult to determine, as the original RTL start point and endpoint names no longer existed. Additionally, for many starting point and endpoints, there are many possible paths. For this reason, many paths needed to be drawn on paper for easier visualization to determine the correct starting points, endpoints, and block false paths.

Static timing analysis brought to attention a number of hold time violations on the communication paths between processors. Hold time violations occur when there is a large enough difference between clock path and datapath delay. Figure 5.3 illustrates this situation and shows a circuit with a large clock path delay and small datapath delay. The timing waveforms showing the effect of the hold time violation is shown in Figure 5.4, while the state change table is shown in Figure 5.5. As seen from these figures, the hold time violation causes uncertain output since it is possible for the new data to arrive before the old data has been registered. The new data arrives at the destination register before the rising clock edge corresponding to the old data.

The lead physical designer and I found a method to overcome this problem. If the valid bit met timing (both setup and hold), it was still possible to use communication links by transmitting data twice, with the receiver retaining only the second copy of the data. As a result, these timing violations didn't make the chip nonfunctional. The small number of timing paths where the valid bit had a timing violation were unusable. A mapping tool is able to route around these problem links. As a result of these timing violations, a method for correctly handling inter-processor communication was developed for KiloCore2 and is described in Section 5.2.1.8.

5.1.2 Chip Details

Figure 5.7 shows a block diagram of the KiloCore chip with the processors and memories highlighted. The chip die dimensions are 8 mm x 8 mm with a used die area of 7940 μm wide

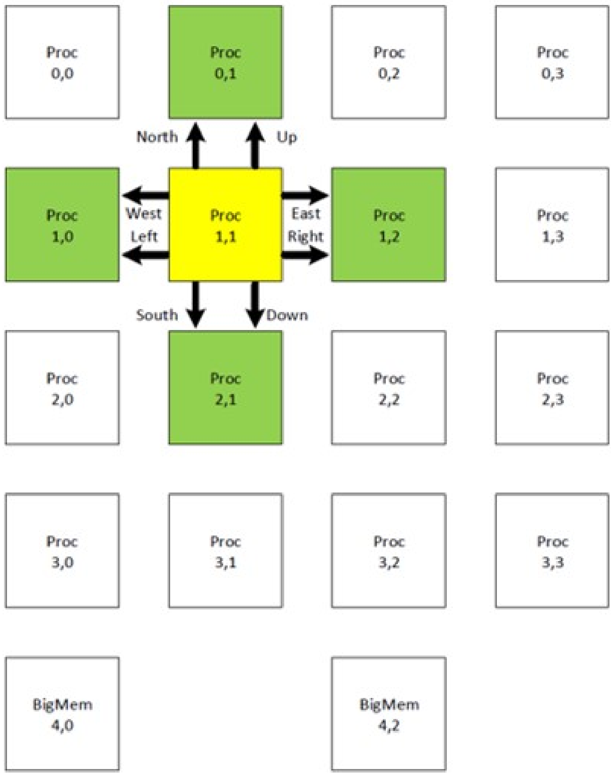


Figure 5.2: Design analyzed for examining inter-processor timing.

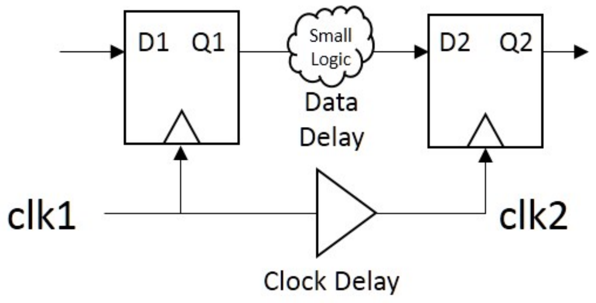


Figure 5.3: Circuit with large clock path delay and small datapath delay.

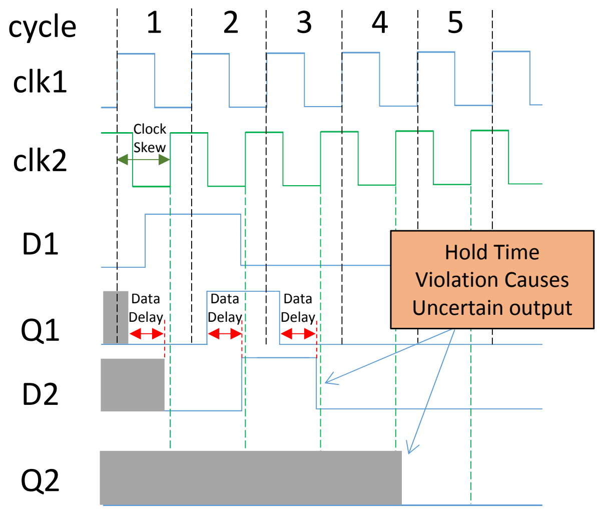
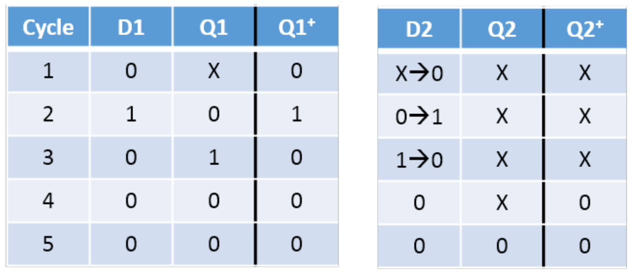


Figure 5.4: Timing waveforms showing hold time violation effects.



Cycle	D1	Q1	$Q1^+$
1	0	X	0
2	1	0	1
3	0	1	0
4	0	0	0
5	0	0	0

D2	Q2	$Q2^+$
X→0	X	X
0→1	X	X
1→0	X	X
0	X	0
0	0	0

Figure 5.5: State change table when hold time violations occur.

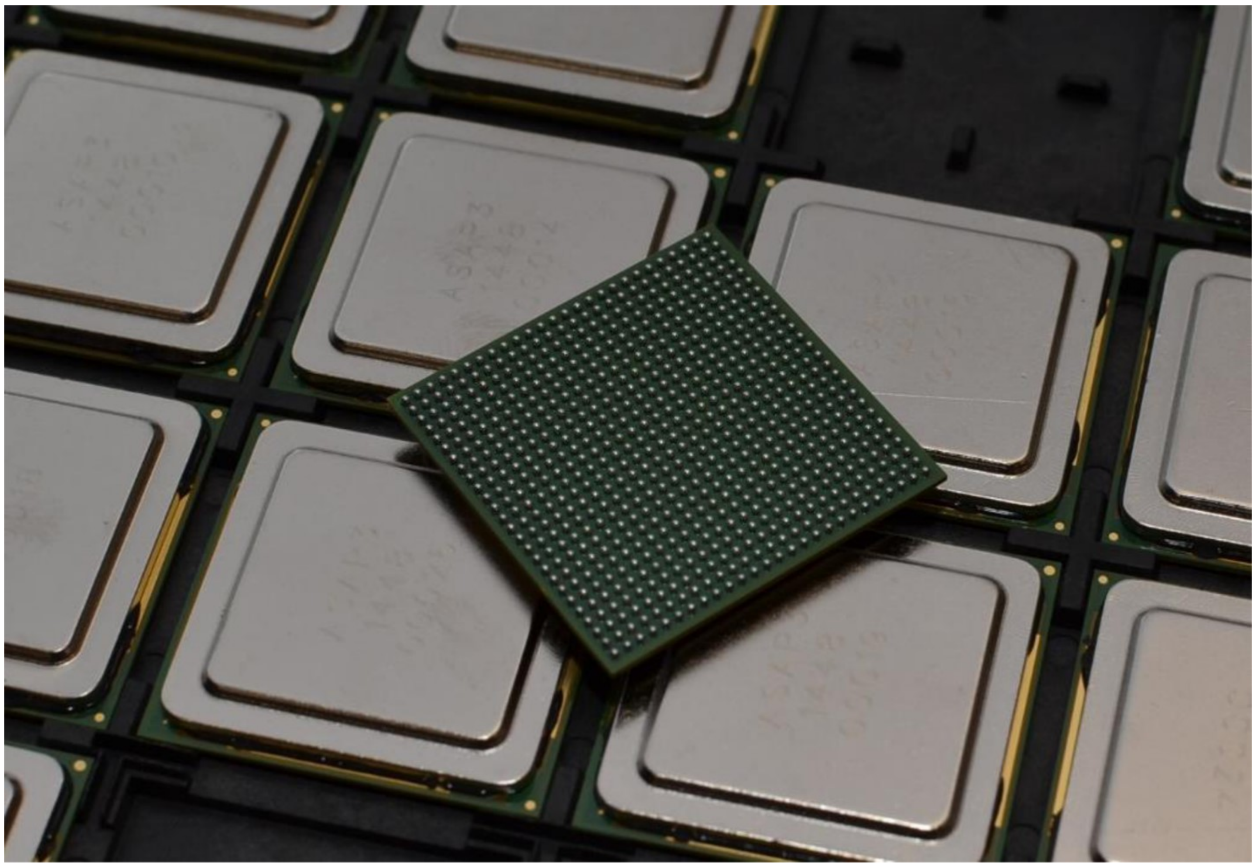


Figure 5.6: Underside of packaged KiloCore BGA package along with tray of packaged chips. The solder balls shown connect to the custom PCB daughtercard [8].

x 7940 μm tall. The chip package was a flip-chip ball grid array (BGA) with controlled collapse chip connected (C4) solder balls as shown in Figure 5.6.

The package features 564 pads to bond between the chip and package for I/O and supplying power. Of these, 64 are configured as pairs for low-voltage differential signaling (LVDS), 38 are for single-ended I/O, and 10 are analog voltage probe points for on-chip voltage measurements. Communication between chips and cores is possible via a direct circuit-switched network and packet router.

Figure 5.7 shows the layout of a single programmable processor tile in the upper right. Each of the 1000 programmable processor tiles is 232 μm x 239 μm and features 575,000 transistors. Each core includes its own local oscillator and is clocked independently of the other processors using a GALS method. Each core features a network-on-chip router. For each core, the instruction memory is 128x40-bits and the data memory is 256x16-bits. Each core has two first-in-first-out

(FIFO) communication buffers for inter-processor communication that are 32x16-bits each.

The core datapath is seven stages, and features a MAC with a 16x16bit multiplier with a 40-bit accumulator.

The SRAM module layout is shown in the bottom left of Figure 5.7. Each of the twelve SRAM modules provides 64 KB of storage and features two 32x18-bit input buffers, two 32x16-bit output buffers, and one 16x2-bit processor response buffer, and provides 28.4 Gb/s of I/O Bandwidth [109].

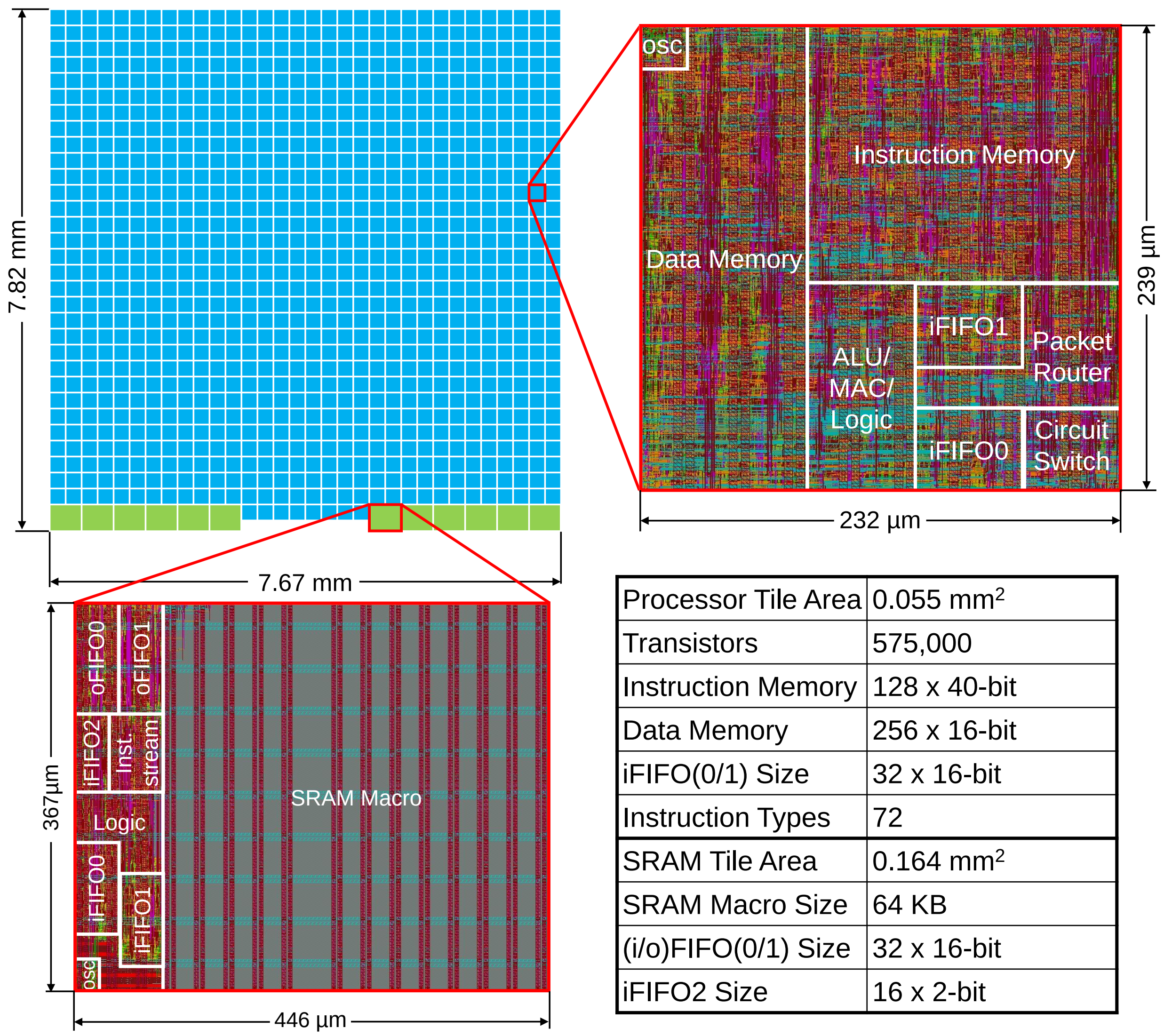
5.1.3 Measured Results

The maximum clock frequency for the processors ranges from 1.70 GHz to 1.87 GHz at 1.10 V [9]. Since the chip package available was a stock BGA, full power is provided only to the processors in the middle of the chip, which account for approximately 16% of the processor array. Using a custom designed package, a maximum of 1.78 trillion multiple instruction, multiple data (MIMD) operations per second per chip is achievable.

Table 5.1 compares KiloCore against other many-core and multi-core chips, as well as the low power Sleepwalker [110] for the metrics of processor count, processor area, clock frequency, energy per operation, energy times time, and bisection bandwidth (BW). The favorable value for each metric is indicated in bold red font. At a supply voltage of 0.56 V, processors require 5.8 pJ per operation at a clock frequency of 115 MHz. At this operating point, processors dissipate 1.3 mW each and execute 115 billion operations per second. The optimal operating point is at 0.9 V, where processors achieve their optimal energy times time metric of 11.1 (pj x ns)/op. The chip's independent memories operate at a clock frequency of 675 MHz at 665 mV, up to 1.49 GHz at 1.1 V. Each router can operate at a clock frequency of 262 MHz at 665 mV, up to 1.49 GHz at 1.1 V.

Table 5.1: Comparison of KiloCore and low power, multi-core and many-core chips [6, 9].

Chip	Processor Count	Technology (nm)	Processor Area (μm^2)	Clock Frequency (MHz)	Supply Voltage (V)	Energy/Op (pJ)	E x T (pJ x ns)	Bisection BW (Tb/s)
Sleepwalker [110]	1	65	0.42	25 23.6	0.4 0.375	2.6 **2.2**	104 93.2	-
IBM Cell [111]	9	90	14.5	**5000**	1.3	1100	220	2.46
Tilera/EZChip Gx72 [112]	72	40	-	1200	-	750	625	3.44
Intel TeraFlops [113]	80	65	3	4000 3130	1.2 1.0	70.6 49.1	17.7 15.7	2.65
Ambric Am2045 [24]	336	130	-	300	-	79.4	265	0.713
KiloCore [6]	**1000**	32	**0.055**	1782 1237 115	1.1 0.9 0.56	21.9 13.8 5.8	12.2 **11.1** 50.3	**4.24**



Processor Tile Area	0.055 mm^2
Transistors	575,000
Instruction Memory	128 x 40-bit
Data Memory	256 x 16-bit
iFIFO(0/1) Size	32 x 16-bit
Instruction Types	72
SRAM Tile Area	0.164 mm^2
SRAM Macro Size	64 KB
(i/o)FIFO(0/1) Size	32 x 16-bit
iFIFO2 Size	16 x 2-bit

Figure 5.7: Block diagram of KiloCore chip, single processor, and SRAM module. Processor and SRAM details are provided in the table [9].

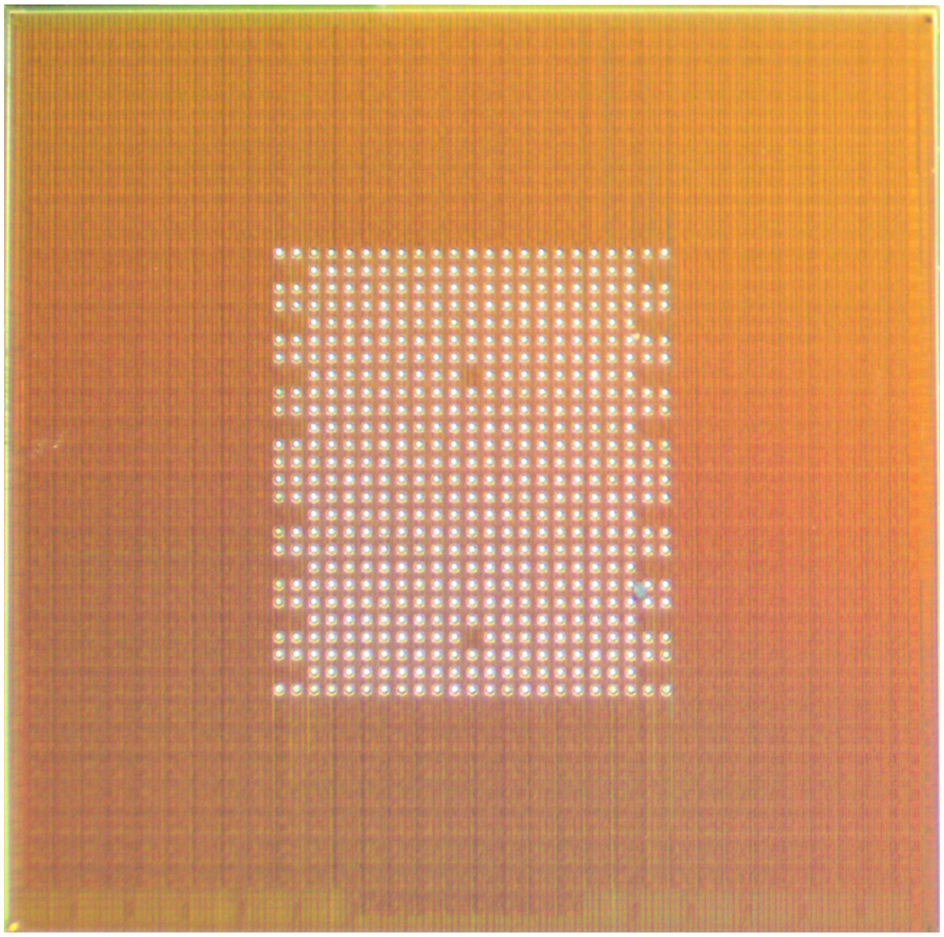


Figure 5.8: KiloCore chip micrograph [10].

5.2 KiloCore2

This section discusses the KiloCore2 chip, which was taped-out on March 1st, 2015 using 32 nm PD-SOI CMOS technology [8]. KiloCore2 was designed by six researchers, including myself. Just as with KiloCore, the architect was Brent Bohnenstiehl. For KiloCore2, I worked on physical design, but I also took on the responsibility of performing LVS, DRC, Ultrasim FastSpice simulations, powergate SPICE simulations, and RC extraction. The chip features 700 cores arranged in a 28 x 25 processor array configuration, with 697 programmable processor tiles and three hardware accelerators, one FFT, and two Viterbi decoder accelerators. There are fourteen 64 KB shared SRAM memories on the chip which are placed on the periphery. The nominal voltage is 0.9 V but the core voltage can range from 0.7 V–1.05 V. This chip has 580 million transistors.

5.2.1 Design Process

The design process for KiloCore2 is described next and was similar to that of KiloCore, with some minor changes due to design differences. Since more time was available for the KiloCore2 tapeout phase, the design and packaging of the chip was considered more carefully, and additional features were included. For instance, I made power planning more efficient by creating a spreadsheet for determining necessary wire widths for each metal layer.

KiloCore2 featured DVFS with three power rails and powergating circuitry. A temperature/voltage sensor was added to provide on-die temperature and voltage measurements. For each of the oscillators, the wireload was considered when determining the handpicked gates. The design of the oscillator was continually refined by manually adjusting the gate selection and performing the synthesis, place and route, RC extraction, and FastSPICE simulation process until an oscillator that met the clock frequency target was achieved. Additional macro blocks were added to the design flow, including a Viterbi decoder, FFT accelerator, and high clock frequency processor. Funding was available for a custom package for KiloCore2, and using data from Encounter, as well as my calculations for power delivery and I/O requirements, a custom package was designed for this chip.

5.2.1.1 Oscillators

For KiloCore2, four oscillators were designed by Timothy Andreas and Emmanuel Adeagbo, each with a major ring oscillator and microWatt oscillator. The major ring oscillator offers a wider range of clock frequency operation, whereas the microWatt oscillator is intended for low power operation. Each macro block (i.e., processor, fast processor, FFT accelerator, Viterbi decoder) contains both the major ring and microWatt oscillator. Additionally there is an oscillator for the router, which is similar to the microWatt oscillator but optimized for a different set of frequencies. The design of the major ring oscillator is adopted from KiloCore except that the gates chosen are for the maximum clock frequency achievable by a KiloCore2 processor. The fast processor's oscillator has gates selected for a higher maximum clock frequency.

As mentioned in Section 5.1.1.1, the RC components of the wireload affect the timing of the oscillator and must be considered. Therefore the design process of the oscillator involved first place and routing the handpicked gate netlists and the netlist from synthesis for the control logic. After performing clock tree synthesis, a gate level netlist and GDS were exported from Encounter by the lead physical designer. I then extracted the RC components of the oscillators using Calibre. Timothy, Emmanuel, and I ran Ultrasim FastSPICE simulations on the extracted RC model of the oscillators to determine the maximum achievable clock frequency. This process was repeated until the correct gates were determined for the major ring and microWatt oscillators which met the clock frequency targets.

5.2.1.2 Macro Blocks

Just as with KiloCore, KiloCore2 contains a set of macro blocks that were synthesized and placed and routed separately before being instantiated in the full chip layout. These macro blocks included a single processor, an independent 64 KB memory block, a fast version of the processor, the Viterbi decoder, and the FFT accelerator. The oscillators are also blackboxed when synthesizing the macro blocks. The hand selected gates for the oscillators, the synthesis produced netlists for the oscillator control logic, and the macro blocks are input to Encounter for placement and routing. Following place and route by the lead physical designer, a GDS and netlist were produced. I then performed RC extraction and verified the DRC and LVS using Calibre. I then ran FastSPICE

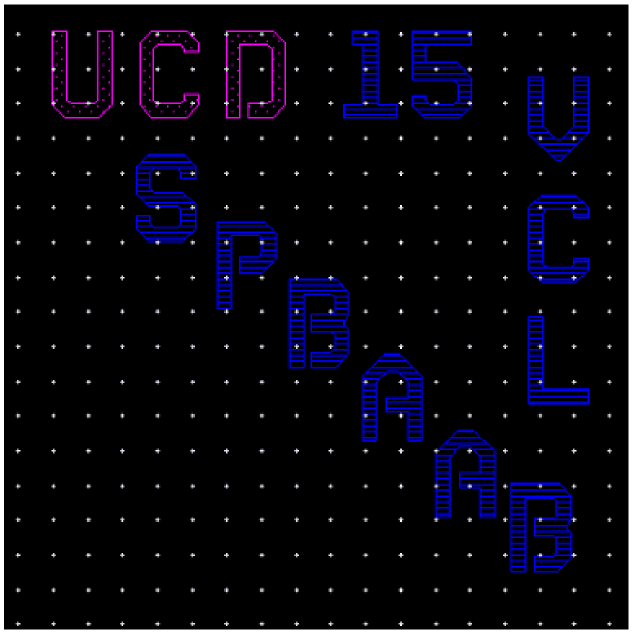


Figure 5.9: KiloCore 2 logo.

simulations on these blocks to verify their functionality after place and route. Once these blocks were instantiated in the full chip layout, wire routing was performed using Encounter.

In addition to the macro blocks, the MAC, router, and clock pulse probing circuitry were synthesized separately before being placed and routed as it was found that this method provided higher quality of results in terms of maximum clock frequency for these blocks.

5.2.1.3 Chip Level

The process for setting up the chip level design was identical to that carried out for KiloCore, except that the fast processors and hardware accelerators were also handled using the same procedure as described in Section 5.1.1.3.

5.2.1.4 Chip Finishing

The chip finishing process was identical to that used for KiloCore2. A picture of the logo I designed is shown in Figure 5.9. The logo features the research university acronym, the two digits of the tape-out year, the research group acronym, and the first initial of each contributing researcher's surname.

5.2.1.5 Power Planning

To assist with power planning, I created a spreadsheet to automatically calculate the copper wire resistivity (Ohms/nm) and resistance (Ohms) for each metal layer based upon the metal width and wire length. By entering the dimensions of the metal wires being used for the power grid into the spreadsheet, the total number of wires and width for each was determined for each metal layer.

5.2.1.6 Chip Package

Funding was provided for a custom KiloCore2 package. The specifications of this package were determined using a spreadsheet I created to automatically calculate the number of pins needed for core and I/O power delivery based on the bit-width and number of ports for I/O, configuration and programming, testing, clocks, sensors, and voltage rails. These I/O ports include both the single-ended and LVDS ports for the circuit-switched network and packet network, and connections for a dedicated optical network. The number of power and ground pins for each voltage rail was also automatically calculated. These calculations consider the number of cores, the active and leakage power on various power grids (using data from place and route), the number of on-chip memories and memory power, the supply voltage of each power grid, the percentage of cores on each grid, the number of idle cores, the pin current limit and derating factor, and I/O drive strength. The core clock frequency on different voltage rails and the cycles per word, router clock frequency and cycles per word, the optical clock frequency, and maximum frequency of the I/O drivers (both single-ended and LVDS) were considered for performing bandwidth calculations to determine which package option would provide sufficient input and output bandwidth.

5.2.1.7 Power Gates

For the DVFS controller, there are three power rails to choose from. To switch between these rails, powergating circuitry was added to the chip. The method used for determining the type and number of powergating cells is described next.

I performed SPICE simulations on the power gates available in the standard cell library to measure drain current (I_{DS}) as supply voltage (V_{DS}) varies by sweeping gate DC voltages from 0 V to V_{DDG}. The effective resistance and current supplied were calculated for 2% and 5% voltage drops across the power gate (i.e., V_{DS}). I created a spreadsheet to automatically calculate the number of power gates necessary depending on the peak processor current and desired V_{DDG} voltage drop, as well as the area overhead for the different power gate cells available in the library. This allowed the lead physical designer and I to determine which power gate type to use that would provide the desired voltage drop with minimal area overhead.

I found that other chips which used power gates targeted a 2–3% voltage drop across their grids [114, 115]. However, based on place and route data, a conservative estimate of 500 mA was estimated for the peak processor current, and since the current estimate was conservative, a 5% voltage drop was targeted. Based on this data, the power gates were dispersed throughout the processor due to the small area overhead and to enable more current sources through the power grid, rather than current being supplied from the periphery of a power domain. Following place and route, full chip simulations were performed in Ultrasim to verify powergate functionality.

5.2.1.8 Inter-processor Timing

As mentioned in Section 5.1.1.6, KiloCore suffered from a number of hold time violations from not properly considering inter-processor communication timing paths. Therefore, a method for KiloCore2 was developed to consider these timing paths. Given that EDI was unable to handle the place and routing for inter-processor timing for 1000 processors, the lead physical designer created a smaller test chip consisting of nine processors in a 3x3 grid as this was a more manageable problem for the tool. The results for nine cores extrapolates to 700 cores because it was possible to determine the timing constraints for a single core communicating to all of its neighbors. The lead physical designer and I altered the timing constraints of the individual processors until Encounter no longer required adding buffers to modify timing for inter-processor communication.

5.2.2 Chip Details

Figure 5.10 shows a block diagram of the KiloCore2 chip with the processors, memories, hardware accelerators, and temperature sensor highlighted. The chip die dimensions are 8 mm x 8 mm with a used die area of 7940 μm wide x 7940 μm tall. The chip package is a flip-chip BGA with C4 solder balls. This package features 2,499 pads to bond between the chip and package. Of these, 459 are for I/O, configured as 149 LVDS pairs and 161 single-ended connections. Twelve of the pins are used as analog probe points. These probe points connect to the local power grid of two processors chosen from different sides of the chip, the power and ground grids for each of the four power rails, and the power and ground rails for a temperature sensor. The remaining 2028 pins are used for power connections. The chip features three core power rails: 531 pins are used for $V_{DD\ HI}$, 200 pins for $V_{DD\ MED}$, and 196 pins for $V_{DD\ LOW}$. The I/O driver/receivers are attached to the $V_{DD\ I/O}$ grid which uses 82 pins. There are 1019 pins used for the ground rail, GND_{COM}. Communication between chips and cores is possible via a direct circuit-switched network and packet router. To enable a high speed link between chips, 70 pins were added to the periphery of the chip and package to connect to an optical interconnect.

Figure 5.11 shows the layout of a single programmable processor tile. Each of the 695 programmable processor tiles is 260.0 μm x 274.5 μm and features 750,601 transistors (364,065 n-channel MOSFETs, and 386,536 p-channel MOSFETs). Each core includes its own local oscillator and is clocked independently of the other processors using a GALS method. The maximum processor clock frequency is estimated at 2.0 GHz at 0.9 V. Each core features a redesigned network-on-chip router with a maximum clock frequency estimated at 2.0 GHz. For each core, the instruction memory is 128x40-bits and the data memory is 256x16-bits. Each core has two FIFO communication buffers for inter-processor communication that are 32x16-bits each. The core datapath is seven stages, and features a MAC with a 16x16bit multiplier with a 40-bit accumulator. The total power is estimated at 85 mW with 2.3 mW leakage. Additionally, there are three power rails that can be switched between; $V_{DD\ HI}$, $V_{DD\ MED}$, and $V_{DD\ LOW}$ by utilizing the powergate circuitry which is controlled by the DVFS controller.

Figure 5.12 shows the layout of a fast programmable processor tile. Two of these tiles were placed on chip and each tile is 274.5 μm x 250.0 μm. Each is a reduced version of a regular core in order to increase clock frequency. By reducing some critical path functionality and some critical path instructions, the maximum clock frequency is estimated to reach 3.85 GHz at 0.9 V. This tile uses 722,231 transistors. This processor only runs on the high voltage rail and uses all low threshold transistors to enable a faster clock frequency.

Figure 5.13 shows the layout of a 64 KB shared SRAM memory. There are fourteen of these memories included on the periphery of the chip for a total of 896 KB of total shared memory

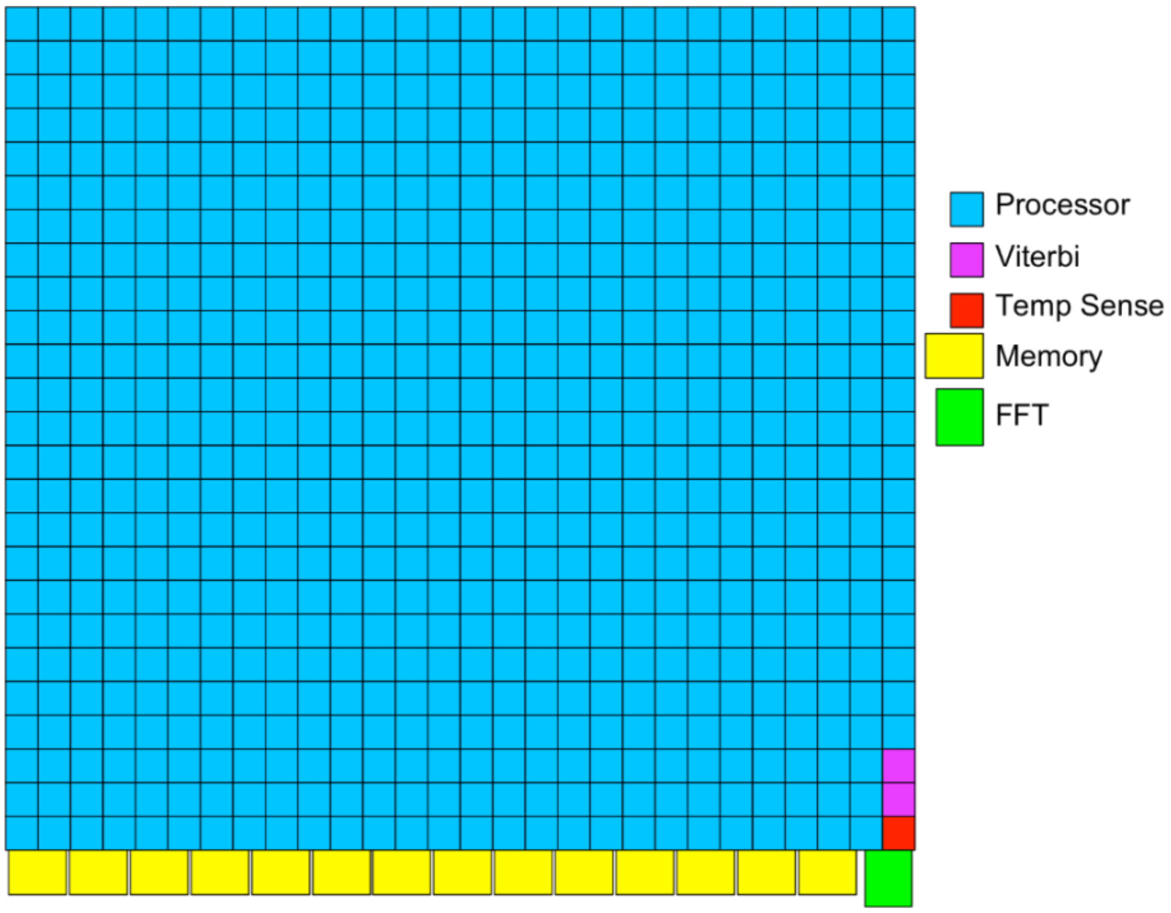


Figure 5.10: KiloCore2 block diagram showing processor, hardware accelerator, memory, and temperature sensor locations.

available on-die. These are useful for applications which require large datasets to be stored. This tile uses 3,835,867 transistors and measures 356.2 μm x 475.4 μm.

Figure 5.14 shows the layout of an FFT accelerator which was one of the hardware accelerators included on chip. This accelerator is useful for efficiently calculating the discrete Fourier transform, which is used in many scientific applications such as biomedical imaging [2] and radar [36]. This tile is 467.8 μm x 390.0 μm and features 2,563,558 transistors.

Figure 5.15 shows the layout of the Viterbi decoder, two of which were placed on the chip. This accelerator is useful for decoding a data sequence that has been encoded using a "finite-state" process. The Viterbi decoder algorithm is widely used in convolutional codes for radio communications, computer storage, and speech recognition. This tile is 274.5 μm x 250.0 μm and features 725,592 transistors.

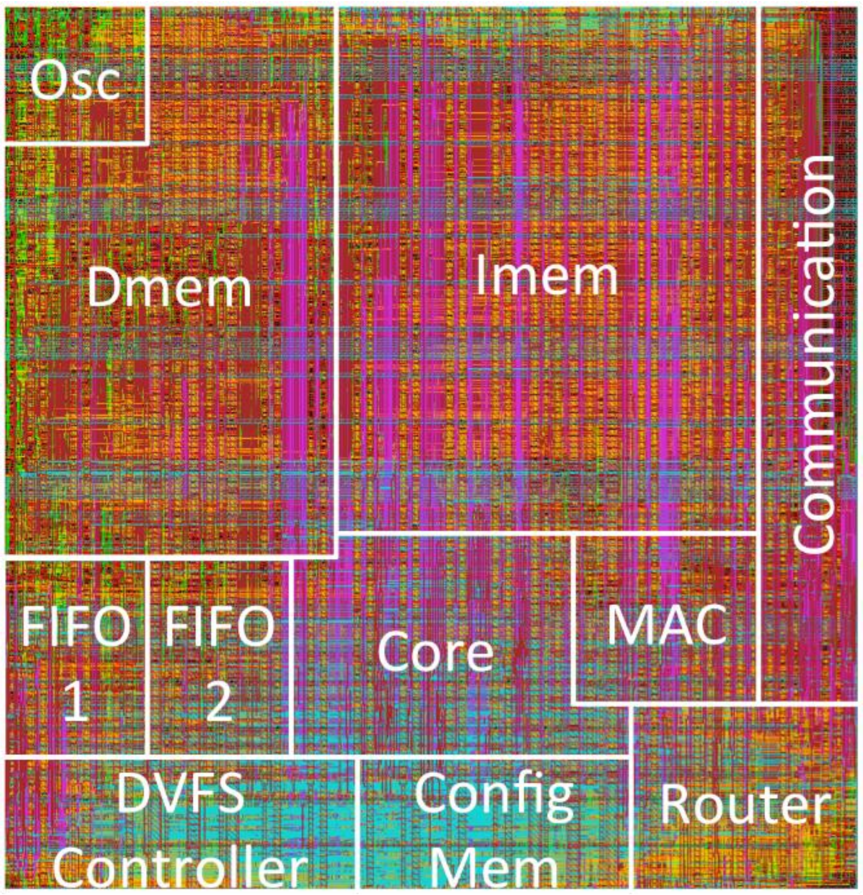


Figure 5.11: Layout of single programmable processor tile of KiloCore2 [8].

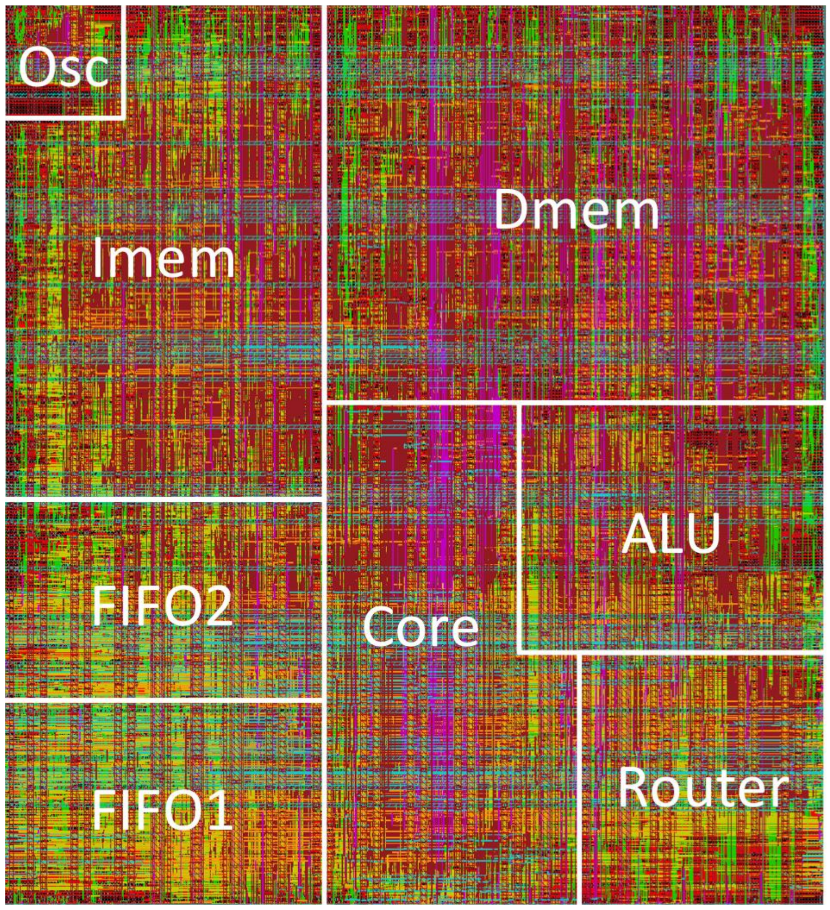


Figure 5.12: Layout of single high clock frequency programmable processor tile of KiloCore2 [8].

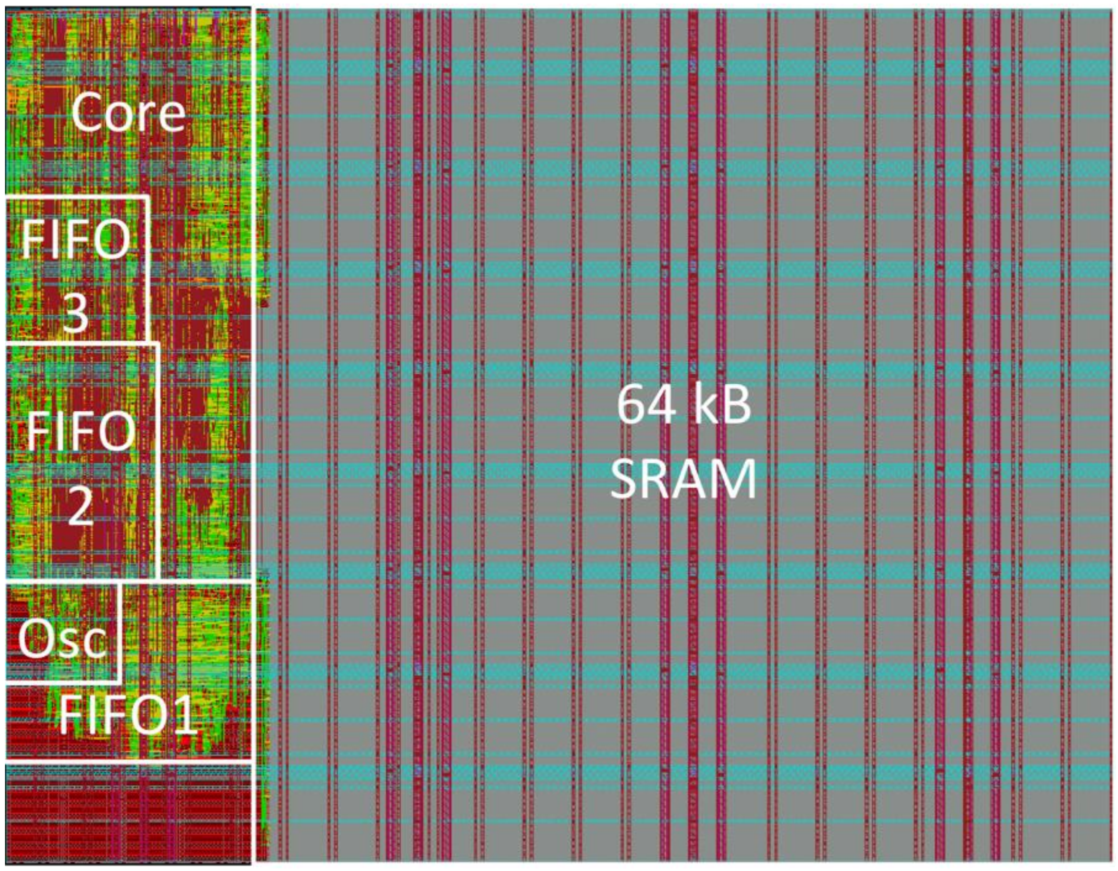


Figure 5.13: Layout of single 64 KB SRAM tile of KiloCore2 [8].

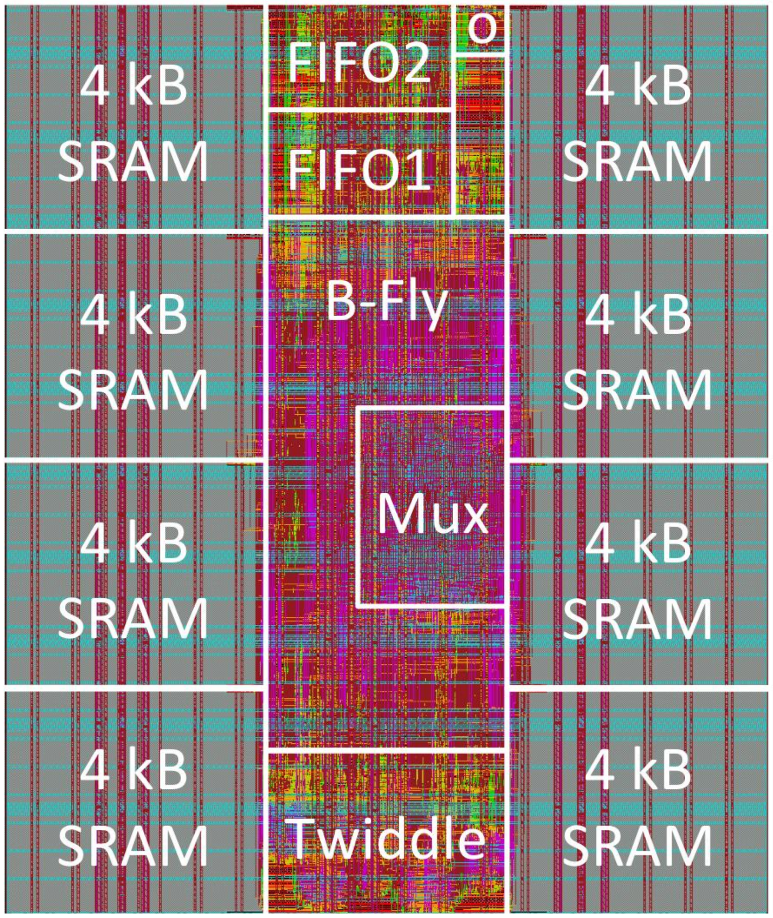


Figure 5.14: Layout of FFT tile of KiloCore2 [8].

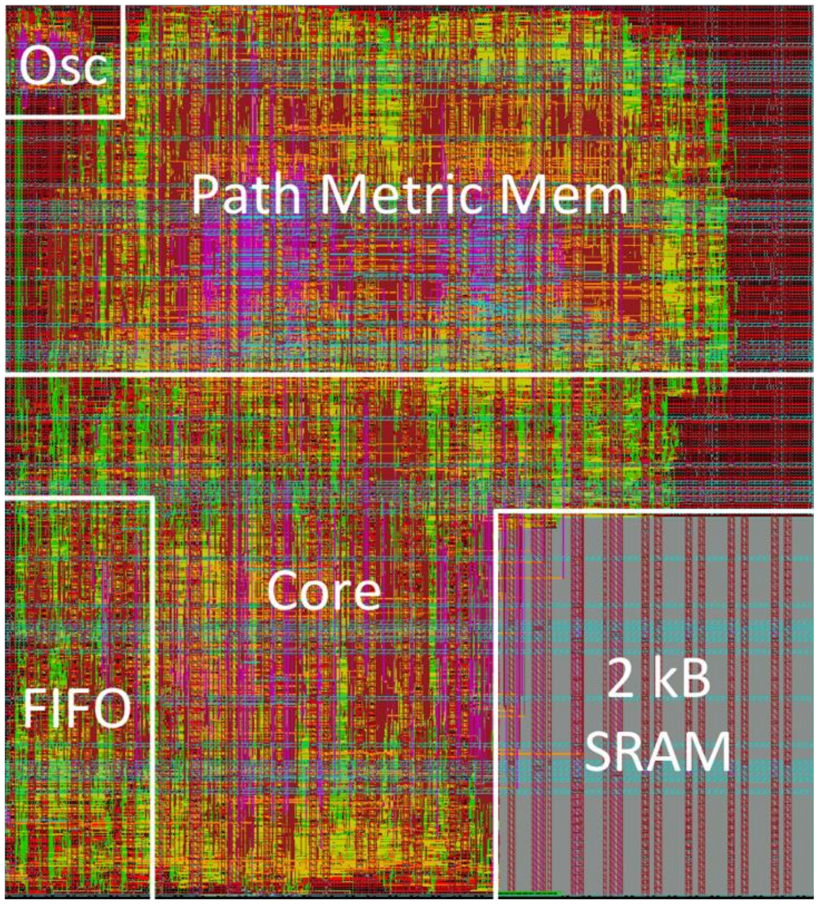


Figure 5.15: Layout of Viterbi decoder tile of KiloCore2 [8].

5.2.3 Measured Results

The assembled PCBs with packaged KiloCore2 chips are expected to be ready in July.

Chapter 6

Sparse Matrix-Vector Multiplication on a Many-Core Platform

The sparse matrix-vector multiplication (SpMV) operation involves calculating $Ax = b$, where A is an M×N sparse matrix, x is an N×1 dense vector, and b is an M×1 dense vector.

6.1 Sparse Matrix-Vector Multiplication Kernels

Several kernels are considered for performing SpMV, namely *SnakeSpMV*, *RowSpMV*, Parallel Subarrays, and Parallel Arrays. Various improvements for each kernel are considered for increasing efficiency, including parallel data distribution, sorting, processing, and accumulation. The platform on which these kernels are implemented is described in Section 5.1.

In the equations that follow, N is the number of rows in the vector *x,* or equivalently, the number of columns in matrix A. The number of elements of x is also equal to *N. M* is the number of rows in matrix *A*. *numLoads* is the number of times x values are loaded on the cores. *numCores* is the number of cores available for a kernel implementation. *memPerCore* is the memory available per core for storing *x. memShared* is the size of on-chip memory excluding each core's local memory. *memPerVal* is the amount of memory required to store each x value. $numStorableVals$ is the number of x values that can be stored on-chip and is shown in Equation 6.1

$$numStorableVals = \frac{memPerCore * numCores + memShared}{memPerVal} \tag{6.1}$$

The rows of A are streamed in using the format expected by the specific SpMV kernel, and the format expectations for each kernel are discussed in the following sections. In general, data from matrix A are stored in a format similar to compressed sparse row (CSR), in that only the nonzero values of A are stored, as well as the column indexes; however, the row pointer is not stored. Column indexes start at 0 and continue to $(N{-}1)$. Column indexes are received, followed by the corresponding nonzero matrix value (e.g., $colInd[0]$, $matVal[0]$, $colInd[1]$, $matVal[1]$, etc.).

While some equation variable names are reused between different kernels (e.g., *numLoads*), the variables are specific to the section in which they are discussed unless stated otherwise. A summary of the variables used in Chapter 6 is provided in Tables 6.3 and 6.4.

For SpMV, two cases are identified; case 1, where the x vector fits entirely on the chip, and case 2, where x doesn't fit on chip and must be loaded in multiple times.

For case 1, x is divided evenly among the cores and stored on the chip. The values of x can be stored in on-chip memory during the programming and configuration stage, or streamed in and stored after the array is programmed. The entries of A are then streamed in.

For case 2, there is not sufficient memory on chip to store x, so the values are streamed in using multiple loads. The values of x are evenly distributed for each load. The sparse data for each row are received in sections according to their column number. Once all rows for a column section have finished being processed, the next set of x values are loaded and data from the next range of columns from each row of A are received.

While this work focuses primarily on case 1, the equations and considerations for implementing case 2 for *SnakeSpMV* and *RowSpMV* are provided. Exploration of case 2 for other SpMV kernels is left as a future research endeavor.

6.2 SnakeSpMV

The *SnakeSpMV* kernel involves streaming data through a set of cores that are connected in a chain. First, x is stored in the array of cores (either during programming or streamed in), and then sparse matrix data from A is streamed in. If the column index of a sparse data point matches the row index of an x value stored in a core, then the data value from A is multiplied by the x value; otherwise, the sparse data point is forwarded to downstream cores. After the last value of a row has been sent, a token is sent to indicate the end of a row. Results are accumulated in each core and flushed when an end of row token (equal to N) has been received instead of a valid column. The largest value of $colHigh$ for any core is (*N–1*), therefore each core will know when it has received a token because input column indices are compared to make sure they are between a core's *colLow* and $colHigh$. If the input value is larger than $colHigh$, then a token has been received.

The following sections include equations used for implementing the *SnakeSpMV* kernel. Variables are indexed by core number, where core (0) is the first core (where data is received), and core (*numCores–1*) is the last core (where data is sent out). The variable i represents the logical core id and the cores are chained together, although any mapping topology is possible as long as data is routable.

6.2.1 Case 1

Case 1 is when the x vector fits on chip, and Equation 6.2 holds true where

$$numLoads = \left\lceil \frac{N}{numStorableVals} \right\rceil == 1. \tag{6.2}$$

The elements of x are evenly distributed among the cores according to

$$numCols[i] = \left\lfloor \frac{N}{numCores} \right\rfloor + (i < k);\ i = 0, ..., (numCores - 1);\ k = (N \mod numCores), \tag{6.3}$$

where $numCols[i]$ is the number of column indexes of matrix A for which core i handles multiplications. The last part of the equation, $(i < k)$, evaluates to 1 when the value of i is less than k. The purpose of k is to account for the additional elements that remain when N is not evenly divisible by $numCores$. Each core keeps a record of the lowest column index of matrix A (i.e., the lowest row index from x) for which values from vector x are stored for multiplying, given by

$$colLow[i] = N - \sum_{m=0}^{i} numCols[m];\ i = 0, ..., (numCores - 1), \tag{6.4}$$

whereas the highest column index of matrix A (i.e., the highest row index from x) for which values

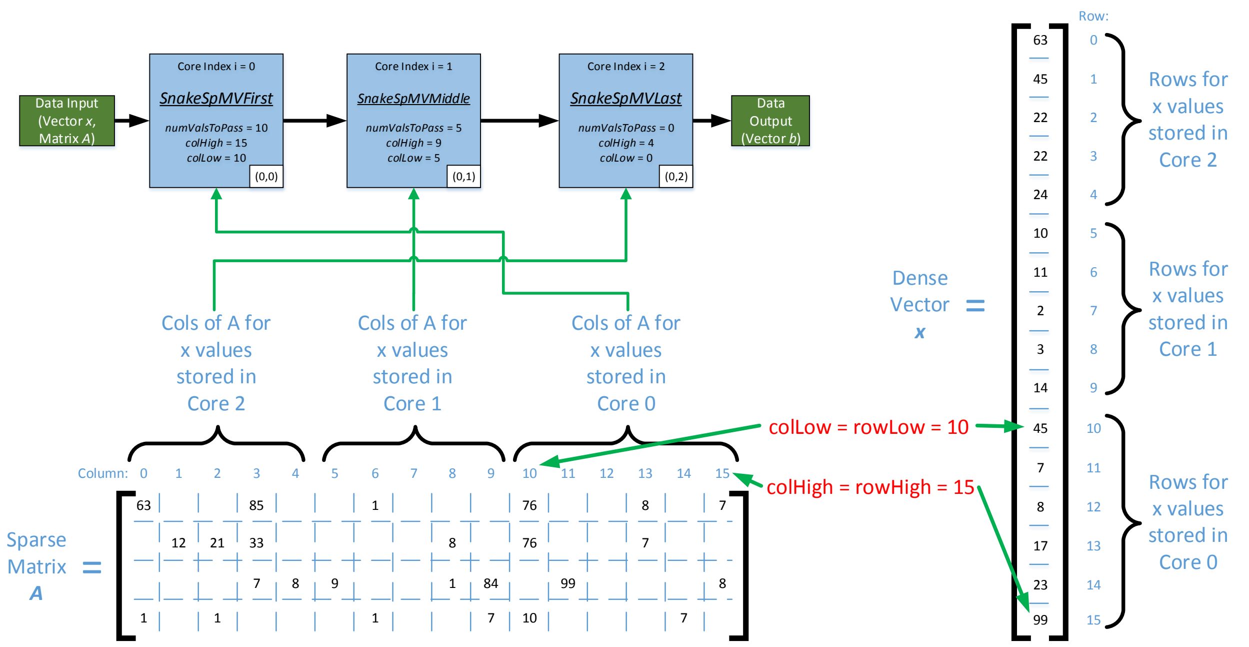


Figure 6.1: Example of how *colLow* and *colHigh* are determined. *colLow* corresponds to the lowest column of A for which rows of x are stored in a specific core. Similarly, *colHigh* is the highest column of A for which rows of x are stored. The top left of the figure shows a three core mapping of the *SnakeSpMV* case 1 program for when N=16, as well as the *colLow* and *colHigh* for each core.

from vector x are stored for multiplication is given by

$$colHigh[i] = N - 1 - \sum_{m=1}^{i} numCols[m]; \; i = 0, ..., (numCores - 1). \tag{6.5}$$

Note that the second portion of Equation 6.5, $\sum_{m=1}^{i} numCols[m]$, is 0 when $i = 0$. Given that referring to column indexes of A is equivalent to referring to row indexes of x, $colLow$ could have been named $rowLow$, and $colHigh$ renamed $rowHigh$ as shown in Figure 6.1.

All cores, except the last, pass incoming x values downstream before storing data in their memory, with the number of values to pass equal to $colLow[i]$,

$$numValsToPass[i] = colLow[i]; \; i = 0, ..., (numCores - 1). \tag{6.6}$$

There are three programs used for the case 1 SnakeSpMV kernel; *SnakeSpMVFirst*, *SnakeSpMVMiddle*, and *SnakeSpMVLast*. *SnakeSpMVFirst* corresponds to the first core in the array with index 0, *SnakeSpMVMiddle* corresponds to cores 1 to ($numCores$–2), and *SnakeSpMVLast*

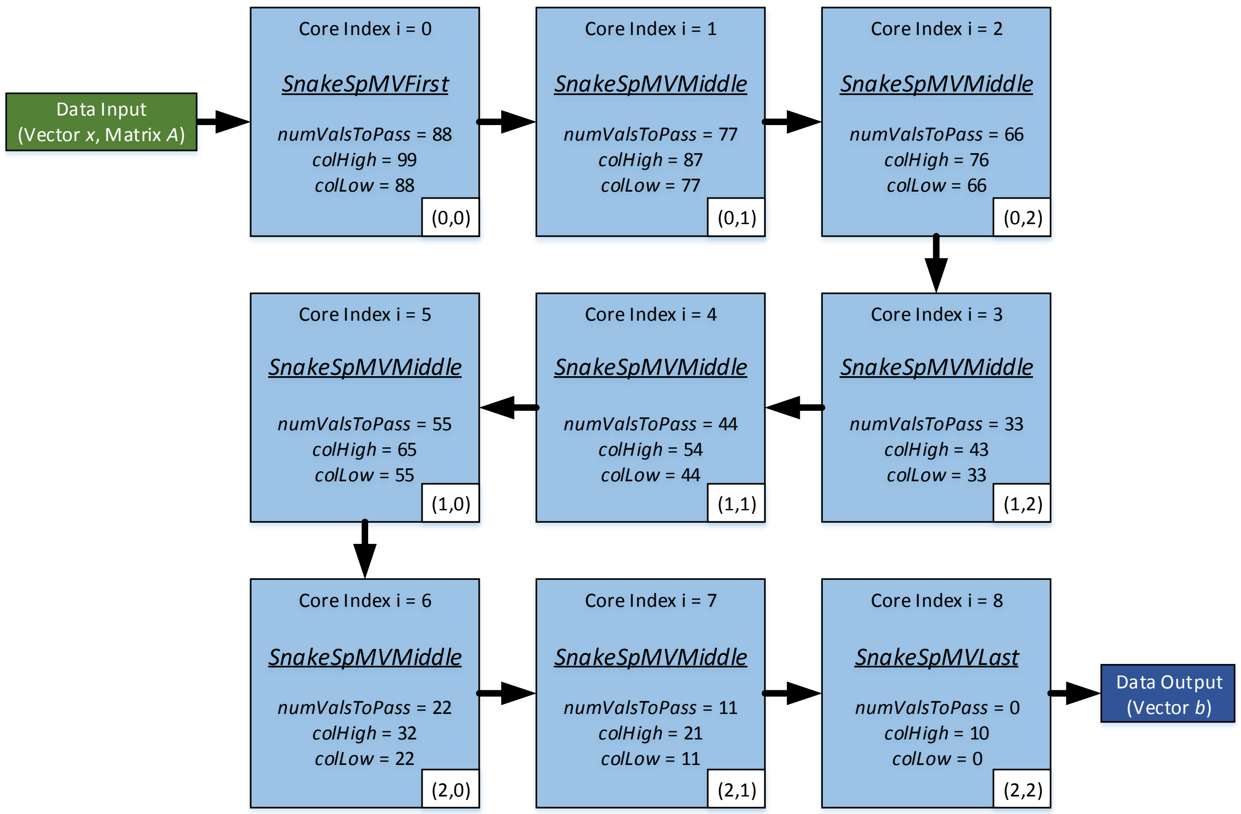


Figure 6.2: Nine core mapping of the *SnakeSpMV* case 1 kernel and $N = 100$. The x vector values are evenly divided among the cores. The physical id (row,column) is shown in the bottom right hand corner of each core. The logical id is listed at the top of each core, and represents the variable i in Equations 6.3–6.6. Dense column vector (i.e., x) and sparse matrix (i.e., A) data is passed into the array through core (0,0) and the resultant dense column vector (i.e., b) is sent out from core (2,2). Each core, except for the last, passes x values downstream before storing x values in memory. The cores are labeled with the number of x values to pass downstream, as well as the low and high column indexes.

corresponds to the last core with index (*numCores–1*). A nine core mapping of this kernel is shown in Figure 6.2 for the case when $N = 100$.

Algorithm 2 shows the pseudocode for *SnakeSpMVFirst*. Downstream cores receive control keys in order to know what kind of data to expect. For instance, if a control key of 0 is received, a core will receive a column index next; however, if a control key of 1 is received, then that core will receive accumulated data. After *SnakeSpMVFirst* has finished passing on values according to Equation 6.6, x vector data is stored in this core's memory. Sparse matrix data is received next. *SnakeSpMVFirst* reads the input column, $colInd$, and verifies that it lies between $colLow[i]$ and $colHigh[i]$. During

Algorithm 2 Pseudocode of *SnakeSpMVFirst* (continued on next page)

```
for ($i = 0$; $i < numValsToPass$; $i++$) do                        \\ Pass x vector values downstream
    $Output \leftarrow Input$
end for
for ($i = 0$; $i < numCols$; $i++$) do                              \\ Store x vector values
    $Mem[i] \leftarrow Input$
end for
checkColumn:                                                        \\ Check if $colHigh \leq colInd \geq colLow$
$column \leftarrow Input$                                           \\ Read column
$vecAddr \leftarrow column - colLow$                                \\ Calculate address of $vecVal$
if $vecAddr \geq 0$ then                                            \\ Check if $column \geq colLow$
    if $column > colHigh$ then
        $Output \leftarrow 0$                    \\ $Input == token$, so end of a row, pass token downstream
        $Output \leftarrow column$
    else
        go to multiply                     \\ $colHigh \leq colInd \geq colLow$ so go multiply $vecVal * matVal$
    end if
else                                                                \\ $colInd < colLow$, so pass input
    $Output \leftarrow 0$                                           \\ Send key of 0, $column$, and $matVal$
    $Output \leftarrow column$
    $Output \leftarrow Input$
end if
go to checkColumn                                                   \\ Check next input
```

this process, *vecAddr* is calculated and is also the memory address for accessing the x value (i.e., $vecVal = x[vecAddr]$) by first subtracting $colLow[i]$. If *colInd* lies between $colLow[i]$ and $colHigh[i]$, then the branch to `multiply` is taken and the matrix value, *matVal*, and the vector value, *vecVal*, are multiplied and stored as the accumulated sum (i.e., *accumSum*), and the next column index is read. If instead *colInd* is less than $colLow[i]$, then a control key value of 0 is sent downstream along with *colInd* and *matVal*. If a token is received (recall that token = *N*, therefore *colInd* $>colHigh[i]$), then a control key value of 0 is sent downstream along with the token. A control key of 0 indicates that a *colInd* is being sent, and possibly a *matVal*. Assuming the first *colInd* was within $colLow[i]$ and $colHigh[i]$ and the two were multiplied, the branch to `checkColumnDataPresent` is taken and the next *colInd* is checked. If it is within $colLow[i]$ and $colHigh[i]$ again, the branch to `multiplyAndAccumulate` is taken, *matVal* and *vecVal* are multiplied, and the product is added to the accumulating sum. If instead of *colInd,* a token is received, then a control key of 1 is passed downstream, along with *accumSum*, and the branch to `checkColumn` is taken. A control key of 1 indicates that the end of row token was received and *accumSum* is being sent. Once a core has sent a control key of 0 or 1, the *accumSum* is available to be overwritten on the next multiplication.

Algorithm 2 Pseudocode of *SnakeSpMVFirst* (continued)

```
multiply:
$vecVal \leftarrow Mem[vecAddr]$                                  \\ Load $vecVal$ from memory
$accumSum \leftarrow Input * vecVal$      \\ Multiply $matVal$ (i.e., input) and $x$ value, overwrite $accumSum$
go to checkColumnDataPresent
multiplyAndAccumulate:
$vecVal \leftarrow Mem[vecAddr]$                                  \\ Load $vecVal$ from memory
$product \leftarrow Input * vecVal$                   \\ Multiply $matVal$ (i.e., input) and $x$ value
$accumSum \leftarrow accumSum + product$                                      \\ Accumulate
checkColumnDataPresent:
$column \leftarrow Input$                                                 \\ Read next input
$vecAddr \leftarrow column - colLow$
    \\ Data are sent in order, so no need to check if $colInd < colLow$
if $column > colHigh$ then                \\ Check if $colInd > colHigh$, if so then input is a token
    $Output \leftarrow 1$               \\ $Input == token$, so end of a row, pass $accumSum$ downstream
    $Output \leftarrow accumSum$
else
    go to multiplyAndAccumulate          \\ $colHigh \leq colInd \geq colLow$ so multiply $vecVal * matVal$
end if
go to checkColumn                                                  \\ Check next input
```

SnakeSpMVMiddle handles inputs similar to *SnakeSpMVFirst.* After the appropriate x vector data is stored into a core's memory, the next expected input is a control key. If a 1 is received, and no accumulated sum has been calculated within this core, then another control key of 1 is sent downstream, followed by the input accumulated sum from either *SnakeSpMVFirst* or *SnakeSpMVMiddle* (depending on which precedes this core). If a control key of 0 is received, then the input $colInd$ is compared against $colLow[i]$ and $colHigh[i]$. If $colInd$ is less than $colLow[i]$, then a control key value of 0 is sent downstream along with the $colInd$ and $matVal;$ if $colInd$ is equal to the token value, then a control key value of 0 is sent downstream along with the token. If $colInd$ is within the bounds of the high and low column index, the input $matVal$ is multiplied by the corresponding $vecVal.$ The next control key is then checked to determine if a) data should be accumulated with the input $accumSum$, sent downstream, and the accumulated sum overwritten on the next multiplication (i.e., key == 1), or b) if the $colInd$ should be compared against $colLow[i]$ and $colHigh[i]$ (i.e., key == 0). If $colInd$ is within the column range, then $matVal$ is multiplied by $vecVal$ and added to $accumSum$, otherwise a key of 1 followed by $accumSum$ is sent downstream. Table 6.1 shows the actions taken depending on the control key the core receives.

Since the core with *SnakeSpMVLast* is the last core in the chain, it does not pass x values downstream. Table 6.1 shows the actions taken depending on the control key the core receives. If a key of 0 is received, followed by a token and an $accumSum$ hasn't been formed, then a zero result is output because the token indicates the end of the row. If, on the other hand, an $accumSum$

Table 6.1: Actions Taken Depending on Control Key Received

Program	Western Input Key is 0	Western Input Key is 1	Northern Input Key is 0	Northern Input Key is 1
SnakeSpMVFirst	N/A*	N/A*	N/A*	N/A*
SnakeSpMVMiddle/ RowSpMVMiddle	Compare incoming column to *colLow* and *colHigh*. Perform multiplication if column is not a token, otherwise a control key of 1 followed by any accumulated data this core has is sent west. If instead no accumulated data has been formed in this core when a token is received, then only a control key of 0 is sent west.	Any accumulated data in this core is added to the incoming data from western direction. A control key of 1 is then sent west followed by the accumulated data.	N/A*	N/A*
SnakeSpMVLast	Compare incoming column to *colLow* and *colHigh* to decide what to do (either multiply or output accumulated sum).	Any accumulated data in this core is added to the incoming data from western direction and sent out.	N/A*	N/A*
RowSpMVFirstCol	N/A*	N/A*	N/A*	N/A*
RowSpMVLastColFirstRow	Compare incoming column to *colLow* and *colHigh*. Perform multiplication if column is not a token, otherwise a control key of 1 followed by any accumulated data this core has is sent south. If instead no accumulated data has been formed in this core when a token is received, then only a control key of 0 is sent south.	Any accumulated data in this core is added to the incoming data from western direction. Another control key of 1 is sent south, followed by the accumulated sum.	N/A*	N/A*
RowSpMVFirstColLastRow	N/A*	N/A*	N/A*	N/A*
RowSpMVLastCol	Compare incoming column to *colLow* and *colHigh* to decide what to do (either multiply or move onto check northern input). If a token a received, the northern input is checked.	Any accumulated data in this core is added to the incoming data from western direction. The northern input is then checked.	Accumulated sum (if one exists) is sent out, otherwise a 0 is sent south.	Any accumulated data in this core is added to the incoming data from northern direction and sent south.
RowSpMVLast	Compare incoming column to *colLow* and *colHigh* to decide what to do (either multiply or move onto check northern input).	Any accumulated data in this core is added to the incoming data from western direction.	Accumulated sum (if one exists) is sent out, otherwize a 0 is sent.	Any accumulated data in this core is added to the incoming data from northern direction and sent out.

* Core with this program doesn't receive control keys or input from this direction

has been formed then it is sent out as the value of b for the current row. If a key of 0 is received followed by a $colInd$ that lies between $colLow[i]$ and $colHigh[i]$, then the matrix data and vector value are multiplied and added to the $accumSum$ if one has already been formed; otherwise, the product will become the next $accumSum$.

6.2.2 Case 2

Case 2 refers to when the vector x doesn't fit on chip and Equation 6.7 holds true where

$$numLoads = \left\lceil \frac{N}{numStorableVals} \right\rceil > 1. \tag{6.7}$$

The equations for case 2 are similar to those for case 1; however, the x values are received using multiple regular loads and one last load. The number of x values for the last load is calculated differently than a regular load to account for when x cannot be evenly divided among the loads (when $(N \mod numLoads)\; != 0$). A regular load is the case where data can be evenly partitioned and the equations are reused each time new data is loaded. The last load represents when a dedicated equation is needed to account for the remaining values to be loaded.

The number of x values stored in a specific core on a regular load is determined by the equation

$$\begin{gathered} numColsInit[i] = numValsPerCoreRegLoad + (i < k), \\ i = 0, ..., (numCores - 1);\; k = numValsAddRegLoad \end{gathered} \tag{6.8}$$

where the lowest number of x values stored per core on a regular load is calculated by

$$numValsPerCoreRegLoad = \left\lfloor \frac{numValsRegLoad}{numCores} \right\rfloor, \tag{6.9}$$

and the additional number of x values that need to be stored that didn't evenly divide among the cores on a regular load is given by

$$numValsAddRegLoad = numValsRegLoad \mod numCores. \tag{6.10}$$

The total number of x values stored across all cores on a regular load is calculated from

$$numValsRegLoad = \left\lceil \frac{N}{numLoads} \right\rceil . \tag{6.11}$$

For a regular load, the lower index of matrix A columns that will be multiplied by values of x in a core initially is given by

$$colLowInit[i] = numValsRegLoad - \sum_{m=0}^{i} numColsInit[m], \tag{6.12}$$
$$i = 0, ...(numCores - 1).$$

The upper index of matrix A columns that will be multiplied by values of x in a core initially is calculated with

$$colHighInit[i] = numValsRegLoad - 1 - \sum_{m=1}^{i} numColsInit[m], \tag{6.13}$$
$$i = 0, ..., (numCores - 1)$$

and each of these values are incremented by *numValsRegLoad* after each load.

For a regular load, the number of x values to pass downstream before storing values from the x vector in a core initially is given by

$$numValsToPassInit[i] = colLowInit[i]. \tag{6.14}$$

The calculations for the final load handles cases where the number of loads doesn't evenly divide into the number of rows of the x vector. Therefore, the number of x values stored in a specific core on the last load is determined using

$$numColsLast[i] = numValsPerCoreLastLoad + (i < k), \tag{6.15}$$
$$i = 0, ..., (numCores - 1);\ k = numValsAddLastLoad$$

where the lowest number of x values stored per core on the last load is given by

$$numValsPerCoreLastLoad = \left\lfloor \frac{numValsLastLoad}{numCores} \right\rfloor. \tag{6.16}$$

The additional number of x values that need to be stored that didn't evenly divide is determined from

$$numValsAddLastLoad = numValsLastLoad \mod numCores, \tag{6.17}$$

and the total number of x values stored across all cores is given by

$$numValsLastLoad = N - \left\lfloor \frac{N}{numStorableVals} \right\rfloor * numValsRegLoad. \tag{6.18}$$

For the last load, the lower index of matrix A columns that will be multiplied by values of x in a core comes from

$$colLowLast[i] = N - \sum_{m=0}^{i} numColsLast[m];\ i = 0, ..., (numCores - 1), \tag{6.19}$$

and the upper index of matrix A columns that will be multiplied by values of x in a core is

$$colHighLast[i] = N - 1 - \sum_{m=1}^{i} numColsLast[m];\ i = 0, ..., (numCores - 1). \tag{6.20}$$

Finally, the number of x values to pass downstream before storing values from the x vector in a core for the last load is given by

$$numValsToPassLast[i] = numValsLastLoad - \sum_{m=0}^{i} numColsLast[m], \tag{6.21}$$
$$i = 0, ..., (numCores - 1).$$

At the end of each row, a token equal to N is received. Once the number of tokens received is equal to the number of rows that matrix A has (i.e., M), the next set of x values are received, followed by the partially computed b vector data from the previous regular load. The partially computed row values are accumulated with the data from the remaining sets of columns of A. After the final load, the final values of the b vector are output.

6.3 RowSpMV

The *RowSpMV* kernel involves streaming data through a set of cores that are connected in a rectangular configuration with a set of processing rows and columns. The advantage of this kernel versus the *SnakeSpMV* method is that data doesn't have to traverse as many hops to reach the output. For instance, with a *SnakeSpMV* implementation with 100 cores, each data item must pass through 100 cores; however, with a *RowSpMV* implementation, the number of cores to travel from input to output is 19. Each downstream core also has fewer upstream cores it must wait on before processing data.

Similar to *SnakeSpMV*, x vector data is first stored in the array of cores, and then sparse matrix data is streamed in. If the column of a sparse data point matches the row of an x value stored in a core, then the data value from A is multiplied by the value from x. Results are accumulated in each core and flushed when an end of row token (equal to N) has been received. Data from the sparse matrix and x vector are distributed through the first column of processing cores and passed east to the appropriate core. Flushing involves passing accumulated results east to the last processing column. Each core of the last processing column accumulates data and passes results south.

In the equations below, *numCoreRows* is the size of the core array in the y dimension, and *numCoreCols* is the size of the core array in the x dimension. Variables are indexed by core number (row, column), where core (0,0) is the first core (where data is received), and core (*numCoreRows*–1, *numCoreCols*–1) is the last core (where data is sent out). The variables i and j are indexes for representing the logical core id; the cores are assumed to be mapped in a 2D grid, although any mapping topology is possible as long as data is routable. Data is stored in the same format described in Section 6.2 for *SnakeSpMV*.

6.3.1 Case 1

Case 1 refers to when the vector x fits on chip and Equation 6.2 holds true. A sixteen core mapping of this kernel is shown in Figure 6.3 for $N = 100$.

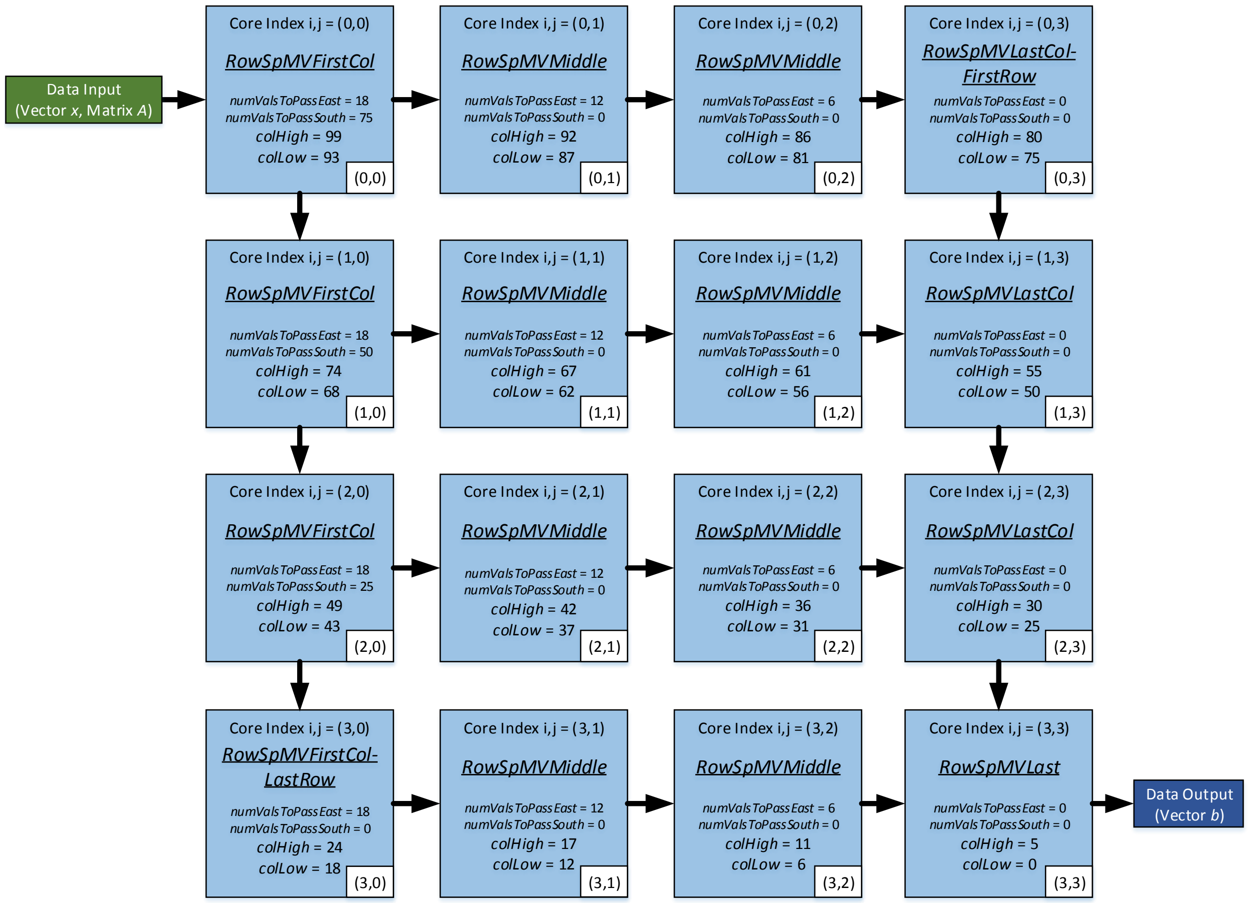


Figure 6.3: Sixteen core mapping of the *RowSpMV* case 1 kernel consisting of four rows and four columns and $N = 100$. The x vector values are evenly divided among the cores. Each core is labeled with its program name. The physical id (row,column) is shown in the bottom right hand corner of each core. The logical id is listed at the top of each core, and represents the variables i and j in Equations 6.22–6.28. Dense column vector (i.e., x) and sparse matrix (i.e., A) data is passed into the array through core (0,0) and the resultant dense column vector (i.e., b) is sent out from core (3,3). Each core, except for the last, passes x values downstream before storing x values in memory. The cores are labeled with the number of x values to pass east and south, as well as the low and high column indexes.

The elements of x are evenly distributed among the cores according to the following equation, where i represents the core's row and j represents the core's column:

$$numCols[i][j] = \left\lfloor \frac{numColsPerRow[i]}{numCoreCols} \right\rfloor + (j < k), \tag{6.22}$$

$$i = 0, \ldots, (numCoreRows - 1);\ j = 0, \ldots, (numCoreCols - 1);$$

$$k = (numColsPerRow[i] \mod numCoreCols)$$

and the number of elements that each row of cores receives is given by

$$numColsPerRow[i] = \left\lfloor \frac{N}{numCoreRows} \right\rfloor + (i < k). \tag{6.23}$$

$$i = 0, \ldots, (numCoreRows - 1);\ k = (N \mod numCoreRows)$$

The lowest column index of matrix A (i.e., the lowest row index from x) for which values from vector x are stored for multiplying is given by

$$\begin{aligned} colLow[i][j] = &\sum_{m=(i+1)}^{(numCoreRows-1)} \sum_{n=0}^{(numCoreCols-1)} numCols[m][n] \\ &+ \sum_{p=(j+1)}^{(numCoreCols-1)} numCols[i][p], \end{aligned} \tag{6.24}$$

$$i = 0, ..., (numCoreRows - 1);\ j = 0, ..., (numCoreCols - 1)$$

and the highest column index of matrix A (i.e., the highest row index from x) for which values from vector x are stored for multiplication is calculated from

$$colHigh[i][j] = colLow[i][j] + numCols[i][j] - 1, \tag{6.25}$$

$$i = 0, ..., (numCoreRows - 1);\ j = 0, ..., (numCoreCols - 1).$$

The *colLow* value for each row is

$$colLowRow[i] = colLow[i][numCoreCols - 1], \tag{6.26}$$

$$i = 0, ..., (numCoreRows - 1).$$

Just as with *SnakeSpMV*, in *RowSpMV* each core except the last passes x values downstream before storing them in its memory. For *RowSpMV* however, since the cores are arranged in a two dimensional grid, some values are passed east and some are passed south. Each core will pass a certain number of x values east as determined by

$$numValsToPassEast[i][j] = colLow[i][j] - colLow[i][numCoreCols - 1], \qquad (6.27)$$
$$i = 0, ..., (numCoreRows - 1);\ j = 0, ..., (numCoreCols - 1)$$

whereas only the left hand column of cores passes x values south and the number of values to pass is calculated using

$$numValsToPassSouth[i] = colLow[i][numCoreCols - 1];\ i = 0, ..., (numCoreRows - 1). \qquad (6.28)$$

There are six programs used for the case 1 *RowSpMV* kernel; *RowSpMVFirstCol*, *RowSpMVMiddle*, *RowSpMVLast, RowSpMVLastColFirstRow, RowSpMVFirstColLastRow,* and *RowSpMVLastCol. RowSpMVFirstCol* is run on cores of the first column, except for the core in the last row, and each has an index of (0 to (*numCoreRows*–2), 0). *RowSpMVMiddle* runs on cores between the first and last column, and each has an index of (0 to (*numCoreRows*–1), 1 to (*numCoreCols*–2)). *RowSpMVLast* runs on the last core in the array and has the index ((*numCoreRows*–1), (*numCoreCols*–1)). *RowSpMVLastColFirstRow* runs on the core located in the last column of the first row and has index (0, (*numCoreCols*–1)), while *RowSpMVFirstColLastRow* runs on the core located in the first column of the last row and has index ((*numCoreRows*–1), 0). Finally, *RowSpMVLastCol* runs on cores of the last column between the first and last rows, and each has an index of (1 to (*numCoreRows*–2), (*numCoreCols*–1)). Table 6.1 shows the actions taken depending on the control key each core receives.

RowSpMVFirstCol is similar to *SnakeSpMVFirst* and its pseudocode is shown in Algorithm 3. *RowSpMVFirstCol* differs in that it passes x values both east and south, and sorts sparse matrix data differently. Cores running *RowSpMVFirstCol* do not receive control keys. If the column of the sparse data point is less than the smallest *colLow* of the current processing row, then the sparse data is passed south because the corresponding x vector value is in a lower processor row.

If the column index is greater than or equal to the processing row's *colLow* but less than the first core's *colLow*, then it is passed east because the corresponding x vector value is located to the east of this core. *RowSpMVFirstCol* passes control words east only, because all of the cores below it in the first processing column will always receive column data from their northern neighbors. If an end of row has been reached and a token is received (i.e., $colInd > colHigh$), the token is passed both east and south. If data has been accumulated in the core running *RowSpMVFirstCol* and a token is received, then a control key of 1 followed by the accumulated data is sent east. A token is also sent south so that the lower processing rows will flush their data.

The rest of the programs for *RowSpMV* utilize a similar column bounds check and multiply-accumulation method as shown in Algorithm 3. All the other programs receive control keys of either 0 or 1. A control key of 0 indicates the transmission of a token or column, followed by sparse data. If a key of 1 is received, accumulated data from that same sending core is received and accumulated data in the receiving core should be flushed. *RowSpMVMiddle* is identical to *SnakeSpMVMiddle* as described in Section 6.2.1. *RowSpMVLast* first checks the input key from its western neighbor then its northern neighbor, and control keys are received from both directions.

RowSpMVLastColFirstRow is similar to the *RowSpMVMiddle* program except that it does not pass x values downstream. If a control key of 1 is received, any accumulated data in *RowSpMVLastColFirstRow* is added to the incoming data from the west. A control key of 1 is then sent south followed by the accumulated data. If a control key of 0 is received, followed by a token, then a control key of 1 followed by any accumulated data this core has is sent south. If instead no accumulated data has been formed in this core when a token is received, then only a control key of 0 is sent south.

RowSpMVFirstColLastRow is similar to *RowSpMVFirstCol*, with the major difference being that it doesn't pass any x values or incoming sparse matrix data south; therefore, the incoming column is not compared against the smallest *colLow* of the last processing row.

RowSpMVLastCol's method for accumulating data depending on the keys or tokens received is similar to the one used for *RowSpMVLast*. However, *RowSpMVLastCol* passes keys and tokens, whereas *RowSpMVLast* does not. If an end of row has been reached and *RowSpMVLastCol* is passing accumulated data, then it sends a key of 1 south. If it does not have accumulated data, it sends a key of 0 south instead.

Algorithm 3 Pseudocode of *RowSpMVFirstCol* (continued on next page)

```
for ($i = 0$; $i < numValsToPassSouth$; $i++$) do          \\ Pass x vector values south
    $Output_{south} \leftarrow Input$
end for
for ($i = 0$; $i < numValsToPassEast$; $i++$) do           \\ Pass x vector values east
    $Output_{east} \leftarrow Input$
end for
for ($i = 0$; $i < numCols$; $i++$) do                     \\ Store x vector values
    $Mem[i] \leftarrow Input$
end for
checkColumn:                                               \\ Check if $colHigh \leq colInd \geq colLow$
$column \leftarrow Input$                                  \\ Read column
if $column < colLowRow$ then                               \\ Check if $column < colLow$ for this processing row
    $Output_{south} \leftarrow column$                     \\ Pass column and data south, no control key needed
    $Outputt_{south} \leftarrow Input$
else
    $vecAddr \leftarrow column - colLow$                   \\ Calculate address of $vecVal$
    if $vecAddr \geq 0$ then                               \\ Check if $column \geq colLow$
        if $column > colHigh$ then
            $Output_{east} \leftarrow 0$                   \\ $Input == token$, so end of a row, pass token east
            $Output_{east,south} \leftarrow column$        \\ Pass column east and south
        else
            go to multiply                                 \\ $colHigh \leq colInd \geq colLow$ so go multiply $vecVal * matVal$
        end if
    else                                                   \\ $colInd < colLow$, so pass input east
        $Output_{east} \leftarrow 0$                       \\ Send key of 0, $column$, and $matVal$
        $Output_{east} \leftarrow column$
        $Output_{east} \leftarrow Input$
    end if
end if
go to checkColumn                                          \\ Check next input
```

Algorithm 3 Pseudocode of *RowSpMVFirstCol* (continued)

```
multiply:
$vecVal \leftarrow Mem[vecAddr]$                                  \\ Load $vecVal$ from memory
$accumSum \leftarrow Input * vecVal$        \\ Multiply $matVal$ (i.e., input) and $x$ value, overwrite $accumSum$
go to checkColumnDataPresent
multiplyAndAccumulate:
$vecVal \leftarrow Mem[vecAddr]$                                  \\ Load $vecVal$ from memory
$product \leftarrow Input * vecVal$                  \\ Multiply $matVal$ (i.e., input) and $x$ value
$accumSum \leftarrow accumSum + product$                                \\ Accumulate
checkColumnDataPresent:
$column \leftarrow Input$                                               \\ Read next input
$vecAddr \leftarrow column - colLow$
    \\ Data are sent in order, so no need to check if $colInd < colLow$
if $column > colHigh$ then                    \\ Check if $colInd > colHigh$, if so then input is a token
    $Output_{east} \leftarrow 1$                    \\ $Input == token$, so end of a row, pass $accumSum$ east
    $Output_{east} \leftarrow accumSum$
    $Output_{south} \leftarrow column$                                    \\ Send column south
else
    go to multiplyAndAccumulate          \\ $colHigh \leq colInd \geq colLow$ so multiply $vecVal * matVal$
end if
go to checkColumn                                                       \\ Check next input
```

6.3.2 Case 2

Case 2 refers to when the vector x doesn't fit on chip and Equation 6.7 holds true. As with *SnakeSpMV*, the equations for case 2 are similar to those for case 1. The x values are received via multiple regular loads and one last load.

The number of x values stored in a specific core on a regular load is calculated as

$$numColsInit[i][j] = \left\lfloor \frac{numColsPerRowInit[i]}{numCoreCols} \right\rfloor + (j < k), \tag{6.29}$$

$$i = 0, ..., (numCoreRows - 1);\ j = 0, ..., (numCoreCols - 1);$$

$$k = (numColsPerRowInit[i] \mod numCoreCols)$$

where the number of x values that will be stored in a processing row for a regular load is given by

$$numColsPerRowInit[i] = \left\lfloor \frac{numValsRegLoad}{numCoreRows} \right\rfloor + (i < k), \tag{6.30}$$

$$i = 0, ..., (numCoreRows - 1);\ k = (numValsRegLoad \mod numCoreRows)$$

and $numValsRegLoad$ is the total number of x values stored across all cores on a regular load and is calculated the same way as *SnakeSpMV* case 2 using Equation 6.11.

The lower column index of matrix A values that are multiplied by values of x in core (i,j)

initially is given by

$$\begin{aligned} colLowInit[i][j] = &\sum_{m=(i+1)}^{(numCoreRows-1)} \sum_{n=0}^{(numCoreCols-1)} numColsInit[m][n] \\ &+ \sum_{p=(j+1)}^{(numCoreCols-1)} numColsInit[i][p], \\ &i = 0, ..., (numCoreRows-1);\ j = 0, ..., (numCoreCols-1) \end{aligned} \tag{6.31}$$

and the upper column index of matrix A values that are multiplied by values of x in a core initially is given by

$$\begin{aligned} &colHighInit[i][j] = colLowInit[i][j] + numColsInit[i][j] - 1, \\ &i = 0, ..., (numCoreRows-1);\ j = 0, ..., (numCoreCols-1). \end{aligned} \tag{6.32}$$

Both $colLowInit[i][j]$ and $colHighInit[i][j]$ are incremented by $numValsRegLoad$ after each load.

The initial $colLow$ value for each row is

$$\begin{aligned} colLowRowInit[i] = colLowInit[i][numCoreCols-1], \\ i = 0, ..., (numCoreRows-1); \end{aligned} \tag{6.33}$$

and is incremented by $numValsRegLoad$ after each load.

The number of x values to pass east before storing values from the x vector in a core initially is given by

$$\begin{aligned} numValsToPassEastInit[i][j] = colLowInit[i][j] - colLowInit[i][numCoreCols-1], \\ i = 0, ..., (numCoreRows-1);\ j = 0, ..., (numCoreCols-1). \end{aligned} \tag{6.34}$$

The number of incoming x values that cores in the first column pass south on a regular load is

$$numValsToPassSouthInit[i] = colLowInit[i][numCoreCols - 1], \qquad (6.35)$$
$$i = 0, ..., (numCoreRows - 1).$$

The number of x values stored in a specific core on the last load is calculated differently than for a regular load to account for when $((N \mod numLoads) \mathrel{!}= 0)$; the equation is

$$numColsLast[i][j] = \left\lfloor \frac{numColsPerRowLast[i]}{numCoreCols} \right\rfloor + (j < k), \qquad (6.36)$$
$$i = 0, ..., (numCoreRows - 1);\ j = 0, ..., (numCoreCols - 1);$$
$$k = (numColsPerRowLast[i] \mod numCoreCols)$$

where the number of x values that will be stored in a row for the last load is

$$numColsPerRowLast[i] = \left\lfloor \frac{numValsLastLoad}{numCoreRows} \right\rfloor + (i < k), \qquad (6.37)$$
$$i = 0, ..., (numCoreRows - 1);\ k = (numValsLastLoad \mod numCoreRows)$$

and $numValsLastLoad$ is the total number of x values across all cores on the last load and is the same as Equation 6.18.

The lower column index of matrix A values that are multiplied by values of x in a core for the last load is found using

$$colLowLast[i][j] = \sum_{m=(i+1)}^{(numCoreRows-1)} \sum_{n=0}^{(numCoreCols-1)} numColsLast[m][n] \qquad (6.38)$$
$$+ \sum_{p=(j+1)}^{(numCoreCols-1)} numColsLast[i][p],$$
$$i = 0, ...(numCoreRows - 1);\ j = 0, ..., (numCoreCols - 1)$$

and the upper column index of matrix A values that are multiplied by values of x in a core for the last load is determined by

$$colHighLast[i][j] = colLowLast[i][j] + numColsLast[i][j] - 1, \tag{6.39}$$
$$i = 0,...,(numCoreRows - 1);\ j = 0,...,(numCoreCols - 1).$$

The final *colLow* value for each row is

$$colLowRowLast[i] = colLowLast[i][numCoreCols - 1], \tag{6.40}$$
$$i = 0,...,(numCoreRows - 1).$$

The number of x values to pass east before storing values from the x vector in a core for the last load is

$$numValsToPassEastLast[i][j] = colLowLast[i][j] - colLowLast[i][numCoreCols - 1], \tag{6.41}$$
$$i = 0,...,(numCoreRows - 1);\ j = 0,...,(numCoreCols - 1).$$

The number of incoming x values that a core passes south on the last load is

$$numValsToPassSouthLast[i] = numValsLastLoad - numColsPerRowLast[i], \tag{6.42}$$
$$i = 0,...,(numCoreRows - 1)$$

and this applies to cores in the first column only.

At the end of each row a token equal to N is sent. Loading and sending of data is performed using the same method described in Section 6.2.2 for *SnakeSpMV* case 2.

6.4 Parallel Subarrays

To increase the amount of parallel processing versus the *SnakeSpMV* and *RowSpMV* kernels, the parallel subarrays kernel includes multiple processing rows that perform multiplications in parallel. As shown in Figure 6.4, these processing rows are fed using a distribution network to evenly supply data followed by a sorting network for data routing. Following the multiplication, the results are summed in the accumulation network and sent off chip.

6.4.1 Distribution Network

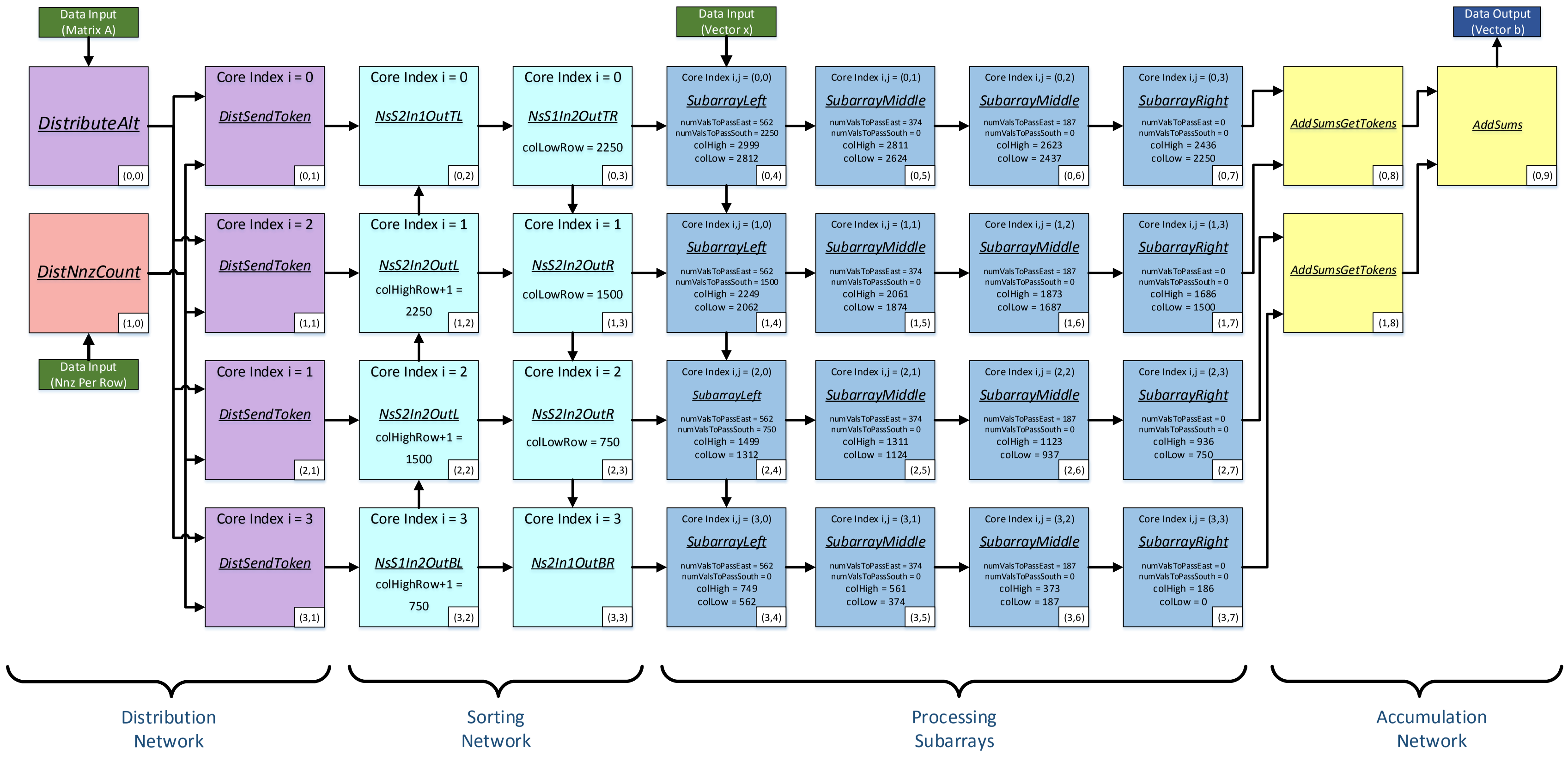
For parallel subarrays, column indexes are received, followed by the corresponding nonzero matrix value (e.g., *colInd[0]*, *matVal[0]*, *colInd[1]*, *matVal[1]*, etc.). The distribution network receives this sparse data and distributes it to the sorting network as shown in Figure 6.4. The goal of distributing the incoming data to multiple entry points in the sorting network is to parallelize the comparison of column indexes against the upper and lower column boundaries. Each distribution core evenly distributes data to at most four downstream cores by alternating its output direction after each transmission. The maximum number of output directions per core is limited to four for this application to allow easier routing; however, each core supports up to eight output directions. Each core of the distribution network's far-right column (running *DistSendToken*) receives the number of nonzeros (NNZ) that it should pass to the sorting network from the nonzeros count distributor. When a Butterfly sort is used, the far-right column of the distribution network is eliminated and the nonzeros count distributor communicates with the first column of the sorting network.

For each nonzero, the *DistSendToken* cores send the column index and matrix data value. Once finished sending the appropriate NNZ, these cores send a token of *-1*. This technique of sending NNZ to each final distribution core (i.e., each core running *DistSendToken*) increases the bandwidth through this network because data passing and control instruction processing occur in parallel, rather than sequentially.

The number of stages in the distribution network depends on the number of processing rows as follows:

$$numDistributionStages = \frac{1}{4} * (2 * log_2(numProcessingRows) - (-1)^{log_2(numProcessingRows)} + 5). \tag{6.43}$$

If a Butterfly sorting network is used, then the number of distribution stages is reduced by one and the final stage doesn't run the *DistSendToken* program; Section 6.4.3.2 presents an example of this case.



Data Input (Matrix A)
DistributeAlt (0,0)
DistNnzCount (1,0)
Data Input (Nnz Per Row)
Core Index i = 0 DistSendToken (0,1)
Core Index i = 2 DistSendToken (1,1)
Core Index i = 1 DistSendToken (2,1)
Core Index i = 3 DistSendToken (3,1)
Core Index i = 0 NsS2In1OutTL (0,2)
Core Index i = 1 NsS2In2OutL colHighRow+1 = 2250 (1,2)
Core Index i = 2 NsS2In2OutL colHighRow+1 = 1500 (2,2)
Core Index i = 3 NsS1In2OutBL colHighRow+1 = 750 (3,2)
Core Index i = 0 NsS1In2OutTR colLowRow = 2250 (0,3)
Core Index i = 1 NsS2In2OutR colLowRow = 1500 (1,3)
Core Index i = 2 NsS2In2OutR colLowRow = 750 (2,3)
Core Index i = 3 Ns2In1OutBR (3,3)
Data Input (Vector x)
Core Index i,j = (0,0) SubarrayLeft numValsToPassEast = 562 numValsToPassSouth = 2250 colHigh = 2999 colLow = 2812 (0,4)
Core Index i,j = (0,1) SubarrayMiddle numValsToPassEast = 374 numValsToPassSouth = 0 colHigh = 2811 colLow = 2624 (0,5)
Core Index i,j = (0,2) SubarrayMiddle numValsToPassEast = 187 numValsToPassSouth = 0 colHigh = 2623 colLow = 2437 (0,6)
Core Index i,j = (0,3) SubarrayRight numValsToPassEast = 0 numValsToPassSouth = 0 colHigh = 2436 colLow = 2250 (0,7)
Core Index i,j = (1,0) SubarrayLeft numValsToPassEast = 562 numValsToPassSouth = 1500 colHigh = 2249 colLow = 2062 (1,4)
Core Index i,j = (1,1) SubarrayMiddle numValsToPassEast = 374 numValsToPassSouth = 0 colHigh = 2061 colLow = 1874 (1,5)
Core Index i,j = (1,2) SubarrayMiddle numValsToPassEast = 187 numValsToPassSouth = 0 colHigh = 1873 colLow = 1687 (1,6)
Core Index i,j = (1,3) SubarrayRight numValsToPassEast = 0 numValsToPassSouth = 0 colHigh = 1686 colLow = 1500 (1,7)
Core Index i,j = (2,0) SubarrayLeft numValsToPassEast = 562 numValsToPassSouth = 750 colHigh = 1499 colLow = 1312 (2,4)
Core Index i,j = (2,1) SubarrayMiddle numValsToPassEast = 374 numValsToPassSouth = 0 colHigh = 1311 colLow = 1124 (2,5)
Core Index i,j = (2,2) SubarrayMiddle numValsToPassEast = 187 numValsToPassSouth = 0 colHigh = 1123 colLow = 937 (2,6)
Core Index i,j = (2,3) SubarrayRight numValsToPassEast = 0 numValsToPassSouth = 0 colHigh = 936 colLow = 750 (2,7)
Core Index i,j = (3,0) SubarrayLeft numValsToPassEast = 562 numValsToPassSouth = 0 colHigh = 749 colLow = 562 (3,4)
Core Index i,j = (3,1) SubarrayMiddle numValsToPassEast = 374 numValsToPassSouth = 0 colHigh = 561 colLow = 374 (3,5)
Core Index i,j = (3,2) SubarrayMiddle numValsToPassEast = 187 numValsToPassSouth = 0 colHigh = 373 colLow = 187 (3,6)
Core Index i,j = (3,3) SubarrayRight numValsToPassEast = 0 numValsToPassSouth = 0 colHigh = 186 colLow = 0 (3,7)
AddSumsGetTokens (0,8)
AddSumsGetTokens (1,8)
AddSums (0,9)
Data Output (Vector b)
Distribution Network
Sorting Network
Processing Subarrays
Accumulation Network

Figure 6.4: Parallel subarrays mapping with column based sorting network and $N = 3000$, with 16 cores used for the processing subarrays. The implementation consists of a distribution network, sorting network, processing subarrays, and an accumulation network. A count of the number of nonzeros for each row is received by the *DistNnzCount* core. A NorthSouth sorting network using column sorting routes data. Data from vector x are received in the top left corner of the processing subarrays and are stored in this group of cores as well. Each row computes dot products in parallel. Products are accumulated in the accumulation network before being sent off chip.

The number of cores per distribution stage is determined using the follow equation, where the number of cores in the last distribution stage is equal to the number of processing rows:

$$numCoresPerDistStage[i] = \left\lceil \frac{numCoresPerDistStage[i-1]}{4} \right\rceil, \tag{6.44}$$
$$i = (numDistributionStages - 2), ..., 0;$$
$$numCoresPerDistStage[i] = numProcessingRows,\ i = (numDistributionStages - 1).$$

6.4.2 Nonzeros Count Distributor

The nonzeros count distributor determines the number of nonzeros that each sorting network row will receive before a token should be passed downstream. The number of nonzeros per row is not always evenly divisible by the number of processing rows; therefore, part of the difficulty for this block is to efficiently determine which core in the distribution network last received data.

6.4.2.1 With Nonzeros Per Row Input (NPRNnzCntDist)

The nonzeros count distributor program, *DistNnzCount*, receives as input the number of nonzeros per row and determines the number of nonzero values that each distribution core running *DistSendToken* should receive before sending a token into the sorting network. If a Butterfly sorting network is used, then the *DistNnzCount* core communicates with cores in the first column of the sorting network, rather than *DistSendToken*.

Since the number of nonzero values per row is not always divisible by the number of processing rows, *DistNnzCount* must keep track of which core was the last to receive a matrix data value. The number of nonzeros each *DistSendToken* core or core in the first column of the Butterfly network receives is calculated as follows:

$$nnzPerDistCore[m][j] = \tag{6.45}$$
$$\left\lfloor \frac{nnz[m]}{numProcessingRows} \right\rfloor + j < (nnz[m] \mod numProcessingRows),$$
$$m = 0, ..., (M-1);\ j = (((0, ..., (numProcessingRows - 1)) + offset) \mod numProcessingRows);$$
$$offset[0] = 0;\ offset[m] = nnz[m-1] \mod numProcessingRows$$

where m represents the current row of the matrix A, j represents the row of the core receiving nonzeros counts, and *nnz[m]* is the number of nonzeros of the current row. Each time *DistNnzCount* sends *nnzPerDistCore*, the value of the offset changes so that the core sent to first for the next row is adjusted properly. As shown in Figure 6.4, the core index for the *DistSendToken* cores does not monotonically increase as the processing row increases because the distribution cores alternate the core to which they send data.

6.4.2.2 Parallel Nonzeros Count Distributor (ParNnzCntDist)

To speed up the process of determining the number of nonzeros that each core should receive, this technique uses multiple nonzeros count distribution cores. Two cores determine the *nnzPerDistCore* in parallel and alternate sending data to another core, which distributes the values. To remember which core will receive data first on the next matrix row, both nonzeros count distribution cores keep track of the last core to receive a nonzero value from matrix A.

6.4.2.3 Nonzeros Distributor With Padded Input (PadInputNnzCntDist)

This format involves padding each sparse matrix row with explicit zeros so that each row is a multiple of the number of processing rows. This reduces control overhead for determining how many nonzeros each core should receive and simplifies keeping track of which core last received data. For instance, with four processing rows, each matrix row would have an appropriate number of zero entries added to ensure that each sorting row receives the same number of inputs. Additionally, the x vector is padded with zero values to ensure that N is also an integer multiple of the number of processing rows. The nonzeros counter simply shifts the nonzeros count for each row to the right by $log_2(numProcessingRows)$, and sends this value, requiring a single instruction. This method involves a trade-off; although the control overhead is reduced with zero padding formatting, useless work is performed when processing zero inputs. The efficiency of this method increases with the density of the matrix A and decreases as the number of processing rows increases.

6.4.2.4 Table Based Nonzeros Distributor (TableNnzCntDist)

Another method for nonzeros count distribution uses a lookup table to increase throughput. The overhead of the method described in Section 6.4.2.1 comes from having to keep track of the last core to receive data because the NNZ for each row is not always evenly divisible by the number of processing rows. The additional number of values that do not evenly divide is given by

$$additionalNnzPerRow[m] = nnzPerRow[m] \mod numProcessingRows; \quad m = 0, ..., (M-1), \tag{6.46}$$

where m is the current row of A, M is the number of rows of A, and *nnzPerRow* is the number of nonzeros for the current row. From this formula, *additionalNnzPerRow* can take on the values 0 to $($*numProcessingRows*–$1)$.

For the table method, all possible combinations of additional nonzero values to send each direction are stored in memory, along with all possible rotations, where the number of values to store is

$$numTableVals = (numProcessingRows)^3. \tag{6.47}$$

An example of this table when four processing rows are used is shown in Table 6.2. The nonzeros distributor sends the nonzeros counts to each *DistSendToken* core of the distribution network; or when a Butterfly sorting network is used, to the cores in the first column of the sorting network. There are four directions to which the nonzeros distributor sends data, denoted as send direction 1–4 in the table, which correspond to the four processing rows. The nonzeros distributor determines the base number of nonzeros to send to each recipient by calculating

$$baseNnzPerRow[m] = \left\lfloor \frac{nnzPerRow[m]}{numProcessingRows} \right\rfloor; \; m = 0, ..., (M-1), \tag{6.48}$$

where m is the current row of A. The value of *additionalNnzPerRow[m]* is used to index into the table to determine which cores receive additional nonzeros; the value from the table is added to *baseNnzPerRow[m]* before sending the value to each recipient. The number of nonzeros are accumulated for each row to determine which rotation of the table to index to. For example, when first starting the program, if the first row contains 5 nonzeros, then *additionalNnzPerRow = 1* and the table rotation is 0, so send direction 1 receives an additional nonzero. The table rotation is

Table 6.2: Data for Nonzeros Table Based Distributor with Four Processing Rows.

		Additional Nonzeros Based on Table Rotation			
additionalNnzPerRow	Send Direction	Rotate by 0	Rotate by 1	Rotate by 2	Rotate by 3
0	1	0	0	0	0
	2	0	0	0	0
	3	0	0	0	0
	4	0	0	0	0
1	1	1	0	0	0
	2	0	1	0	0
	3	0	0	1	0
	4	0	0	0	1
2	1	1	0	0	1
	2	1	1	0	0
	3	0	1	1	0
	4	0	0	1	1
3	1	1	0	1	1
	2	1	1	0	1
	3	1	1	1	0
	4	0	1	1	1

based on the following equation:

$$tableRotation[m] = \left(\sum_{j=0}^{m-1} nnzPerRow[j] \right) \mod numProcessingRows; \; m = 0, ..., (M-1), \tag{6.49}$$

which masks the lower $log_2(numProcessingRows)$ bits of the accumulated number of nonzeros per row, such that the first table rotation is to rotate by 0; this value is unaffected by rollover when stored as an unsigned integer.

Since Table 6.2 consists of 16 values per rotation, a total of 64 values are stored, and the four right columns of the table are stored in column major order. The method for indexing into this table is

$$\begin{gathered}
tableAddress[m] = (tableRotation[m] << (2 * log_2(numProcessingRows))) \mid \\
(additionalNnzPerRow[m] << log_2(numProcessingRows)) + offset, \\
m = 0, ..., (M-1);\ offset = 0, ..., (numProcessingRows - 1)
\end{gathered} \tag{6.50}$$

where m represents the current row of A and the *offset* value changes depending on sending direction; for instance when there are four processing rows, the *offset* ranges from 0–3 for send directions 1–4.

When there are eight processing rows, the table must store 512 values, which can be done using two cores which store 256 table values each. Each core receives a copy of the *nnzPerRow[m]* value. This method requires using an entire core's data memory for table storage, never reading from both memory banks on any instruction, and using bypass registers to avoid using temporary storage. One core sends nonzeros counts to the first four processing rows while the other core sends nonzeros counts to the remaining four processing rows.

When there are eight processing rows, the nonzeros count distribution can also occur in parallel if an additional copy of the table is stored, requiring four cores to store two copies of the table. The nonzeros count distribution cores each receive a copy of the *nnzPerRow[m]* value, keep track of the table rotation, accumulate the NNZ and alternate sending the number of nonzeros per processing row. Following these four cores are two more cores which receive the input from the four NNZ distributor cores. Each of these receiving cores pass data to either the last stage of the distribution network or the first stage of a Butterfly sorting network. The upper four sorting rows receive data from one of these cores, while the bottom four sorting rows receive NNZ data from the other core. For a larger number of processing rows, additional methods could also be used where more than 1 table value is stored per row, although this would likely require masking and shifting the relevant bit from the table entry.

6.4.3 Sorting Network

The sorting network receives sparse matrix data from the distribution network and sorts the values based on the column index before sending them to the appropriate processing subarray. Two sorting network types are explored, NorthSouth sort and Butterfly sort.

6.4.3.1 NorthSouth Sort

The NorthSouth sorting network sorts data in an arc fashion. For sorting on column, the first column passes data north if the column index is greater than current processing row's *colHigh*, whereas the second column passes data south if the column index is less than a processing row's *colLow*. For sorting on LSBs, the *numLSBsToSortOn* (defined in Equation 6.52) LSBs of the column index are compared to the *LSBSortVal* (defined in Equation 6.53) of the processing row, and sorting occurs in a similar fashion, as discussed in Section 6.4.3.1.2. Cores in both columns pass data east otherwise.

6.4.3.1.1 Sort on Column (NSColSort) The NorthSouth sorting network based on column sorting includes six programs, *NsS2In1OutTL*, *NsS1In2OutTR, NsS1In2OutBL*, *NsS2In1OutBR*, *NsS2In2OutL*, and *NsS2In2OutR*. A parallel subarray kernel implementation with column based NorthSouth sorting is shown in Figure 6.4. As each program name implies, the cores in the network accept input from either one or two directions and output data to either one or two directions. If a core does not receive input from either direction, it stalls to save power. Each core must receive a token from each of its input directions before it passes a token, and each core sends a token to each of its output directions.

NsS2In1OutTL runs on the core in the top left corner of the NorthSouth sorting network and receives input from the west and south directions and passes that data east. Once this core receives a token (marking the end of a row) from either input direction it will exclusively monitor the other direction for input. Once it has received a token from both input directions it will send a token east.

NsS1In2OutTR runs on the top right core of the sorting network and receives input from the west direction and passes it west and/or south. If the column index of a data point is less than *colLowRow*, then the column and data value are passed south, otherwise they are passed east. Upon receiving a token, a token is passed in both output directions and a data value of zero is passed east because each processing row expects each input to be two data words (column or token, and data value).

NsS1In2OutBL executes on the bottom left core, receives input from the west, and passes data north and/or east. If the input column is less than *colHighRow+1* then it and the following input are sent east, otherwise they are sent north. Once a token has been received, it is sent to both output directions.

NsS2In1OutBR runs on the bottom right core of the sorting network, receives inputs from the west and north, and passes data east. Once this core receives a token (marking the end of a row) from either input direction, it will exclusively monitor the other direction for input. Once it has received a token from both input directions it will send a token east followed by a data value of *0*.

NsS2In2OutL runs on the cores in the left hand column of the network between the first and last rows. Cores with this program receive input from the west and south and pass data north and east. If the input column is less than *colHighRow+1* then it and the following input are sent east, otherwise they are sent north. Once this core receives a token (marking the end of a row) from either input direction it will exclusively monitor the other direction for input. Once it has received a token from both input directions it will send a token to both output directions.

NsS2In2OutR runs on the cores in the right hand column of the network between the first and last rows. These cores receive input from the north and west and output data to the east and south. If the column index of a data point is less than *colLowRow*, then the column and data value are passed south, otherwise they are passed east. Once this core receives a token from either input direction, it will exclusively monitor the other direction for input. Upon receiving a token from both input directions, a token is passed in both output directions and a data value of zero is passed east.

The NorthSouth sorting network uses Equation 6.26 and the following, where i represents the row of the core in the sorting network:

$$\begin{gathered} colHighRow[i] = colHigh[i][0], \\ i = 0, ..., (numCoreRows - 1); \end{gathered} \tag{6.51}$$

colHighRow+1 is calculated for the cores in the left hand column, while *colLowRow* is calculated for the cores in the right hand column, and *NsS2In1OutTL* and *NsS2In1OutBR* do not use either of these values. In order to calculate *colLowRow* and *colHighRow+1* for the sorting network, the values from Equations 6.22 through 6.27 must first be calculated for the processing subarray. The variables i and j for those equations refer to the row and column of the cores in the processing subarray only.

6.4.3.1.2 Sort on LSBs (NSLsbSort) Another method for routing sparse matrix data is to sort on the LSBs of the column index. Sorting on LSBs allows the x vector values to be stored based upon the LSBs of the vector row they correspond to, thereby forcing the values from contiguous rows to be stored in separate processing rows. If the sparse data matrix contains values that are grouped, then this method helps evenly distribute data to all of the processing rows to perform multiplications in parallel.

The number of LSBs to sort on depends on the number of processing rows, as shown by the equation

$$numLSBsToSortOn = log_2(numProcessingRows). \tag{6.52}$$

For example, when there are four processing rows, sorting occurs on the two LSBs of the column index.

The NorthSouth sorting network based on using LSBs includes six programs, *NsSLSBs2In1OutTL*, *NsSLSBs1In2OutTR, NsSLSBs1In2OutBL*, *NsSLSBs2In1OutBR*, *NsSLSBs2In2OutL*, and *NsSLSBs2In2OutR*.

NsSLSBs2In1OutTL is identical to the column based sort program *NsS2In1OutTL* from Section 6.4.3.1.1, except that sorting is based on the *numLSBsToSortOn* LSBs of the column index.

NsSLSBs1In2OutTR receives a column or token input from the west. If it receives a token, it passes it to the east and south, and then also passes a data value of zero east. If instead a column is received, the *numLSBsToSortOn* LSBs of the column index is compared to *LSBSortVal*, which is equal to the processing row as shown below,

$$LSBSortVal[i] = ProcessingRow[i];\ i = 0, ..., (numProcessingRows - 1). \tag{6.53}$$

The *numLSBsToSortOn* LSBs of the column index are masked and compared against *LSBSortVal*, which in this case has a value of 0; if the *numLSBsToSortOn* LSBs are greater than 0, then the column index and data value are passed south. If they are equal to 0, then the column is right shifted by *numLSBsToSortOn* and sent east, along with the data value. The column is right shifted because the processing rows store the x vector values based upon the column values of A (row values of x) shifted by *numLSBsToSortOn*. For instance, with four processing rows, the x value with row

36 would get stored in the first processing row because its two LSBs are equal to 0, and this x value would get stored at core memory index 9.

NsSLSBs1In2OutBL is similar to the column based sort program *NsS1In2OutBL,* except that sorting is based on the *numLSBsToSortOn* LSBs of the column index. If the *numLSBsToSortOn* LSBs are equal to *numProcessingRows-1*, then the column index and matrix data value are sent east, otherwise they are sent north.

NsSLSBs2In1OutBR is similar to *NsS2In1OutBR* except that if a column index is received, then it is right shifted by *numLSBsToSortOn* before being sent to the processing subarray row. The columns of the sparse matrix for this processing row will have their *numLSBsToSortOn* LSBs equal to *numProcessingRows-1.*

NsSLSBs2In2OutL runs on the left hand column of cores in the sorting network, between the first and last rows. It operates similar to *NsS2In2OutL*, except that sorting is based on the *numLSBsToSortOn* LSBs of the column index. If the *numLSBsToSortOn* LSBs of the column index are less than *LSBSortVal,* then the input column and matrix value are routed north, otherwise they are sent east.

NsSLSBs2In2OutR runs on the right hand column of cores in the sorting network, between the first and last rows. It operates similar to *NsS2In2OutR*, except that it sorts based on LSBs. If the *numLSBsToSortOn* LSBs of the input column index are greater than *LSBSortVal*, then data is routed south, otherwise it is sent east after shifting the column index right by *numLSBsToSortOn.*

6.4.3.2 Butterfly Sort (BSort)

To increase the number of parallel comparisons and to simplify the comparison code, Butterfly sorts were implemented and evaluated. Similar to a technique used with the NorthSouth sorting network in Section 6.4.3.1.2, the Butterfly sorting method sorts using the LSBs of the column index. The number of stages for the Butterfly network depends on the number of processing rows as follows:

$$numButterflyStages = log_2(numProcessingRows) + 1. \tag{6.54}$$

The number of LSBs to sort on is also dependent on the number of processing stages as shown by Equation 6.52.

Figure 6.5 shows that the first *numButterflyStages-1* stages of the Butterfly network sort using individual bits, while the last stage, *Get2Send1*, passes data to the processing subarrays. When using the Butterfly sorting network, one stage of the distribution network is eliminated. Each processing row multiplies in parallel.

Get2Send1 receives sparse data and tokens from the previous stages of the Butterfly sorting network and passes them to the processing subarrays. Once a token has been received from one input direction, data is exclusively handled for the input from the other direction until another token is received, at which point a single token is transmitted to the processing subarrays followed by a data value of zero since the processing subarrays expect two data words per input. The *colLow* and *colHigh* values are identical for each processing row because *Get2Send1* shifts the column indexes right by two bits before sending them to the processing subarrays.

The second to last stage uses the program *SortOnLSB* which sorts on the LSB of the column index. Cores running this program monitor both inputs for data and route based on the LSB. If the LSB is 1, the column index and matrix value are passed right, otherwise they are sent east. Once a token has been received, input is exclusively accepted from the other input direction until another token is received, at which point a token will be sent to both output directions. Before the stages with *Get2Send1* and *SortOnLSB* cores, there are *numButterflyStages-2* stages for sorting on the next most significant LSBs.

The first stage of the Butterfly sorting network receives sparse matrix data from the distribution network and the NNZ to expect from the nonzeros count distribution before sending a token to both output directions. The first stage sorts data based upon the *numLSBsToSortOn-1* bit of the column index, and passes data right if the bit is 1, and east otherwise.

Routing for the Butterfly sort is configured such that column indexes whose *numLSBsToSortOn* LSBs are 0 are sent to the first processing row, whereas those whose *numLSBsToSortOn* LSBs are equal to *numProcessingRows-1* are sent to the last processing row.

6.4.4 Processing Subarrays

The processing subarrays receive data and tokens from the sorting network, perform multiplication, and send the sums to the accumulation network. The x vector is received by the core

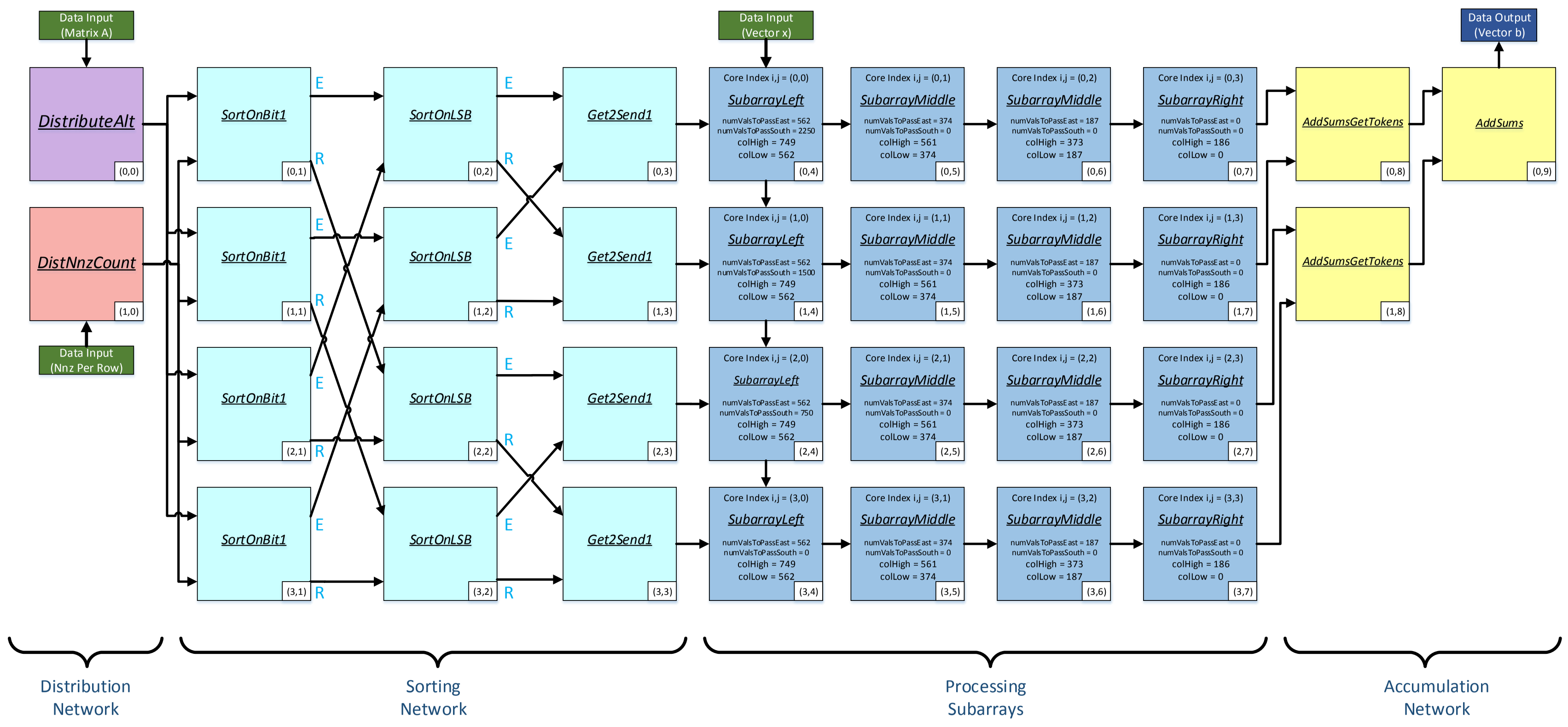


Figure 6.5: Parallel subarrays mapping with Butterfly sorting network and $N = 3000$, with 16 cores used for the processing subarrays. The implementation consists of a distribution network, sorting network, processing subarrays, and an accumulation network. A Butterfly sorting network uses the LSBs of the column index to route data to the appropriate processing row. Sending directions for each Butterfly sorting core output are labeled with "E" and "R" for east and right. Data from vector x are received in the top left corner of the processing subarrays and are stored in the processing subarrays cores. Products are accumulated in the accumulation network before being sent out.

in the top left of the array and distributed through the first column and rows by passing this data south and east, respectively. There are three programs for the processing subarrays, *SubarrayLeft*, *SubarrayMiddle*, and *SubarrayRight*.

SubarrayLeft runs on the cores in the first column, and has index (0 *to* (*numCoreRows*–1),0). It first receives x data and forwards the data both south and east, it then stores the appropriate x values in its memory. If the input column index is less than *colLow,* then the column and the data value are forwarded east; otherwise, the multiplication is performed. If the multiplication was performed, then a control key of -2 is sent east followed by the product. A control key of -2 is used to instruct all downstream cores to forward the product when they compare the key against *colLow.*

SubarrayMiddle runs on the cores between the first and last columns and rows and these cores have an index of (1 to (*numCoreRows*–2), 1 to (*numCoreCols–2)*). After passing x vector data east, these cores store x data in their memory. This program is similar to *RowSpMVMiddle* and *SnakeSpMVMiddle*, although it doesn't accumulate data and passes a control key of -2 before passing the products it has computed.

SubarrayRight runs on the cores in the last column, has index (0 to (*numCoreRows*–1), (*numCoreCols–1)*), and is identical to *SubarrayMiddle* except that it doesn't pass any x data values east. The outputs of this core will be either a control key of -2 followed by a product, or a token of -1 followed by a 0. The products are added in the accumulation network.

6.4.4.1 Processing Subarrays When Sorting on Column

When sorting on the input column, the equations for implementing the processing subarrays are identical to those for the *RowSpMV* kernel as discussed in Section 6.3.1; however, the processing subarrays pass x data south and east, and sparse matrix data is only passed east because the data has already been routed to the correct processing row with the sorting network.

6.4.4.2 Processing Subarrays When Sorting on LSBs

When sorting on a certain number of LSBs of the column index (via a NorthSouth sort on LSBs or Butterfly sort), the x vector is stored differently in memory than when sorting on columns, and the equations for *colLow* and *colHigh* must be modified to adjust for this. The equations for *numCols*, *numColsPerRow*, *numValsToPassEast*, and *numValsToPassSouth* are the same as those for the *RowSpMV* kernel from Section 6.3.1.

The x vector is stored in a vertically striped fashion, where the first value goes to row 0, the second value goes to row 1, etc., until an x value is stored in the last processing row and storing begins at row 0 again. Equivalently, this can be thought of as modifying the row indexes of the x vector values according to

$$xRowValLSBSort[i] = (i >> 2) + \sum_{j=1}^{ProcessingRow[i]-1} numColsPerRow[i], \quad (6.55)$$
$$i = 0, ..., (numProcessingRows - 1)$$

and the processing row that the x value will be stored in is given by

$$xValStorageRow[i] = (i \ \& \ (2^{numLSBsToSortOn} - 1)); \ i = 0, ..., (numProcessingRows - 1), \quad (6.56)$$

where i corresponds to the processing row.

The upper column index to compare against for each core now depends upon the number of columns stored in each core;

$$colHighLSBSort[i][j] = \sum_{m=0}^{j} numCols[i][m] - 1, \quad (6.57)$$
$$i = 0, ..., (numProcessingRows - 1); \ j = 0, ..., (numProcessingCols - 1)$$

and the lower column index to compare against becomes

$$colLowLSBSort[i][j] = colHighLSBSort[i][j] - numCols[i[[j] + 1, \quad (6.58)$$
$$i = 0, ..., (numProcessingRows - 1); \ j = 0, ..., (numProcessingCols - 1)$$

and i and j represent the core index within the processing array section of the parallel subarrays implementation.

6.4.4.3 Accumulation Network

The accumulation network adds the products from each of the processing subarray rows. This network has two programs, *AddSumsGetTokens* and *AddSums*.

The cores running *AddSumsGetTokens* receive inputs from two processing rows and accumulate them before sending the sum downstream to the cores running the *AddSums* program. *AddSumsGetTokens* runs on the cores in the first stage of the accumulation network. If a control key of -2 is received, the next input value is added to an accumulating sum. Once this core receives a token (marking the end of a row) from either input direction it will exclusively monitor the other input direction for input. Once it has received a token from both input directions it will output the accumulated sum; if no products were ever received, then a 0 is output.

AddSums runs on the remaining stages of the accumulation network beyond the first. Each of these cores receives an accumulated sum from each of its upstream neighbors, adds them, and outputs the result. If the number of processing rows is greater than four, then there will be multiple stages of cores running the *AddSums* program.

The number of stages in the accumulation network is dependent on the number of processing rows and is given by

$$numAccumulationStages = log_2(numProcessingRows). \tag{6.59}$$

The number of cores per accumulation stage halves for each subsequent stage and is determined using

$$numCoresPerAccumStage[m] = 2^{(numAccumulationStages - m - 1)}, \tag{6.60}$$
$$m = 0, ..., (numAccumulationStages - 1)$$

where k is the accumulation stage.

6.5 Parallel Arrays

Parallel arrays are useful for increasing throughput when the x vector can fit on chip multiple times and Equation 6.61 holds true, where

$$numVectorCopies = \left\lfloor \frac{numStorableVals}{N} \right\rfloor > 1. \tag{6.61}$$

Depending on the number of cores available on the platform, *numProcessingRows* can be as large as *numVectorCopies.* Each processing array stores a copy of the x vector and the distribution network alternates which processing row receives data. The parallel arrays kernel is similar to *SnakeSpMV* except that accumulation doesn't occur until the final processor, and the processing rows do not receive any end of row tokens to flush their data. The equations for implementing each row of the processing arrays section are identical to Equations 6.3–6.6 from the *SnakeSpMV* kernel.

Figure 6.6 shows a parallel arrays mapping with two parallel arrays for $N = 250$. This implementation uses two processing arrays and consists of three parts; the distribution network, the processing arrays, and the accumulation network. For the processing arrays there are three programs used, *PArrayFirst*, *PArrayMiddle*, and *PArrayLast.* Similar to *SnakeSpMV*, these programs first pass the x vector downstream before storing it in their local core memory. Each processing row receives a copy of the x vector to store.

The nonzero values are evenly distributed to the processing arrays by the *DistributeAlt* core, which alternates which processing row it sends data to from matrix A. Each core of the processing arrays has a nearly identical program except that *PArrayLast* doesn't include instructions for passing x vectors. After storing the x vector, *PArrayFirst*, *PArrayMiddle*, and *PArrayLast* compare their input column to *colLow.* If *colInd* is less than *colLow*, then *colInd* and the next input are passed east. If instead *colInd* lies between *colHigh* and *colLow*, then the multiplication occurs and a -1 token and the product are sent east. A token of -1 is used to ensure that all processing array programs pass products east, since -1 is less than any *colLow* value. *GetNnzAddSums* receives from *DistNnzCount* the number of values it should accumulate before passing the result to *AddSums.* Since *AddSums* expects to receive only products, *PArrayLast* doesn't pass any tokens it receives and also doesn't send a token after multiplying.

PArrayFirst runs on the first core of each processing row with core index 0. *PArrayMiddle* runs on the cores between first and last columns of each processing row and has core index (1 to (*numProcCoresPerRow*–2)), where *numCoresPerProcRow* is the number of cores per processing row and can be substituted for *numCores* in Equations 6.3–6.6. *PArrayLast* runs on the last core of each processing row and has index (*numProcCoresPerRow*–1).

The first stage of the accumulation network receives products from each core running the *PArrayLast* program and accumulates them before passing the result to *AddSums.* The number of inputs to accumulate is provided by the *DistNnzCount* program.

The number of stages for the distribution network is determined in a manner similar to the parallel subarrays method except that one less stage is used as shown below;

$$numDistributionStagesParallel = \lceil \frac{1}{4} * (2 * log_2(numProcessingRows) - (-1)^{log_2(numProcessingRows)} + 5) - 1 \rceil. \tag{6.62}$$

The number of cores per distribution stage is also computed similar to the parallel subarrays equation using

$$\begin{gathered} numCoresPerDistStageParallel[k] = \left\lceil \frac{numCoresPerDistStage[k-1]}{4} \right\rceil, \\ k = (numDistributionStages - 2), ..., 0; \\ numCoresPerDistStageParallel[k] = \frac{numProcessingRows}{4}, \\ k = (numDistributionStages - 1) \end{gathered} \tag{6.63}$$

where k represents the distribution stage.

The number of cores for the accumulation network is determined similarly to the parallel subarrays kernel. The number of stages in the accumulation network is dependent on the number of processing rows and is given by

$$numAccumulationStagesParallel = log_2(numProcessingRows) + 1. \tag{6.64}$$

The number of cores per accumulation stage halves for each subsequent stage and is

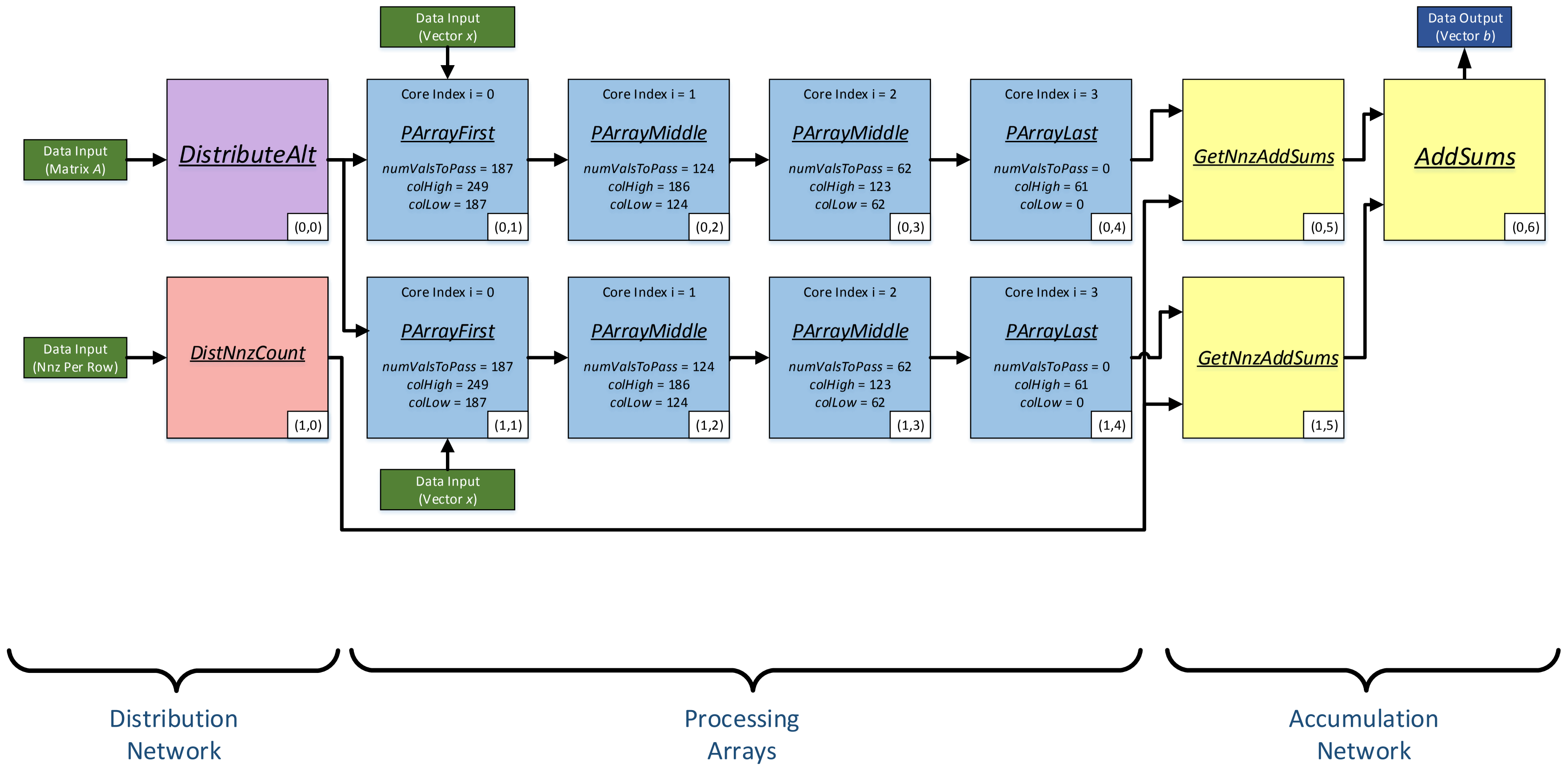


Figure 6.6: Parallel arrays mapping with two processing arrays and $N = 250$. The implementation consists of three parts, the distribution network, the processing arrays, and the accumulation network. The distribution network alternates to which processing row it sends data. For the processing arrays, the x vector values are evenly divided among the cores and identical copies are stored in both processing rows. The (row,column) physical id is shown in the bottom right hand corner of each core. The logical id is listed at the top of each core, and represents the variable i in Equations 6.3–6.6. The processing array cores are labeled with the number of x values to pass downstream, as well as the low and high column indexes. Following multiplication of the matrix value and vector value, products are accumulated in the accumulation network. The accumulation network *GetNnzAddSums* cores know how many inputs to accumulate before passing the sum to the *AddSums* core. The number of values to accumulate is provided by the *DistNnzCount* core.

determined using

$$numCoresPerAccumStage[k] = 2^{(numAccumulationStages-k-1)}, \tag{6.65}$$
$$k = 1, ..., (numAccumulationStages - 1);$$
$$numCoresPerAccumStage[0] = numProcessingRows$$

where the first stage runs *GetNnzAddSums* and each subsequent stage runs *AddSums*, and *k* represents the distribution stage.

6.5.1 Nonzeros Count Distributor

Similar to the parallel subarrays kernels presented in Section 6.4, the parallel arrays are also capable of using various nonzeros count distributors including using the number of nonzeros per row, parallel distributors, padding the sparse matrix to match the number of processing rows, and using a lookup table based method.

6.6 Comparison of Sparse Matrix-Vector Multiplication Methods

The throughput, power dissipation, and area for the SpMV implementations was determined. The SpMV implementations include the kernels on the many-core platform, and the SpMV implementations on the CPUs and GPUs. Five different sparse matrices from distinct problem types as described in Section 2.8 were used for performing the SpMV operation. The throughput per watt and throughput per area results are plotted to analyze power and area efficiency for each design.

Throughput per watt is a useful metric for determining which design among multiple options is the most power efficient [116]. Throughput per area is also useful for choosing the most area efficient design, as increasing area decreases yield and increases the fabrication cost per chip. Throughput is calculated by dividing the total number of operations by the execution time, a common metric [117, 118], as shown in Equation 6.66:

$$Throughput = \frac{2 * NNZ}{execution\ time} \tag{6.66}$$

For each of the implementations, execution time is the time for the sparse matrix multiplication operation only [119]. For the CPU-based implementations, this excludes the time to read the sparse matrix A and dense vector x from a file, load them into memory, and save the result to a file. For the GPU-based implementations, this also excludes the time to load the data from the host memory (i.e., CPU) to the device memory (i.e., GPU) and vice versa. For the many-core implementations this excludes only the time to load the x vector into memory, but includes the time to load the sparse matrix A.

For the implementations on the CPUs and GPUs, area, power, and throughput are scaled to 32 nm values. Area is scaled using $1/\mathrm{S}^2$ scaling, and throughput and power are scaled using trends in gate delay and switching energy for 65–14 nm technology nodes [120].

Throughput data for the implementations on the many-core platform described in Section 5.1 are obtained with a cycle-accurate C++ simulator. Power measurements from the 32 nm CMOS PD-SOI fabricated chip are input to the simulator to obtain power data. Chip power measurements and core area are scaled by the relative increases determined from synthesis in 32 nm CMOS PD-SOI at 0.9 V and 1.8 GHz when adding a single-precision 32-bit FP adder and multiplier with denormal support and switching to a 32-bit datapath. Each core has 256 memory locations and up to 240 x vector values are stored per core. Each of the many-core implementation programs use up to 16 memory locations for storing data other than the x vector. Some programs use less memory, and therefore the cores on which these programs run can store more than 240 x values, but a maximum value of 240 is used to simplify partitioning x. Since a code scheduler is not yet written, the implementations on the many-core platform use unscheduled code; using scheduled code will decrease power, and increase throughput by at least 2–3x. Code optimization will increase the use of bypass registers, thereby freeing memory locations for storing x values.

While implementing SpMV for the CPUs and GPUs is relatively simple due to the availability of libraries, programming the kernels for performing SpMV on the many-core platform involves some difficulty as each of the programs are written in assembly and are unscheduled. Programming will become easier and throughput will increase once a scheduler and compiler are available. Since case 1 is targeted, each individual program must fit within a core's local instruction memory, while also ensuring that the data memory usage does not exceed the core's local memory. After receiving an input, the number of instructions to execute before sending an output must be small to ensure high throughput. Additionally, it must be determined when it is advantageous to add additional cores to increase throughput without reducing area and power efficiency. While it is not always possible to achieve; ideally, all cores are active and performing no redundant operations. In an attempt to increase the number of simultaneously active cores, more cores were added to some implementations to evenly distribute data. For some implementations, adding these additional cores increases power and area efficiency.

CPU throughput data is gathered from a multi-threaded C++ SpMV implementation using the Eigen library [88], and power consumption is obtained from measuring Intel Architecture (IA) core power while excluding uncore power using HWMonitor PRO [121]. All compiler optimizations are turned on. For each SpMV test, 8 independent threads operate on separate copies of cached data; this ensures that all processors are 100% utilized, which is observed from HWMonitor measurements. The CPUs used for SpMV are the Intel Core i7-3770 [122] and the Core i7-2630QM [123], whose specifications are shown in Table 6.5.

Each CPU SpMV test uses 100,000 iterations per thread. Sparse data is read into a *SparseMatrix* object and the dense vector into a *VectorXf* object, and the two are multiplied. When calculating throughput, the execution time for the multiplication step is divided by the number of iterations and threads used.

Table 6.5: Details of CPUs and GPUs Utilized for SpMV Comparisons.

Chip	Technology (nm)	TDP (W)	Area (mm^2)
Intel Core-i7 3770	22	77	160
Intel Core-i7 2630QM	32	45	216
NVIDIA GeForce GT 620 OEM	40	30	79
NVIDIA NVS 4200M	40	25	79

GPU throughput data is obtained from a C++ implementation utilizing CUDA and cuSPARSE, with all compiler optimizations turned on. Power consumption is estimated using half of the thermal design power (TDP/2) since the power usage for some workloads averages roughly TDP/2 [124]. Using TDP/2 for the GPU power consumption estimate is conservative since power monitors indicated CPU power usage was above TDP/2 during SpMV and therefore GPU power is likely to be above TDP/2 as well. Additionally, some sources indicate peak power usage as 1.5 times TDP [125]. Since more accurate power data is not available, using the most conservative number is the fairest approach when calculating efficiency. The GPUs used for SpMV are the NVIDIA GeForce GT 620 [126], and the NVS 4200M [127], whose specifications are shown in Table 6.5.

SpMV on the GPUs utilizes the cuSPARSE application program interface (API), and the programmer must allocate memory and copy data from the CPU to the GPU using CUDA API routines, perform sparse matrix multiplication, then copy the data from the GPU back to the CPU [89]. Sparse matrix data is first read from a file, stored in coordinate (COO) format, then converted to CSR format. The CSR data format and Hybrid (HYB) data formats were considered for storing the sparse matrix, since CSR is a commonly used data format for SpMV, and HYB was considered as it has been shown to provide higher performance than CSR [119]; however, early tests demonstrated that using CSR provides higher throughput for the GPUs considered, so it is the format utilized for this work. Multiple instances of the matrix and vector are copied from the host to the device and the sparse matrix operation is executed for 1000 iterations. The *cudaDeviceSynchronize()* function determines when the sparse matrix multiplication operation is complete. The time from executing *cusparseScsrmv()* on all instances of A and x until *cudaDeviceSynchronize()* returns is measured and divided by the number of instances and iterations to obtain average execution time.

6.6.1 Power and Area Efficiency Results

Figure 6.7 displays throughput per watt versus throughput per area for several *SnakeSpMV*, *RowSpMV*, CPU, and GPU implementations when performing sparse matrix vector multiplication using the Averous-epb1 sparse matrix. The minimum number of cores to store the x vector is first used for the *SnakeSpMV* and *RowSpMV* implementations, and then increased to create additional implementations. Increasing the number of cores for the *SnakeSpMV* and *RowSpMV* beyond what is required to store the x vector decreases both the throughput per watt and throughput per area. The optimal *SnakeSpMV* and *RowSpMV* implementations use the fewest cores to store the x vector. Modifying the *RowSpMV* implementation to use more rows than columns or vice versa provides lower throughput per watt than maintaining an equal number of rows and columns. Several of the *RowSpMV* implementations and the minimum size *SnakeSpMV* implementation provide greater power efficiency than the CPU and GPU based implementations.

The relatively low throughput per area of the *SnakeSpMV* and *RowSpMV* implementations motivated finding alternative kernels for sparse matrix vector multiplication. The parallel subarays

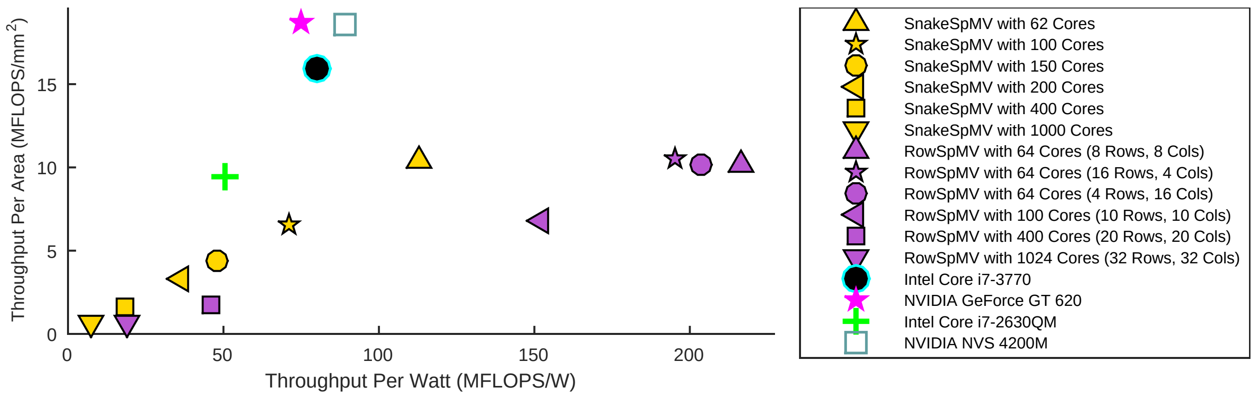


Figure 6.7: *SnakeSpMV* and *RowSpMV* compared against CPU and GPU implementations for the Averous-epb1 sparse matrix. The optimal *SnakeSpMV* and *RowSpMV* implementations use the fewest cores to store the x vector. Several of the *RowSpMV* implementations and the minimum size *SnakeSpMV* implementation provide greater throughput per watt than the CPU and GPU based implementations.

and parallel arrays kernels attempt to achieve this goal by increasing the amount of parallel processing with multiple processing rows.

The throughput per area versus throughput per watt for various SpMV implementations operating on different sparse matrices are shown in Figures 6.8–6.12. The most efficient designs provide the largest throughput per watt and throughput per area, therefore the most efficient designs are located near the upper-right corner and the least efficient designs are nearest the lower-left corner of each plot.

For the HB-gre_1107 sparse matrix shown in Figure 6.8, all of the many-core implementations explored are more efficient in terms of throughput per watt and throughput per area than the CPU and GPU implementations. The implementations on the many-core platform provide 3.41-36.6x higher throughput per watt and 1.66-27.9x higher throughput per area than the CPU and GPU designs. The most efficient implementation is the *Parallel Arrays w/ 1 Compute Row w/ NPRNnzCountDist*, and since N for this matrix is 1107, this implementation uses only six cores. While this implementation uses only one more core than *SnakeSpMV,* it provides nearly double the performance. The parallel arrays do not have to accumulate results until the final stage and have less control overhead than *SnakeSpMV*. The least efficient implementation is with the *NVIDIA NVS 4200M*. Although N for this matrix is half of the median N value for all matrices of the sparse matrix database, the GPU is unable to efficiently perform the SpMV operation. The other

GPU and CPU implementations provide similar area and power efficiency as the *NVIDIA NVS 4200M*. The most efficient implementations tend to be smaller and simpler, namely the parallel arrays, *SnakeSpMV* and *RowSpMV* designs. The least efficient many-core implementations tend to use more area and require cores for a sorting network, as exemplified in the parallel subarray implementations.

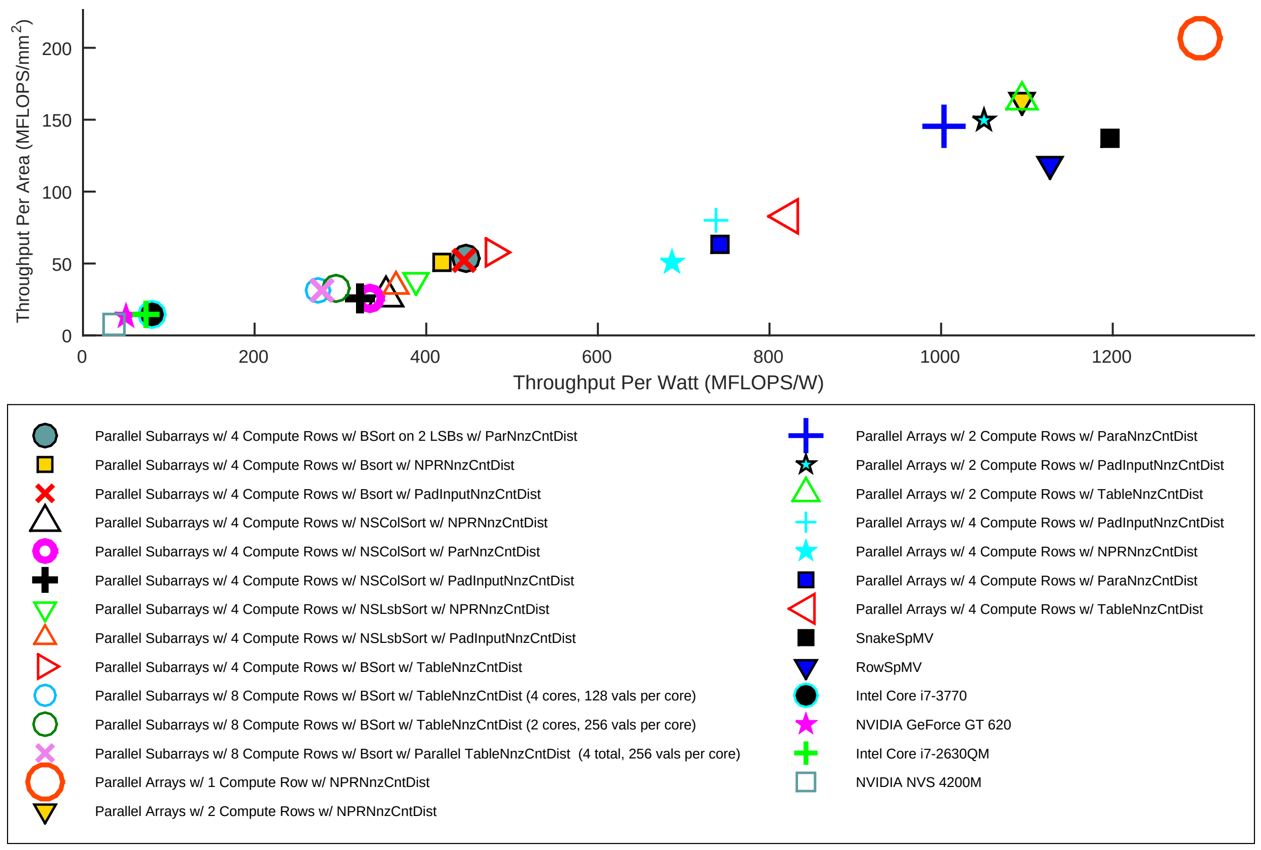


Figure 6.8: Throughput per watt versus throughput per area for various SpMV implementations operating on the HB-gre_1107 sparse matrix. *SnakeSpMV, RowSpMV,* parallel subarray and parallel array implementations are compared against CPU and GPU implementations. The optimal design has the largest throughput per watt and throughput per area. The implementations on the many-core platform provide 3.41-36.6x higher throughput per watt and 1.66-27.9x higher throughput per area than the CPU and GPU designs. Data for Figures 6.9–6.12 is gathered in the same manner described in Figure 6.7.

From Figure 6.9, all of the many-core implementations are once again more power and area efficient than the CPU and GPU implementations with the Bai-tols2000 matrix. These implementations provide 2.14–23.8x higher throughput per watt and 1.10–15.9x higher throughput per area than the CPU and GPU designs. The most area and power efficient implementation is again *Parallel Arrays w/ 1 Compute Row w/ NPRNnzCountDist.* The *NVIDIA NVS 4200M* provides

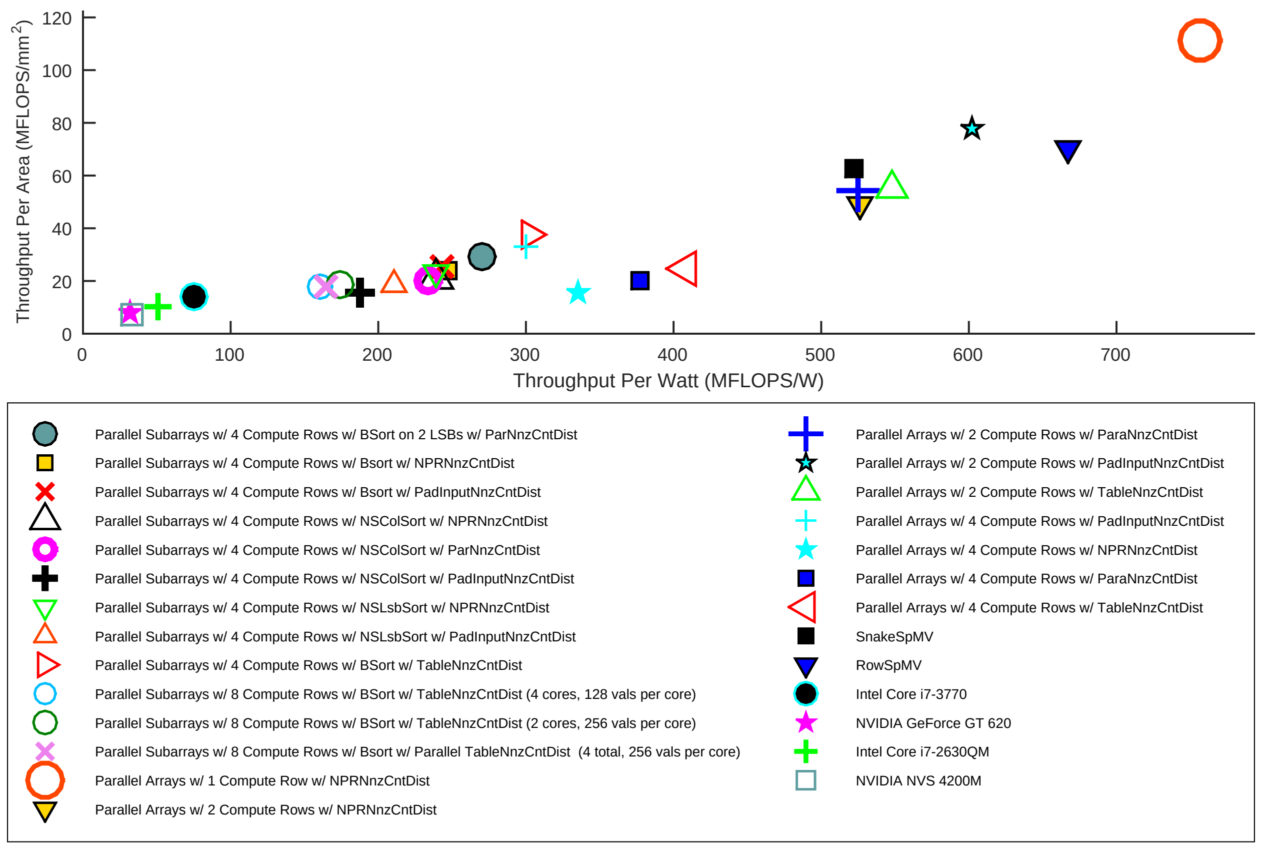


Figure 6.9: Throughput per watt versus throughput per area for various SpMV implementations operating on the Bai-tols2000 sparse matrix. *SnakeSpMV, RowSpMV,* parallel subarray and parallel array implementations are compared against CPU and GPU implementations. The optimal design has the largest throughput per watt and throughput per area. The implementations on the many-core platform provide 2.14–23.8x higher throughput per watt and 1.10–15.9x higher throughput per area than the CPU and GPU designs.

the lowest throughput per area, and the *NVIDIA GeForceGT 620* provides the lowest throughput per watt. Just as with Figure 6.8, the most efficient implementations are the simpler ones which do not have a sorting network (the *SnakeSpMV*, *RowSpMV*, and parallel arrays implementations). *Parallel Arrays w/ 2 Compute Rows w/ PadInputNnzCntDist* also provided a higher power and area efficiency. Therefore, padding the number of nonzeros per row equal to the number of computation rows provides higher efficiency (with the parallel arrays kernel using two compute rows) than using the *NPRNnzCountDist* method.

The plot in Figure 6.10 shows that all of the many-core implementations are more power efficient than the CPU and GPU implementations and most are also more area efficient for the Hamrle-Hamrle2 matrix. The many-core implementations provide 2.08–7.31x higher throughput per watt and 1.02–3.90x higher throughput per area than the CPU and GPU designs. The most area efficient implementation is *Parallel Arrays w/ 1 Compute Row w/ NPRNnzCountDist*, and the most power efficient is *RowSpMV.* The least area efficient implementation is *Parallel Arrays w/ 4 Compute Rows w/ NPRNnzCountDist,* and the least power efficient is with the *NVIDIA GeForce GT 620.* As N is slightly larger for this matrix, the efficiency of the simpler implementations decreases. The simpler implementations (which have less sophisticated or nonexistent sorting and nonzeros count distribution methods) increase core count to store the x vector. The area and power efficiency decreases as more cores are required for storing the x vector, and additional techniques must be used to increase throughput and offset this penalty. As N increases, the parallel subarray implementations become the optimal choice for achieving both high power and area efficiency, as shown in the following figures.

Figure 6.11 plots the throughput per area versus throughput per watt results when operating on the Averous-epb1 sparse matrix. With one exception, the implementations on the many-core platform provide higher throughput per watt than the CPU and GPU designs. The implementations on the many-core platform provide 1.03–4.28x higher throughput per watt and up to 2.50x higher throughput per area than the CPU and GPU designs. For this matrix, $N = 14734$ and the parallel subarrays provide the largest throughput per area and nearly the largest throughput per watt. Only the *RowSpMV* implementation provides a higher throughput per watt. The most area efficient implementation is *Parallel Subarrays w/ 8 Compute Rows w/ BSort w/ TableNnzCountDist* using two cores for the table. The least power efficient implementation uses the Intel Core i7-2630QM and the least area efficient implementation is *Parallel Arrays w/ 4 Compute Rows w/ NPRNnzCountDist.*

The results for the last matrix considered are plotted in Figure 6.12. Just as with the Averous-epb1 matrix, every implementation on the many-core platform, except one, is more power efficient than the CPU and GPU implementations. Also, the most power efficient implementation is *RowSpMV,* and the least power efficient implementation uses the Intel Core i7-2630QM. The implementations on the many-core platform provide 1.03–3.02x higher throughput per watt and up to 1.83x higher throughput per area than the CPU and GPU designs. The most area efficient implementation is *Parallel Arrays w/ 1 Compute Row w/ NPRNnzCountDist* and the least area efficient implementation is *Parallel Arrays w/ 4 Compute Rows w/ NPRNnzCountDist.*

For values of N near the median value of the sparse matrices in the sparse matrix database,

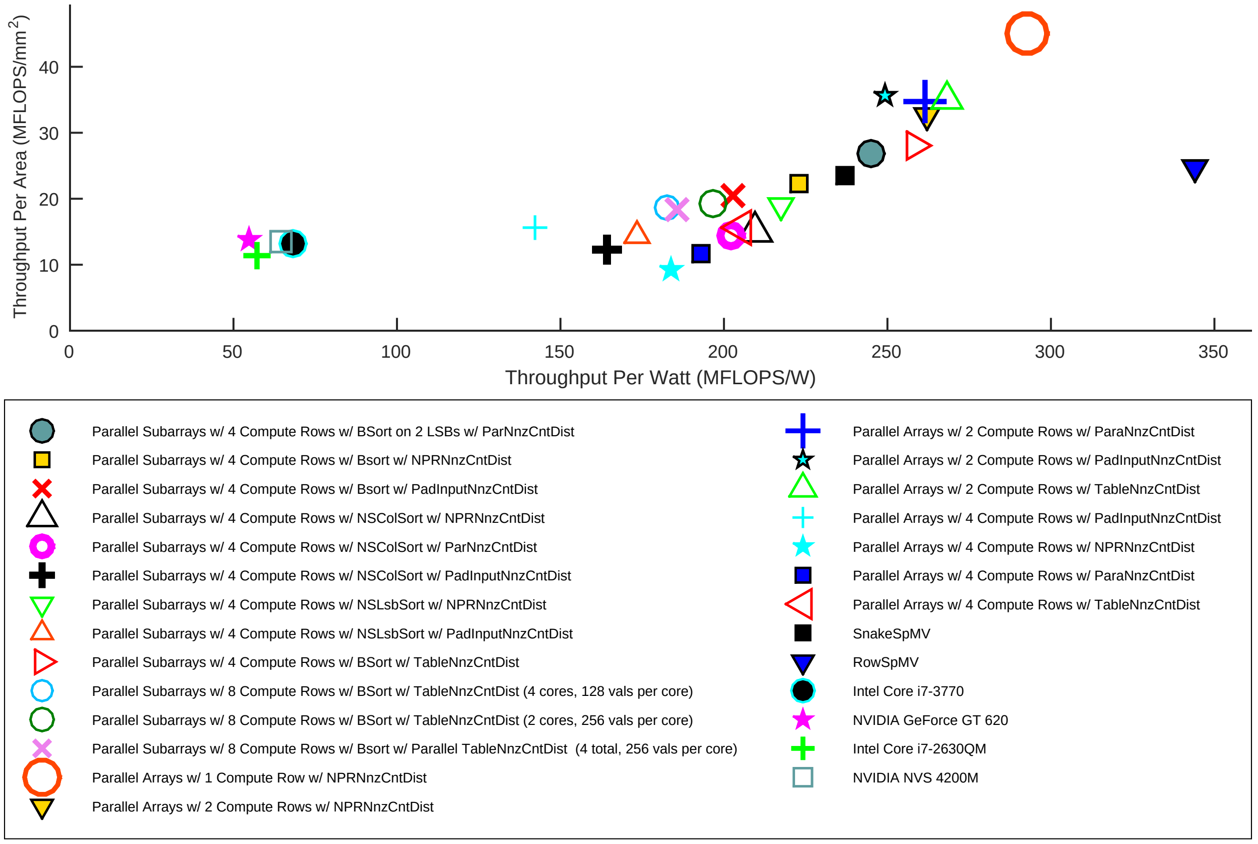


Figure 6.10: Throughput per watt versus throughput per area for various SpMV implementations operating on the Hamrle-Hamrle2 sparse matrix. *SnakeSpMV, RowSpMV,* parallel subarray and parallel array implementations are compared against CPU and GPU implementations. The optimal design has the largest throughput per watt and throughput per area. The implementations on the many-core platform provide 2.08–7.31x higher throughput per watt and 1.02–3.90x higher throughput per area than the CPU and GPU designs.

the simpler SpMV implementations (e.g., *SnakeSpMV, RowSpMV*) tend to provide the highest throughput per watt and throughput per area. As N increases, the most power and area efficient implementations typically use a parallel subarrays processing method with Butterfly sorting and a table method for nonzeros count distribution. As N increases, more memory is required to store the x vector, and higher throughput must be obtained by parallelizing the SpMV operation. For the matrices presented, which include those with N values greater and less than the median database value, the implementations on the many-core platform are capable of providing higher power and area efficiency than the CPU and GPU implementations.

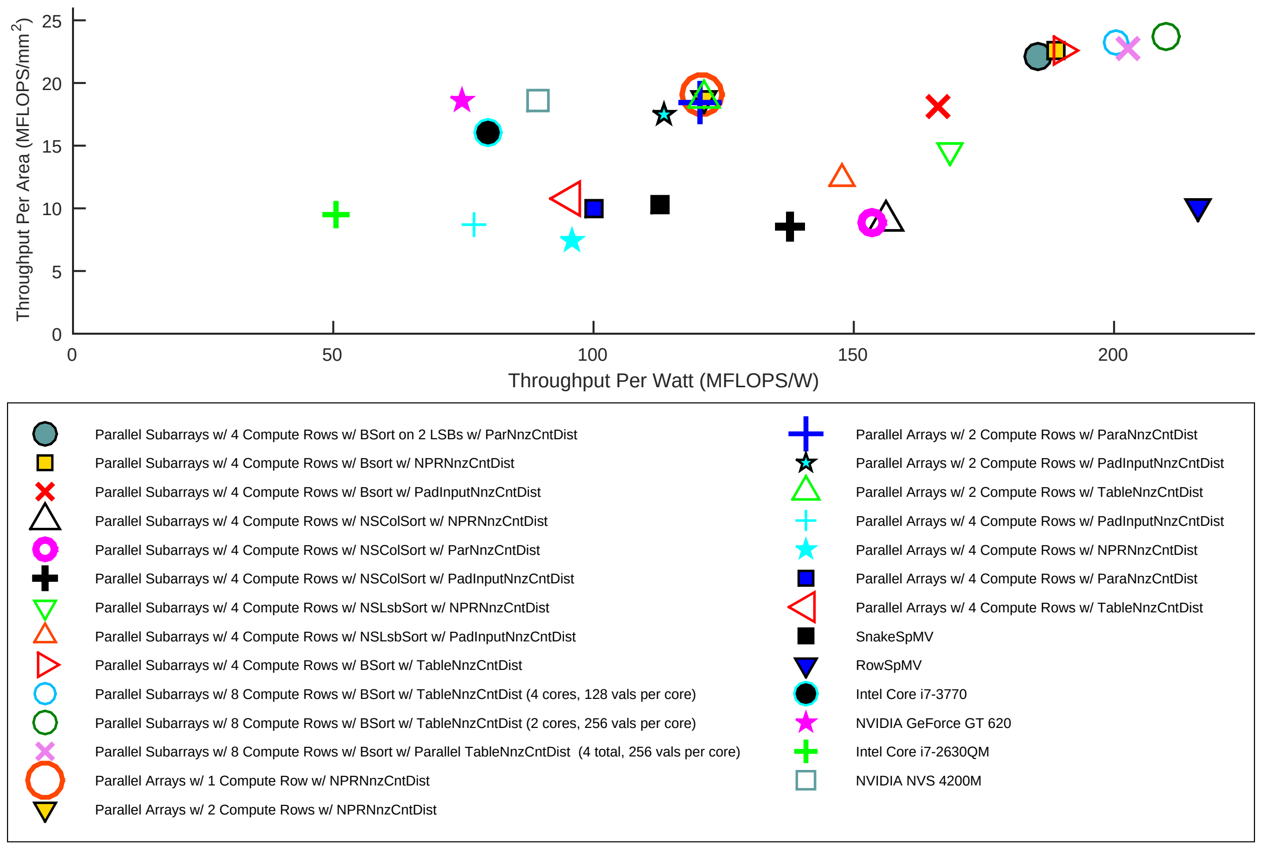


Figure 6.11: Throughput per watt versus throughput per area for various SpMV implementations operating on the Averous-epb1 sparse matrix. *SnakeSpMV*, *RowSpMV*, parallel subarray and parallel array implementations are compared against CPU and GPU implementations. The optimal design has the largest throughput per watt and throughput per area. The implementations on the many-core platform provide 1.03–4.28x higher throughput per watt and up to 2.50x higher throughput per area than the CPU and GPU designs.

6.6.2 Sorting and Nonzeros Count Distribution Power Efficiency Comparisons

NPRNnzCntDist is the baseline method for nonzeros count distribution, and *ParNnzCntDist*, *PadInputNnzCntDist*, and *TableNnzCntDist* are potential alternatives to improve area and power efficiency. Additionally, *NSColSort* is the baseline method for the sorting network, and *NSLsbSort* and *BSort* are alternative methods for improving area and power efficiency. For the parallel arrays implementations with two compute rows, *TableNnzCntDist* provides the largest throughput per watt improvements for each of the matrices, with improvements of up to 4.10%. Padding the NNZ to an integer multiple of the number of processing rows reduces power efficiency for a majority of cases. Using *ParNnzCntDist* also reduces power efficiency for the case with parallel arrays with two compute rows..

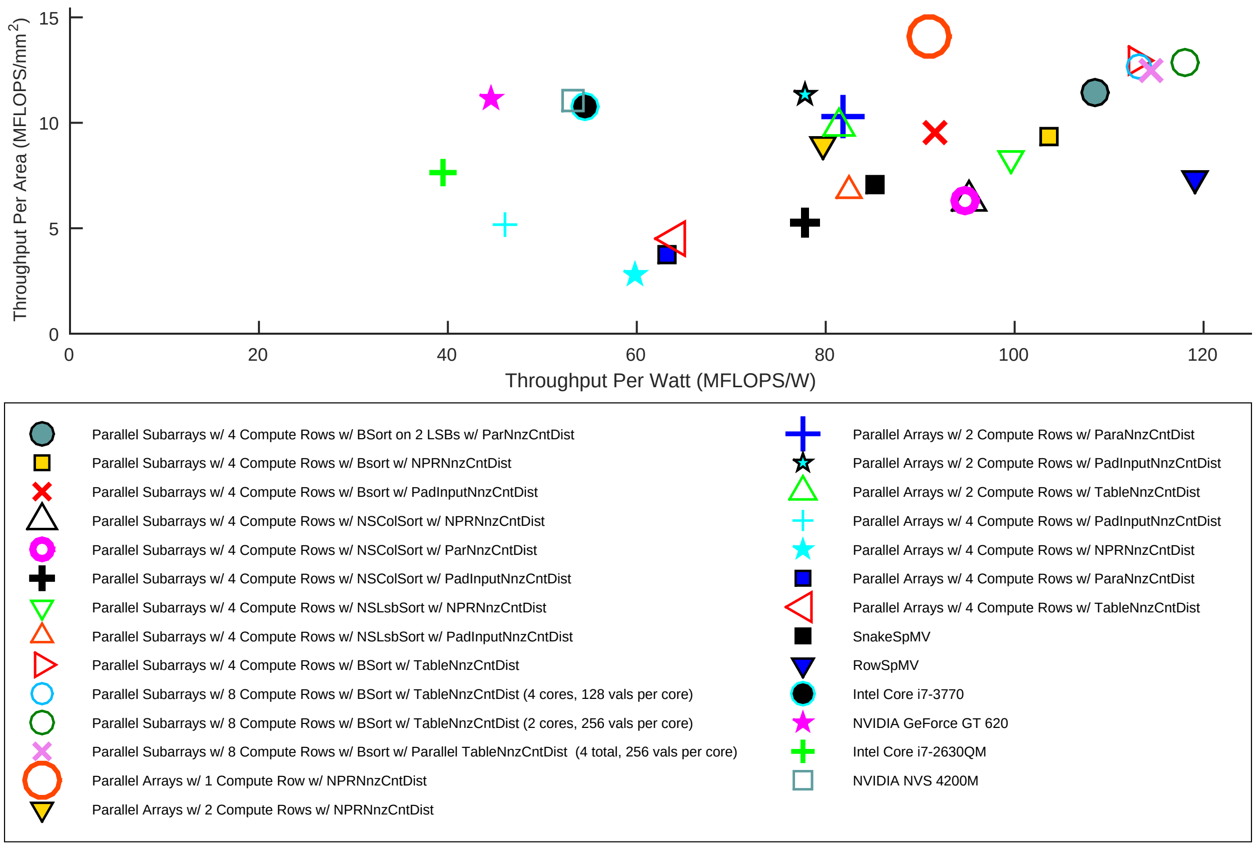


Figure 6.12: Throughput per watt versus throughput per area for various SpMV implementations operating on the Rommes-descriptor_xingo6u sparse matrix. *SnakeSpMV, RowSpMV,* parallel subarray and parallel array implementations are compared against CPU and GPU implementations. The optimal design has the largest throughput per watt and throughput per area. The implementations on the many-core platform provide 1.03–3.02x higher throughput per watt and up to 1.83x higher throughput per area than the CPU and GPU designs.

For the parallel arrays implementations with four compute rows, *ParNnzCntDist* and *TableNnzCntDist* are the most effective methods for improving throughput per watt (versus *NPRNnzCntDist*), whereas zero padding the *NNZ* reduces power efficiency in most cases. Although padding the NNZ simplifies the nonzeros count distribution, processing the zero values decreases power efficiency. *ParNnzCntDist* and *TableNnzCntDist* improve power efficiency by up to 12.4% and 21.7%, respectively.

Switching from sorting on columns to sorting on LSBs provides higher power efficiency. For the parallel subarrays implementations with four compute rows using *NSColSort*, switching to *NSLsbSort* or *BSort* improves throughput per watt by up to 10.2% or 20.7%, respectively.

For parallel subarrays with four compute rows with *BSort*, switching from *NPRNnzCntDist* to *ParNnzCntDist* or *TableNnzCntDist* increases throughput per watt by up to 9.79% or 5.66%, respectively.

The *SnakeSpMV* and *RowSpMV* methods are the two basic implementations first considered for SpMV. Additional implementations are added which increase throughput by increasing the number of parallel computation rows. The most advanced of these is *Parallel Subarrays w/ 8 Compute Rows w/ BSort w/ TableNnzCountDist (2 cores, 256 vals per core).* As N increases, the parallel subarrays implementations with eight computation rows gradually becomes more power efficient than the *SnakeSpMV* method and increases throughput per watt by up to 86.1%. The power efficiency of this parallel subarrays implementation also increases compared to the *RowSpMV* implementation, but never has a greater throughput per watt for the sparse matrices evaluated.

The implementations on the many-core platform are considered as alternatives to GPU and CPU implementations in order to improve power and area efficiency. The implementations on the many-core platform increase power efficiency by up to 14.0x versus the CPU implementations, and by up to 27.9x versus the GPU implementations.

The most power efficient implementation is *RowSpMV*; however, it is not the most area efficient. Therefore, one of the alternative implementations should be considered for achieving a large throughput per area value. Implementations with multiple computation rows generally become more power efficient as the size of the sparse matrix increases, and as the density increases. For each implementation, the most effective methods for improving power efficiency are to switch from *NPRNnzCntDist,* as the nonzeros count distribution method, to either *ParNnzCntDist* or *TableNnzCntDist.* For smaller matrix sizes, the simpler implementations such as *SnakeSpMV*, *RowS*pMV, and *Parallel Arrays w/ 1 Compute Row w/ NPRNnzCntDist* achieve the largest throughput per watt.

6.6.3 Sorting and Nonzeros Count Distribution Area Efficiency Comparisons

For parallel arrays with two computation rows, switching from *NPRNnzCntDist* improves area efficiency, with up to 14.4% improvement with *ParNnzCntDist*, 59.2% with *PadInputNnzCntDist*, and 12.2% with *TableNnzCntDist.*

For parallel arrays with four computation rows, all of the other nonzeros count distribution methods improve area efficiency versus *NPRNnzCntDist.* Throughput per area improves by up to

32.9% with *ParNnzCntDist*, 113% with *PadInputNnzCntDist*, and 71.3% with *TableNnzCntDist*.

When considering the parallel subarrays with four compute rows and *NSColSort*, switching to *NSLsbSort* or *BSort* consistently improves throughput per area, by up to 60.9% or 150%, respectively. For a majority of the matrices, switching from *NSColSort* to *NSLsbSort* and using *PadInputNnzCntDist* is effective for improving area efficiency, by as much as 35.9%.

For the parallel subarrays with four compute rows and *BSort*, generally only the *ParNnzCntDist* and *TableNnzCntDist* improve area efficiency (versus *NPRNnzCntDist*), by as much as 22.7% and 57.4%, respectively.

As *N* increases for the sparse matrices, implementations with more processing rows, such as the parallel subarrays method with eight computation rows, become more area efficient. Compared to the *SnakeSpMV* and *RowSpMV* methods, there is a positive correlation between *N* and the improvement in area efficiency. Across all five matrices, *Parallel Subarrays w/ 8 Compute Rows w/ BSort w/ TableNnzCountDist (2 cores, 256 vals per core)* is capable of improving area efficiency by up to 129% versus *SnakeSpMV* and by up to 132% versus *RowSpMV*.

The implementations on the many-core platform are effective at improving area efficiency versus using the methods on the GPUs or CPUs. The implementations on the many-core platform increase area efficiency by as much as 17.8x versus the CPU implementations, and up to 36.6x versus the GPU implementations.

Parallel Arrays w/ 1 Compute Row w/ NPRNnzCntDist is typically the most area efficient implementation, although as *N* increases, other implementations provide a higher throughput per watt. More complex implementations such as the parallel subarrays method using the *BSort* sorting method provide higher area efficiency, while also delivering competitive power efficiency, second to *RowSpMV*.

Table 6.3: Summary of Variables from Chapter 6.

Variable	Description
A	A sparse matrix of dimensions MxN.
x	A dense vector of dimensions Nx1.
b	A dense vector of dimensions Mx1.
N	The number of columns of A. For this dissertation, $N == M$.
NNZ	Number of nonzeros.
numStorableVals	Number of x values that can be stored on-chip.
numLoads	Number of times x values are loaded on the cores.
numCores	Number of cores available for a kernel implementation.
memPerCore	Memory available per core for storing x.
memShared	Size of on-chip memory, excluding each core's local memory.
memPerVal	Memory required to store each x value.
numCols	The number of column indexes of A for which a core handles multiplications.
colLow	Lowest column index of A for which values from x are stored for multiplication.
colHigh	Highest column index of A for which values from x are stored for multiplication.
numValsToPass	Number of x values to pass downstream before a core stores x data in memory.
*numColsInit**	Number of x values stored in a specific core on a regular load.
*numValsPerCoreRegLoad**	Lowest number of x values stored per core on a regular load.
*numValsRegLoad**	Total number of x values stored across all cores on a regular load.
*numValsAddRegLoad**	The additional number of x values that need to be stored that didn't evenly divide among the cores on a regular load.
*colLowInit**	For a regular load, the lower index of matrix A columns that will be multiplied by values of x in a core initially.
*colHighInit**	For a regular load, the upper index of matrix A columns that will be multiplied by values of x in a core initially.
*numValsToPassInit**	The number of x values to pass downstream before storing values from the x vector in a core initially.
*numColsLast**	Number of x values stored in a specific core on the last load.
*numValsPerCoreLastLoad**	The lowest number of x values stored per core on the last load.
*numValsAddLastLoad**	The additional number of x values that need to be stored that didn't evenly divide among the cores on the final load.
*numValsLastLoad**	The total number of x values stored across all cores for the last load.
*colLowLast**	For the last load, the lower index of matrix A columns that will be multiplied by values of x in a core.
*colHighLast**	For the last load, the upper index of matrix A columns that will be multiplied by values of x in a core.
*numValsToPassLast**	The number of x values to pass downstream before storing values from the x vector in a core for the last load.
numCoreRows	The size of the core array in the y dimension.
numCoreCols	The size of the core array in the x dimension.
numColsPerRow	Number of elements that each row of cores receives.
colLowRow	The *colLow* value for each row.

* Variable used only for case 2.

Table 6.4: Summary of Variables from Chapter 6 (continued).

Variable	Description
numValsToPassEast	Number of x values a core should pass east before storing x values in a local memory.
numValsToPassSouth	Number of x values a core should pass south before storing x values in a local memory.
*numColsPerRowInit**	Number of x values that will be stored in a processing row for a regular load.
*colLowRowInit**	The initial *colLow* value for each row that is incremented by *numValsRegLoad* after each load.
*numValsToPassEastInit**	Number of x values a core should pass east before storing x values in a local memory on a regular load.
*numValsToPassSouthInit**	Number of x values a core should pass south before storing x values in a local memory on a regular load
*numColsPerRowLast**	Number of x values that will be stored in a row for the last load.
*colLowRowLast**	Final *colLow* for each row.
*numValsToPassEastLast**	Number of x values a core should pass east before storing x values in a local memory on the last load.
*numValsToPassSouthLast**	Number of x values a core should pass south before storing x values in a local memory on the last load.
numDistributionStages	Number of stages in the distribution network.
numProcessingRows	Number of processing rows.
numCoresPerDistStage	Number of cores per distribution stage.
nnzPerDistCore	Number of nonzeros each *DistSendToken* core or core in the first column of the Butterfly network receives.
additionalNnzPerRow	When having to keep track of the last core to receive data, sometimes the *NNZ* for each row is not evenly divisible by the number of processing rows. The additional number of values that do not evenly divide is given by this variable.
nnzPerRow	Number of nonzeros for the current row.
numTableVals	Number of values to store for the table based nonzeros distributor.
baseNnzPerRow	Base number of nonzeros to send to each recipient for the table based nonzeros distributor.
tableRotation	The amount of rotation required for the table based nonzeros distributor.
tableAddress	The address for indexing into the table for the table based nonzeros distributor.
colHighRow	The row of the core in the sorting network for *North*South sorting.
numLSBsToSortOn	Number of LSBs to sort on.
LSBSortVal	Equal to the processing row. Used for sorting on LSBs.
numButterflyStages	Number of stages for the Butterfly network. Dependent on the number of processing rows.
xRowValLSBSort	The modified row index of the x vector value. Used with processing subarrays when sorting on LSBs.
xValStorageRow	The processing row that an x value will be stored in. Used with processing subarrays when sorting on LSBs.
colHighLSBSort	The upper column index to compare against for each row. Used with processing subarrays when sorting on LSBs.
colLowLSBSort	The lower column index to compare against for each row. Used with processing subarrays when sorting on LSBs.
numAccumulationStages	Number of stages in the accumulation network.
numCoresPerAccumStage	Number of cores per accumulation stage.
numVectorCopies	The number of copies of x that fit on-chip.
numDistributionStagesParallel	Number of stages for the distribution network for parallel arrays.
numCoresPerDistStageParallel	Number of cores per distribution stage for parallel arrays.
numAccumulationStagesParallel	Number of accumulation stages for parallel arrays.

* Variable used only for case 2.

Chapter 7

Summary and Future Work

A summary of the work covered in this dissertation and a list of future work is presented in this chapter.

7.1 Summary

In this dissertation, eight hybrid implementations with CFP hardware and six hybrid implementations with USL support were presented for a fixed-point processor. These implementations increased the throughput of FP operations by adding USL support instructions to the ISA, as well as some custom FP instructions. The area overhead was kept low by utilizing the existing fixed-point functional units.

The circuit area and throughput were found for 38 multiply-add, 8 addition/subtraction, 6 multiplication, 45 division, and 45 square root designs. This dissertation presented designs which both improved FP throughput versus a baseline software implementation. When compared to other works, some of the designs presented had a lower area impact (compared to an FMA). Several examples demonstrated how to easily determine the optimal FP designs given an area constraint. For example, hybrid implementations were shown to be an effective design method for increasing FP throughput and require up to 97.0% less area than a traditional FMA.

The second portion of this dissertation highlighted the effect that reducing the FP word width has on image quality and chip area when performing the backprojection algorithm to form airborne spotlight-mode SAR images from the X-band. The backprojection image formation algorithm was split into seven functional blocks and the effect of reducing precision and dynamic range was quantified using image quality and area comparisons. These reductions in width and area were considered a first step towards future SAR backprojection ASIC design and algorithm development. The image quality metrics of PSNR and SSIM were utilized to determine potential area savings while maintaining high image quality. The effect on final image quality when the FP word width was reduced for all blocks simultaneously was also demonstrated and showed no visible image quality degradation when using settings to obtain an SSIM of 0.9 or higher. Each functional block uses 48.4–91.2% less area than that required by DP-FP hardware.

This dissertation next presented the design process of two many-core chips in 32 nm PD-SOI CMOS. My contributions to the design flow are detailed, including my work on the physical design flow, synthesis, DRC, LVS, Ultrasim FastSpice simulations, static timing analysis, power planning, powergate SPICE simulations, RC extraction, adding the chip finishing structures, and determining the chip package specifications. The first chip features 1000 cores arranged in a 31 x 32 processor array with twelve 64 KB shared SRAM memories on the periphery of the chip. This chip operates at a nominal voltage of 0.9 V, with a core voltage range of 0.7–1.05 V. At 1.10 V, the processors' maximum clock frequency ranges from 1.70–1.87 GHz. Only 5.8 pJ per operation is required when operating at 0.56 V and 115 MHz. The second chip features 700 cores arranged in a 28 x 25 processor array, one of which is an FFT, and two which are Viterbi decoder accelerators. There are fourteen 64 KB shared SRAM memories on the chip and the nominal voltage is also 0.9 V with an operating range of 0.7–1.05 V. A custom package is designed for the second chip to add additional power delivery and I/O bandwidth. A DVFS mechanism is added to switch between three power rails for additional power savings. Both chips feature a GALS clocking method.

Finally, this dissertation explored implementing a scientific kernel on a many-core platform. Twenty-three functionally-equivalent SpMV designs were created for a many-core platform. *Snake-SpMV*, *RowSpMV*, Parallel Subarrays, and Parallel Arrays were evaluated for performing SpMV on the many-core platform. Using the metrics of throughput per watt and throughput per area, the area and power efficiency of these designs was measured against that of SpMV implementations on two GPUs and CPUs which used sparse matrix APIs for implementing SpMV. The performance, power, and area requirement of the designs were measured for performing SpMV on five unstructured sparse matrices from distinct scientific workloads of varying sizes. The many-core implementations increased power efficiency by up to 14.0x versus the CPU SpMV, and by up to 27.9x versus the GPU SpMV. Similarly, they provided up to 17.8x improvement in area efficiency versus the CPU SpMV, and up to 36.6x improvement versus the GPU SpMV.

This summary is followed by a list of the author's proposed future work.

7.2 Future Work

7.2.1 Synthetic Aperture Radar Imaging on a Fine-Grained Many-Core Array

The SAR image formation backprojection algorithm could be implemented on a many-core array. The author implemented the backprojection algorithm using software FP for AsAP2. Additionally, a C version of the backprojection algorithm was implemented by the author. A MATLAB implementation is also available [77]. One possibility to improve performance is to create a reduced FP word length software implementation of the backprojection algorithm and compare the throughput and energy costs versus using SP-FP. Another area that could be explored would be to implement the backprojection algorithm using fixed-point arithmetic and determine how the image quality and area compares to implementing the algorithm using FP hardware.

7.2.2 Synthetic Aperture Radar Image Processing Chip Design

The author proposes expanding the work presented in Chapter 4 into an ASIC for accelerating SAR backprojection. This work includes the physical layout of a full chip, as well as determining the throughput, area, and energy overhead compared to SP-FP and DP-FP implementations. This includes expanding the methods for image comparison by incorporating coherent change detection [128] when comparing formed images versus the gold standard image. This work could be expanded by adding a functional block for calculating the image grid, whereas the previous work assumed this data would be read from memory. Adding circuitry to perform image formation for the far field case could be considered, which requires the implementation of an alternate circuit for performing differential range calculations [100]. Additionally, circuitry for implementing the SAR matched filter algorithm could be designed for comparison.

7.2.3 Scientific Kernel on a Many-Core Platform

This dissertation explored implementing SpMV on a many-core platform in Chapter 6. However, there are many other common scientific kernels that could be implemented on a many-core platform to explore tradeoffs. Some of the kernels to explore include dense general matrix-matrix multiplication, stencil computations, and one dimensional and two dimensional FFTs [85]. Furthermore, exploration of case 2 for other SpMV kernels is left as a future research endeavor.

Glossary

AFRL Air Force Research Laboratory. The AFRL provided the SAR imaging data sets.

ALU Arithmetic Logic Unit. A unit of a computer that executes logical and arithmetic operations.

API Application Programming Interface. APIs are a set of programming routines, and tools for building software.

AsAP2 The second generation *Asyncronous Array of simple Processors* (AsAP) chip. AsAP2 is a fine-grained many-core system with 164 independently clocked homogeneous programmable processors.

ASIC Application-Specific Integrated Circuit. ASICs are integrated circuits that are designed for a specific application and are not capable of being reprogrammed.

BFP Block Floating-Point. With block floating-point, several values share the same exponent.

BGA Ball Grid Array. BGA is a type of surface-mount packaging where solder bumps connect a chip package to a printed circuit board.

BLAS Basic Linear Algebra Subroutines. A set of routines for performing common linear algebra operations.

C4 Controlled Collapse Chip Connection. C4 is a method for bonding semiconductors using solder bumps, for example, a chip die to a package.

CFP Custom Floating-Point. CFP instructions perform operations on data stored in FP registers. They increase FP throughput by reducing the bottlenecks of software kernels.

CMOS Complementary Metal Oxide Semiconductor. CMOS circuits are based on field-effect transistors and use both n-channel and p-channel transistors. Most modern chips use CMOS technology.

CT Computed Tomography. An imaging procedure used to reconstruct cross-sectional images by combining projection images of an object taken from different angles.

CUDA Compute Unified Device Architecture. An application programming interface (API) by NVIDIA to enable general purpose processing on a GPU.

DP Double-Precision. A 64-bit FP format composed of a sign bit, an 11 bit exponent, and 52 mantissa bits.

DRC Design Rule Check. This check determines if the layout of the physical chip satisfies a set of rules specified by the foundry.

DSP Digital Signal Processor. An integrated circuit designed for handling digital processing workloads.

DVFS Dynamic Voltage Frequency Scaling. A energy reduction technique to reduce a circuit's operating voltage and clock frequency depending on workload.

FFT Fast Fourier Transform. An efficient algorithm to calculate the discrete Fourier transform of a vector.

FIFO First-In First-Out. A type of buffering protocol where data are sent out in the same order in which they were received.

FLOP Floating-Point Operation.

FMA Fused Multiply-Add. An operation that performs the $a + b \times c$ and rounds only after the product has been added to the addend. FMA also refers to the unit capable of performing this operation.

FP Floating-Point. The most common method for approximating real numbers in modern computers. Real values are represented using a sign bit, a set of exponent bits, and significand bits.

FPGA Field-Programmable Gate Array. FPGAs are integrated circuits that are capable of being reprogrammed for a desired functionality. A hardware description language is typically used to program an array of logic blocks and reconfigurable interconnects.

FPU Floating-Point Unit. A unit that executes arithmetic operations on FP values. In this dissertation, a FPU is defined to be a unit capable of at least FP addition/subtraction and multiplication using software kernels, hardware modules, and/or hybrid implementations.

GALS Globally Asynchronous Locally Synchronous. GALS involves synchronous circuits communicating with each other asynchronously.

GDS Graphic Database System. GDS refers to a file format for storing the physical layout of an integrated circuit.

GMTI Ground Moving Target Indictor. A radar operation mode to distinguish moving objects from clutter.

IEEE-754 A techincal standard for FP arithmetic and data representation. The standard specifies of a set of formats, operations, rounding rules, flags, and the handling of exceptions.

IFFT Inverse Fast Fourier Transform. An efficient algorithm to calculate the inverse discrete Fourier transform of a vector.

Intel MKL Intel Math Kernel Library. A library of math processing routines.

ISA Instruction Set Architecture. The instruction set and programmer visible aspects of the CPU.

LSB Least Significant Bit. In this dissertation, LSB refers to the right-most bit of a binary value.

LVDS Low-voltage Differential Signaling. A communication protocol that enables low power fast data transmission.

LVS Layout Versus Schematic. This check verifies that the physical chip layout matches the gate level netlist.

LZA Leading Zeros Anticipator. A circuit to anticipate the left-shift required for normalization.

MAC Multiply-Accumulate. The multiplication of two values followed by adding the product to an accumulated sum.

MIMD Multiple Instruction, Multiple Data. MIMD is a type of parallel architecture where multiple processing elements execute independent instructions on different data.

MSB Most Significant Bit. In this dissertation, MSB refers to the left-most bit of a binary value.

NaN Not a Number. NaN is a symbol for representing the result of invalid FP operations.

PD-SOI Partially Depleted Silicon On Insulator. A fabrication technology where partially depleted layered silicon is placed on an insulator, which is then placed on a silicon substrate. SOI lowers parasitic capacitance by isolating the silicon junction using an insulator.

Processor In this dissertation, processor or core refers to an integrated circuit capable of independent program execution.

PSNR Peak Signal-to-Noise Ratio. PSNR measures the ratio between the maximum signal power and the noise corrupting the image.

RISC Reduced Instruction Set Computing. A CPU design strategy focused around a simple ISA.

RTL Register-Transfer Level. An abstraction used in a hardware description language such as Verilog. RTL expresses circuit behavior in terms of registers and combinational logic.

SAR Synthetic Aperture Radar. A radar imaging method where pulses of microwave energy are transmitted and received from a series of locations. SAR provides a means for day, night, and all weather imaging while producing resolution that otherwise requires a large antenna aperature.

SIMD Single Instruction, Multiple Data. SIMD is a type of parallel architecture where multiple processing elements execute the same operation on different data.

SP Single-Precision. A 32-bit FP format composed of a sign bit, an 8 bit exponent, and 23 mantissa bits.

SpMV Sparse Matrix-Vector Multiplication. A common scientific kernel involving the multiplication of a sparse matrix with a dense vector.

SSIM Structural Similarity. The SSIM index is a metric used to quantify the quality of a reconstructed image and is based on the notion that human visual perception is adapted for extracting structural information about an image.

USL Unsigned, Shift-Carry, and Leading Zero Detection. USL support is added to a processor to enable unsigned operations, shift operations with the ability to set a carry flag if data is shifted out, and the ability to count leading zeros.

VLIW Very Long Instruction Word. A computer architecture which allows a program to specify multiple instructions to execute in parallel.

X-band A frequency range from 8–12 GHz often used for radar imaging.

Bibliography

[1] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*, 1st ed. Birkhäuser Basel, 2009.

[2] J. Munson, D.C., J. O'Brien, and W. Jenkins, "A tomographic formulation of spotlight-mode synthetic aperture radar," *Proc. IEEE*, vol. 71, no. 8, pp. 917–925, Aug 1983.

[3] W. Carrara, R. Goodman, and R. Majewski, *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*, ser. Artech House remote sensing library. Artech House, 1995.

[4] C. Jakowatz Jr., D. Wahl, P. Eichel, D. Ghiglia, and P. Thompson, *Spotlight-mode Synthetic Aperture Radar: A signal processing approach.* Boston, MA: Kluwer Academic Publishers, 1996.

[5] J. Park, P. Tang, M. Smelyanskiy, D. Kim, and T. Benson, "Efficient backprojection-based synthetic aperture radar computation with many-core processors," in *Proc. of Intl. Conf. for High Perf. Comp., Net., Storage and Analysis (SC)*, Nov. 2012, pp. 1–11.

[6] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "KiloCore: A 32 nm 1000-processor array,," in *IEEE HotChips Symp. on High-Perf. Chips (HotChips 2016)*, Jun. 2016.

[7] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia, and B. Baas, "A 167-processor computational platform in 65 nm CMOS," *IEEE J. Solid-State Circuits*, 2009.

[8] A. Stillmaker, "Design of energy-efficient many-core MIMD GALS processor arrays in the 1000-processor era," Ph.D. dissertation, University of California, Davis, Davis, CA, USA, Dec. 2015, http://vcl.ece.ucdavis.edu/pubs/theses/2015-1/.

[9] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "A 5.8 pJ/Op 115 billion Ops/sec, to 1.78 trillion Ops/sec 32 nm 1000-processor array," in *VLSI Circuits, IEEE Symp. on*, Jun. 2016.

[10] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M. Baas, "KiloCore: A 32-nm 1000-processor computational array," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 891–902, April 2017.

[11] S. Gilani, N. S. Kim, and M. Schulte, "Energy-efficient floating-point arithmetic for software-defined radio architectures," in *Application-Specific Syst., Architectures and Processors (ASAP), 2011 IEEE Int. Conf. on*, Sep. 2011, pp. 122–129.

[12] S. S. Hasan, "An investigation of the Blackfin/uClinux combination as a candidate software radio processor," Virginia Polytechnic Institute & State University, Tech. Rep., 2006.

[13] "Floating point arithmetic on the picoArray," picoChip, Nov. 2013, Application Note.

[14] C. Iordache and P. Tang, "An overview of floating-point support and math library on the Intel XScale architecture," in *Comput. Arithmetic, 2003. Proc. 16th IEEE Symp. on*, Jun. 2003, pp. 122–128.

[15] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, D. Truong, T. Mohsenin, and B. Baas, "AsAP: An asynchronous array of simple processors," *IEEE J. Solid-State Circuits*, vol. 43, no. 3, pp. 695–705, Feb. 2008.

[16] Y. J. Chong and S. Parameswaran, "Configurable multimode embedded floating-point units for FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst*, vol. 19, no. 11, pp. 2033–2044, Nov. 2011.

[17] The industry's first floating-point FPGA. Altera. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/po/bg-floating-point-fpga.pdf

[18] J. J. Pimentel and B. M. Baas, "Software/hardware hybrid floating-point units with low area overhead on a fine-grained processing core," in *Technology and Talent for the 21st Century (TECHCON 2014)*, Sep. 2014.

[19] J. J. Pimentel and B. M. Baas, "Hybrid floating-point modules with low area overhead on a fine-grained processing core," in *IEEE Asilomar Conf. on Signals, Syst. and Comput. (ACSSC)*, Nov. 2014, pp. 1829–1833.

[20] J. J. Pimentel, B. Bohnenstiehl, and B. M. Baas, "Hybrid hardware/software floating-point implementations for optimized area and throughput tradeoffs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst*, vol. PP, no. 99, pp. 1–14, 2016.

[21] W. Commons. (1974) Venus in real colors, processed from clear and blue filtered Mariner 10 images. File: `Venus-real_color.jpg`. [Online]. Available: https://commons.wikimedia.org/wiki/File:Venus-real_color.jpg

[22] NASA. (1991) PIA00104: Venus - computer simulated global view centered at 180 degrees east longitude. File: `PIA00104_modest.jpg`. [Online]. Available: http://photojournal.jpl.nasa.gov/catalog/PIA00104

[23] S. Borkar, "Thousand core chips: A technology perspective," in *Proc. of the 44th Annual Design Automation Conf.*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 746–749.

[24] M. Butts, "Synchronization through communication in a massively parallel processor array," *IEEE Micro*, vol. 27, no. 5, pp. 32–40, Sep. 2007.

[25] P. Colella, "Defining software requirements for scientific computing," 2004, Presentation.

[26] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," University of California, Berkeley, Tech. Rep., Dec. 2006.

[27] L. Oliker, G. Gorden, P. Husbands, and J. Chame, “Evaluation of architectural paradigms for addressing the processor-memory gap,” Jul. 2003, Lawrence Berkeley National Laboratory.

[28] S. Boldo and J.-M. Muller, “Exact and approximated error of the FMA,” *IEEE Trans. Comput.*, vol. 60, no. 2, pp. 157–164, 2011.

[29] S. Galal and M. Horowitz, “Energy-efficient floating-point unit design,” *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 913–922, Jul. 2011.

[30] M. Beauchamp, S. Hauck, K. Underwood, and K. Hemmert, “Architectural modifications to enhance the floating-point performance of FPGAs,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst*, vol. 16, no. 2, pp. 177–187, Feb. 2008.

[31] K. Kalliojarvi and J. Astola, “Roundoff errors in block-floating-point systems,” *IEEE Trans. Signal Process.*, vol. 44, no. 4, pp. 783–790, Apr 1996.

[32] S. Gilani, N. S. Kim, and M. Schulte, “Virtual floating-point units for low-power embedded processors,” in *Application-Specific Syst., Architectures and Processors (ASAP), 2012 IEEE Intl. Conf. on*, 2012, pp. 61–68.

[33] T. Viitanen, P. Jaaskelainen, and J. Takala, “Inexpensive correctly rounded floating-point division and square root with input scaling,” in *Signal Process. Syst. (SiPS), 2013 IEEE Workshop on*, Oct. 2013, pp. 159–164.

[34] F. Fang, T. Chen, and R. A. Rutenbar, “Lightweight floating-point arithmetic: case study of inverse discrete cosine transform,” *EURASIP J. on Appl. Signal Process.*, vol. 2002, no. 1, pp. 879–892, Jan. 2002.

[35] S.-W. Lee and I.-C. Park, “Low cost floating-point unit design for audio applications,” in *Circuits and Syst., 2002. ISCAS 2002. IEEE Intl. Symp. on*, vol. 1, 2002, pp. I–869–I–872 vol.1.

[36] J. Pimentel, A. Stillmaker, B. Bohnenstiehl, and B. Baas, “Area efficient backprojection computation with reduced floating-point word width for SAR image formation,” in *Signals, Syst. and Comput., 2015 49th Asilomar Conf. on*, Nov. 2015.

[37] J. Tong, D. Nagle, and R. Rutenbar, “Reducing power by optimizing the necessary precision/range of floating-point arithmetic,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst*, vol. 8, no. 3, pp. 273–286, 2000.

[38] P. Gysel, J. J. Pimentel, M. Motamedi, and S. Ghiasi, “On resource-efficient inference using trained convolutional neural networks,” *IEEE Trans. Neural Networks and Learning Systems*, 2017, In Review.

[39] H.-J. Oh, S. Mueller, C. Jacobi, K. Tran, S. Cottier, B. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. H.Dhong, “A fully pipelined single-precision floating-point unit in the synergistic processor element of a CELL processor,” *IEEE J. Solid-State Circuits*, vol. 41, no. 4, pp. 759–771, 2006.

[40] N. Hockert and K. Compton, “Improving floating-point performance in less area: Fractured Floating Point Units (FFPUs),” *J. of Signal Process. Syst.*, vol. 67, no. 1, pp. 31–46, Apr. 2012.

[41] *IEEE Standard for Floating-Point Arithmetic*, 2008, IEEE Std 754-2008.

[42] D. Lutz and C. Hinds, “Novel rounding techniques on the neon floating-point pipeline,” in *IEEE Asilomar Conf. on Signals, Syst. and Comput. (ACSSC)*, Oct. 2005, pp. 1342–1346.

[43] A. H. Karp and P. Markstein, “High-precision division and square root,” *ACM Trans. Math. Softw.*, vol. 23, no. 4, pp. 561–589, Dec. 1997.

[44] M.-B. Lin, *Digital System Designs and Practices: Using Verilog HDL and FPGAs.* Wiley Publishing, 2008.

[45] S. Oberman and M. Flynn, “Design issues in division and other floating-point operations,” *IEEE Trans. Comput.*, vol. 46, no. 2, pp. 154–161, Feb. 1997.

[46] Z. Jin, R. N. Pittman, and A. Forin, “Reconfigurable custom floating-point instructions,” Tech. Rep. MSR-TR-2009-157, Aug. 2009.

[47] S. Oberman and M. Flynn, “Division algorithms and implementations,” *IEEE Trans. Comput.*, vol. 46, no. 8, pp. 833–854, Aug 1997.

[48] “CRAY T3E Fortran optimization guide,” Jun. 1997.

[49] H. Nikmehr, “Architectures for floating-point division,” Ph.D. dissertation, The University of Adelaide, 2005.

[50] N. Takagi, S. Kadowaki, and K. Takagi, “A hardware algorithm for integer division,” in *Comput. Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symp. on*, Jun. 2005, pp. 140–146.

[51] M. Schulte, J. Omar, and J. Swartzlander, E.E., “Optimal initial approximations for the Newton-Raphson division algorithm,” *Computing*, vol. 53, no. 3-4, pp. 233–242, 1994.

[52] Y. Li and W. Chu, “Implementation of single precision floating point square root on FPGAs,” in *Proc. IEEE FCCM*, Apr 1997, pp. 226–232.

[53] P. Soderquist and M. Leeser, “Division and square root: choosing the right implementation,” *IEEE Micro*, vol. 17, no. 4, pp. 56–66, Jul 1997.

[54] P. Montuschi and M. Mezzalama, “Optimal absolute error starting values for Newton-Raphson calculation of square root,” *Computing*, vol. 46, no. 1, pp. 67–86, 1991.

[55] P. Markstein, *IA-64 and elementary functions: speed and precision.* Prentice-Hall, 2000.

[56] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. M. Baas, “An asynchronous array of simple processors for DSP applications,” in *IEEE Intl. Solid-State Circuits Conf., (ISSCC ’06)*, Feb. 2006, pp. 428–429.

[57] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the clouds: A study of emerging scale-out workloads on modern hardware,” in *Proc. of the Seventeenth Intl. Conf. on Architectural Support for Programming Languages and Operating Syst.*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 37–48.

[58] M. Horowitz, “1.1 computing’s energy problem (and what we can do about it),” in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 10–14.

[59] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, A. Tran, J. Webb, E. Work, Z. Xiao, and B. Baas, "A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling," in *VLSI Circuits, IEEE Symp. on*, Jun. 2008.

[60] Z. Xiao and B. M. Baas, "A 1080p H.264/AVC baseline residual encoder for a fine-grained many-core system," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 21, no. 7, pp. 890–902, Jul. 2011.

[61] Z. Xiao and B. Baas, "A high-performance parallel CAVLC encoder on a fine-grained many-core system," in *Comput. Design, 2008. ICCD 2008. IEEE Int. Conf. on*, Oct. 2008, pp. 248–254.

[62] A. Tran, D. Truong, and B. Baas, "A complete real-time 802.11a baseband receiver implemented on an array of programmable processors," in *IEEE Asilomar Conf. on Signals, Syst. and Comput. (ACSSC)*, Oct. 2008, pp. 165–170.

[63] B. Liu and B. M. Baas, "Parallel AES encryption engines for many-core processor arrays," *IEEE Trans. Comput.*, vol. 62, no. 3, pp. 536–547, Mar. 2013.

[64] B. Liu, "Energy-efficient computing with fine-grained many-core systems," Ph.D. dissertation, University of California, Davis, Davis, CA, USA, Sep. 2016, http://vcl.ece.ucdavis.edu/pubs/theses/2016-1/.

[65] D. Truong and B. Baas, "Massively parallel processor array for mid-/back-end ultrasound signal processing," in *Biomedical Circuits and Syst. Conf. (BioCAS), 2010 IEEE*, Nov. 2010, pp. 274–277.

[66] A. Yegulalp, "Fast backprojection algorithm for synthetic aperture radar," in *Proc. IEEE RadarCon*, 1999, pp. 60–65.

[67] M. Desai and W. Jenkins, "Convolution backprojection image reconstruction for spotlight mode synthetic aperture radar," *IEEE TIP*, vol. 1, no. 4, pp. 505–517, Oct. 1992.

[68] R. Portillo, S. Arunagiri, P. J. Teller, S. J. Park, L. H. Nguyen, J. C. Deroba, and D. Shires, "Power versus performance tradeoffs of GPU-accelerated backprojection-based synthetic aperture radar image formation," in *Proc. SPIE*, vol. 8060, Jun. 2011, pp. 806 008–806 008–21.

[69] D. Wang and M. Ali, "Synthetic aperture radar on low power multi-core digital signal processor," in *Proc. IEEE HPEC*, Sep. 2012, pp. 1–6.

[70] K.-N. Chia, H. J. Kim, S. Lansing, W. Mangione-Smith, and J. Villasensor, "High-performance automatic target recognition through data-specific VLSI," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst*, vol. 6, no. 3, Sep. 1998.

[71] N. Agrawal and K. Venugopalan, "Analysis of complex SAR raw data compression," in *Proc. PIERS*, Cambridge, MA, Jul. 2008, pp. 155–160.

[72] S. Bhattacharya, T. Blumensath, B. Mulgrew, and M. Davies, "Synthetic aperture radar raw data encoding using compressed sensing," in *Radar Conf., 2008. RADAR '08. IEEE*, May 2008, pp. 1–5.

[73] A. Capozzoli, C. Curcio, and A. Liseno, "Fast GPU-based interpolation for SAR backprojection," *Progress In Electromagnetics Research*, vol. 133, pp. 259–283, 2013.

[74] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst*, vol. 8, no. 3, pp. 273–285, Jun. 2000.

[75] C. H. Casteel, Jr., L. A. Gorham, M. J. Minardi, S. M. Scarborough, K. D. Naidu, and U. K. Majumder, "A challenge problem for 2D/3D imaging of targets from a volumetric data set in an urban environment," in *Proc. SPIE*, vol. 6568, 2007, pp. 65 680D–65 680D–7.

[76] S. M. Scarborough, C. H. Casteel, Jr., L. Gorham, M. J. Minardi, U. K. Majumder, M. G. Judge, E. Zelnio, M. Bryant, H. Nichols, and D. Page, "A challenge problem for SAR-based GMTI in urban environments," in *Proc. SPIE*, vol. 7337, 2009.

[77] L. A. Gorham and L. J. Moore, "SAR image formation toolbox for MATLAB," in *Proc. of SPIE*, vol. 7699, no. 1. SPIE, 2010, p. 769906.

[78] A. Lugowski, D. Alber, A. Buluç, J. R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis, "A flexible open-source toolbox for scalable complex graph analysis," in *Proc. of the 2012 SIAM Intl. Conf. on Data Mining*, 2012, pp. 930–941.

[79] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system implementation and observations," in *Proc. of the 2009 Ninth IEEE Intl. Conf. on Data Mining*, ser. ICDM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 229–238.

[80] S. Rousseaux, D. Hubaux, P. Guisset, and J.-D. Legat, "A high performance FPGA-based accelerator for BLAS library implementation," in *Proceedings of the 3rd Annual Reconfigurable Systems Summer Institute (RSSI 2007)*, 2007.

[81] R. Dorrance and D. Markovic, "A 190GFLOPS/W DSP for energy-efficient sparse-BLAS in embedded IoT," in *2016 IEEE Symp. on VLSI Circuits (VLSI-Circuits)*, June 2016, pp. 1–2.

[82] H.-V. Dang and B. Schmidt, "Cuda-enabled sparse matrix-vector multiplication on gpus using atomic operations," *Parallel Comput.*, vol. 39, no. 11, pp. 737–750, Nov. 2013.

[83] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units," *SIAM J. on Scientific Comput.*, vol. 36, no. 5, pp. C401–C423, 2014.

[84] W. Liu and B. Vinter, "CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proc. 29th ACM on Intl. Conf. on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 339–350.

[85] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "Scientific computing kernels on the cell processor," *Intl. J. of Parallel Programming*, vol. 35, no. 3, pp. 263–298, 2007.

[86] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[87] T. A. Davis. (2012) Cxsparse. [Online]. Available: https://github.com/PetterS/CXSparse

[88] G. Guennebaud, B. Jacob, et al. (2016) Eigen: A C++ linear algebra library. V3. [Online]. Available: http://github.com/kylelutz/compute

[89] NVIDIA, "cuSPARSE library," NVIDIA, User Guide, September 2016, DU-06709-001_v8.0.

[90] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[91] S. O. Yesylevskyy, "Pteros 2.0: Evolution of the fast parallel molecular analysis library for c++ and python," *Journal of Computational Chemistry*, vol. 36, no. 19, pp. 1480–1488, 2015.

[92] (2017, January) cuSPARSE. [Online]. Available: https://developer.nvidia.com/cusparse

[93] R. Montoye, E. Hokenek, and S. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," *IBM J. of Research and Development*, vol. 34, no. 1, pp. 59–70, Jan. 1990.

[94] E. Hokenek and R. Montoye, "Leading-zero anticipator (LZA) in the IBM RISC System/6000 floating-point execution unit," *IBM J. of Research and Development*, vol. 34, no. 1, pp. 71–77, Jan. 1990.

[95] V. G. Oklobdzija and R. K. Krishnamurthy, Eds., *High-Performance Energy-Efficient Microprocessor Design (Integrated Circuits and Systems)*, 2006th ed. Springer, Aug. 2006.

[96] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel, "FPgen - a test generation framework for datapath floating-point verification," in *High-Level Design Validation and Test Workshop, 2003. Eighth IEEE Int.*, 2003, pp. 17–22.

[97] S. F. Oberman and M. J. Flynn, "An analysis of division algorithms and implementations," Stanford University, Tech. Rep., 1995.

[98] C. Wolff. (2017) Radar basics - synthetic aperture radar. [Online]. Available: http://www.radartutorial.eu/20.airborne/ab07.en.html

[99] A. W. Doerry and F. M. Dickey, "Synthetic aperture radar," *Opt. Photon. News*, vol. 15, no. 11, pp. 28–33, Nov 2004.

[100] B. Rigling and R. Moses, "Taylor expansion of the differential range for monostatic SAR," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 41, no. 1, pp. 60–64, Jan. 2005.

[101] M. Richards, "A beginner's guide to interferometric SAR concepts and signal processing," *IEEE Aerosp. Electron. Syst. Mag.*, vol. 22, no. 9, pp. 5–29, Sep. 2007.

[102] A. V. Oppenheim and R. W. Schafer, *Discrete-time Signal Processing*, 3rd ed. Prentice Hall, 2010, pp. 730–731.

[103] K. C. Ng, "Argument reduction for huge arguments: Good to the last bit," Tech. Rep., 1992.

[104] J. F. Hart, *Computer Approximations.* Krieger Publishing Co., 1978.

[105] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Trans. Image Process.*, vol. 13, no. 4, pp. 600–612, Apr. 2004.

[106] W. J. Townsend, E. E. Swartzlander, Jr., and J. A. Abraham, “A comparison of Dadda and Wallace multiplier delays,” in *Advanced Signal Process. Algorithms, Architectures, and Implementations XIII*, F. T. Luk, Ed., vol. 5205, Dec. 2003, pp. 552–560.

[107] “DW01_add,” Synopsys, Inc., Mar. 2007, DesignWare Building Block IP.

[108] “DW02_mult,” Synopsys, Inc., Sep. 2008, DesignWare Building Block IP.

[109] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M. Baas, “KiloCore: A fine-grained 1000 processor array for task parallel applications,” *IEEE Micro*, 2017.

[110] D. Bol, J. D. Vos, C. Hocquet, F. Botman, F. Durvaux, S. Boyd, D. Flandre, and J. D. Legat, “Sleepwalker: A 25-MHz 0.4 V sub-mm^2 μw/MHz microcontroller in 65-nm LP/GP CMOS for low-carbon wireless sensor networks,” *IEEE J. Solid-State Circuits*, vol. 48, no. 1, pp. 20–32, Jan. 2013.

[111] D. C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. M. Harvey, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa, “Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor,” *IEEE J. Solid-State Circuits*, vol. 41, no. 1, pp. 179–196, Jan. 2006.

[112] EZChip Semiconductor, Inc., “TILE-Gx72 processor product brief,” Feb. 2015, Product Brief.

[113] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, “An 80-tile 1.28 TFLOPS network-on-chip in 65nm CMOS,” in *IEEE Intl. Solid-State Circuits Conf., (ISSCC ’07)*, Feb. 2007, pp. 98–589.

[114] D. Foley, P. Bansal, D. Cherepacha, R. Wasmuth, A. Gunasekar, S. Gutta, and A. Naini, “A low-power integrated x86-64 and graphics processor for mobile computing devices,” *IEEE J. Solid-State Circuits*, vol. 47, no. 1, pp. 220–231, Jan 2012.

[115] J. Warnock, Y. H. Chan, S. Carey, H. Wen, P. Meaney, G. Gerwig, H. H. Smith, Y. Chan, J. Davis, P. Bunce, A. Pelella, D. Rodko, P. Patel, T. Strach, D. Malone, F. Malgioglio, J. Neves, D. L. Rude, and W. Huott, “Circuit and physical design implementation of the microprocessor chip for the zenterprise system,” *IEEE J. Solid-State Circuits*, vol. 47, no. 1, pp. 151–163, Jan 2012.

[116] S. Huck, “Measuring processor power,” Apr. 2011, White Paper. [Online]. Available: http://www.intel.com/content/dam/doc/white-paper/resources-xeon-measuring-processor-power-paper.pdf

[117] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, “Understanding the performance of sparse matrix-vector multiplication,” in *16th Euromicro Conf. on Parallel, Distributed and Network-Based Processing (PDP 2008)*, Feb 2008, pp. 283–292.

[118] S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds., *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, 1st ed. Springer International Publishing, 2015, vol. 8966, pp. 178.

[119] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. of the Conf. on High Perf. Comput. Networking, Storage and Analysis*, Nov 2009, pp. 1–11.

[120] W. M. Holt, "1.1 moore's law: A path going forward," in *IEEE Intl. Solid-State Circuits Conf., (ISSCC '16)*, Jan 2016, pp. 8–13.

[121] CPUID. (2017) HWMonitor PRO. [Online]. Available: http://www.cpuid.com/softwares/hwmonitor-pro.html

[122] B. Heaney, "DAC 2012 Keynote: Designing a 22nm Intel architecture multi-CPU and GPU," DAC Design Automation Conference 2012, Jun. 2012, Keynote Address.

[123] "2nd generation Intel Core processor family desktop and Intel Pentium processor family desktop, and LGA1155 socket," May 2011, Thermal Mechanical Specifications and Design Guidelines (TMSDG).

[124] M. Butler, "AMD Bulldozer Core - a new approach to multithreaded compute performance for maximum efficiency and throughput." in *IEEE HotChips Symp. on High-Perf. Chips (HotChips 2010)*, Aug. 2010.

[125] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Elsevier, 2012, p. 22.

[126] (2017) NVIDIA GeForce GT 620 OEM. [Online]. Available: https://www.techpowerup.com/gpudb/358/geforce-gt-620-oem

[127] (2017) NVIDIA NVS 4200M. [Online]. Available: https://www.techpowerup.com/gpudb/1741/nvs-4200m

[128] P. Wright, T. Macklin, C. Willis, and T. Rye, "Coherent change detection with SAR," in *Radar Conf., 2005. EURAD 2005. European*, Oct. 2005, pp. 17–20.