Compute Efficient Embedded Processors

By

Syed Zohaib Gilani

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN-MADISON

2012

Date of final oral examination: 11/29/2012

The dissertation is approved by the following members of the Final Oral Committee:
Katherine Morrow, Associate Professor, Electrical and Computer Engineering
Karthikeyan Sankaralingam, Assistant Professor, Computer Sciences
Michael Schulte, Fellow, AMD Research
Mikko Lipasti, Philip Dunham Reed Professor, Electrical and Computer Engineering
Nam Sung Kim, Assistant Professor, Electrical and Computer Engineering

ACKNOWLEDGEMENTS

There is a long list of people who have contributed directly and indirectly to this dissertation. Without these people, I could not have succeeded at the University of Wisconsin. First of all, I would like to thank my parents, Lt. Col. Masood Gilani (late) and Dr. Ghausia Gilani, and my brother, Dr. Aamir Gilani, for their unflinching support throughout my life.

I want to express my deepest gratitude to my dissertation advisors, Professor Nam Sung Kim and Dr. Michael Schulte, for their guidance and support during my graduate studies. Their knowledge, work ethic and research insight has been inspirational. Throughout my dissertation I have continuously strived to emulate their qualities. I look forward to more professional collaboration with them.

During my internship at Advanced Micro Devices, I was fortunate to have Dr. Srilatha Manne as my mentor. I found her to be extremely knowledgeable and understanding. My experience of working with her greatly helped develop my research aptitude. I am highly grateful for her support both during and after the internship.

I would also like to thank my dissertation committee members, Professor Mikko Lipasti, Professor Katherine Morrow and Professor Karu Sankaralingam for their feedback that helped improve this dissertation.

I owe thanks to Mitch Hayenga, Atif Hashmi, Andy Nere, Arslan Zulfiqar, Tony Gregerson, Shreesha Srinath, Daniel Chang, Vignyan Reddy, Dave Palframan, Paula Aguilera, Amin Farmahini, Jungseob Lee and Hamid Ghasemi for their support and company in the lab. My graduate studies were funded by the US government through the Fulbright program. I am grateful to the US government for providing me with the opportunity to perform research at a premier re-

search university.

ABSTRACT

Emerging embedded computing applications are becoming increasingly compute intensive and require high performance processors. However, embedded processors are typically battery-powered with limited power and energy budgets. Traditional approaches to improve the performance of general purpose processors, such as increasing resources and/or the frequency of the processor can significantly increase their energy and power consumption. These approaches are often not feasible for low-power embedded processors.

The power budget of low-power embedded processors is typically limited due to their small form-factor and stringent thermal constraints. Moreover, many modern embedded computing applications are floating-point (FP) intensive. Providing efficient support for FP arithmetic is thus critical for future embedded processors, yet current-generation embedded processors do not efficiently support FP arithmetic.

This dissertation focuses on improving the power efficiency of modern embedded processors. For FP-intensive applications, this dissertation proposes a novel FP fused multiply-add (FMA) design that improves performance, accuracy and power efficiency. The proposed FMA unit compresses long dependence chains of FP operations, allowing a large percentage of dependent FP operations to be issued in consecutive cycles despite the long latency of FP operations.

For low-power embedded processors, this dissertation also proposes virtual FP units that can provide architectural support for FP operations in a fixed-point (FxP) processor. The proposed approach allows FP operations to be executed in FxP processors with extremely low area and power overhead. The proposed virtual FPUs can also increase the performance of many applications compared to dedicated hardware FPUs by reducing the dependence chains in FP operations.

Modern embedded architectures integrate graphics processing units (GPUs) with embedded processors [1, 2]. These integrated GPUs can be used to accelerate general-purpose applications [3, 4]. This dissertation proposes approaches to improve the performance and power-efficiency of GPUs.

GPUs typically share hardware resources between thousands of active threads. Consequently, they employ several collision detection and avoidance stages in their pipelines to allow efficient sharing of hardware resources. However, the addition of these stages increases their pipeline depth, and hence the read-after-write (RAW) latency of instructions. Moreover, due to their high power overhead, most embedded GPUs do not support traditional data-forwarding networks (DFNs).

This dissertation proposes a novel compiler-directed data-forwarding approach that can significantly improve the performance of general-purpose GPU (GPGPU) applications without the high power overhead of traditional DFNs. The proposed approach is also used to reduce the power consumption of GPUs by lowering the voltage of execution units without increasing the RAW time of a large percentage of instructions. This allows a significant reduction in the GPU power consumption with negligible performance impact.

This dissertation also proposes to improve the performance of integer applications by efficiently utilizing the FP execution units in GPUs. This allows considerable energy and performance improvements for GPGPU applications. Further improvements in performance and power efficiency are achieved by exploiting computational redundancy within a set of co-issued threads in GPUs. This computational redundancy exists whenever the operand values for all co-issued threads are identical and thus produce the same result.

Finally, to efficiently utilize the register file and execution bandwidth in GPUs, this dissertation proposes a sliced GPU architecture that considerably increases instruction throughput for instructions whose operands only require 16 or fewer bits for accurate representation.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

This chapter describes the motivation for improving the compute efficiency of embedded processors, provides an overview of the research described in the dissertation, and describes the dissertation's main contributions. Section 1.1 delineates the power consumption, compute performance, programmability and energy efficiency challenges faced by low-power embedded processors and the compute requirements of modern embedded applications. It also describes sources of energy inefficiency in embedded processors and motivates research into increasing their compute efficiency. Section 1.2 gives an overview of research included in this dissertation, and Section 1.3 describes its contribution. Section 1.4 outlines the remaining chapters of the dissertation.

1.1 MOTIVATION

With the emergence of pervasive computing, many battery-powered embedded computing devices need to provide support for highly compute-intensive applications, such as face recognition, physical layer processing of wireless communication standards and high definition video encoding/decoding. The compute capability of general purpose processors can be increased by increasing their issue widths, instruction window sizes, operating frequencies and the number of compute resources. However, each of these approaches significantly increases the power consumption of processors. On the other hand, the power budget of low-power embedded processors is typically limited due to their stringent form-factor and thermal constraints. Thus, conventional approaches to increasing performance of processors often cannot be applied directly to embedded processors.

Many workloads encountered in embedded computing are from the domains of image process-

ing, computer vision and wireless communications [5]. These applications typically involve an abundance of matrix-based operations, such as matrix multiplications, inversions and decompositions. The computational complexity of most of these algorithms increases as $O(N^3)$, where N is the matrix dimension. Hence, the complexity of these applications is likely to increase super-linearly as the wireless communication bandwidths, matrix dimensions, and image resolutions are increased. Moreover, these compute intensive applications often have hard real-time execution deadlines. These execution deadlines coupled with the limited power budgets and increasing computational requirements of applications necessitate that future embedded processors improve their compute performance significantly with little or no increase in power consumption.

Another important consideration for low-power embedded processors is energy consumption. Since the battery life of these devices is also limited, future embedded processors must provide performance improvements while also increasing the energy efficiency of the architecture to prolong battery life. Modern embedded devices are designed as system-on-chip (SoC) architectures that integrate dedicated accelerators with embedded processors to improve the performance of compute-intensive applications (e.g. face recognition and computational photography [6]). However, enhancing on-chip embedded processors to support such applications can provide more flexibility and allow programmers to develop platform-independent applications.

Many emerging compute-intensive embedded computing applications are FP intensive. In particular, applications in the domains of wireless communications, computer vision, and media processing perform extensive FP arithmetic operations. Providing efficient support for FP arithmetic is thus critical for future embedded processors. Many current generation low-power processors (e.g. Texas Instruments TMS320C64x [7], Sandbridge Sandblaster DSP [8], ARM Ardbeg

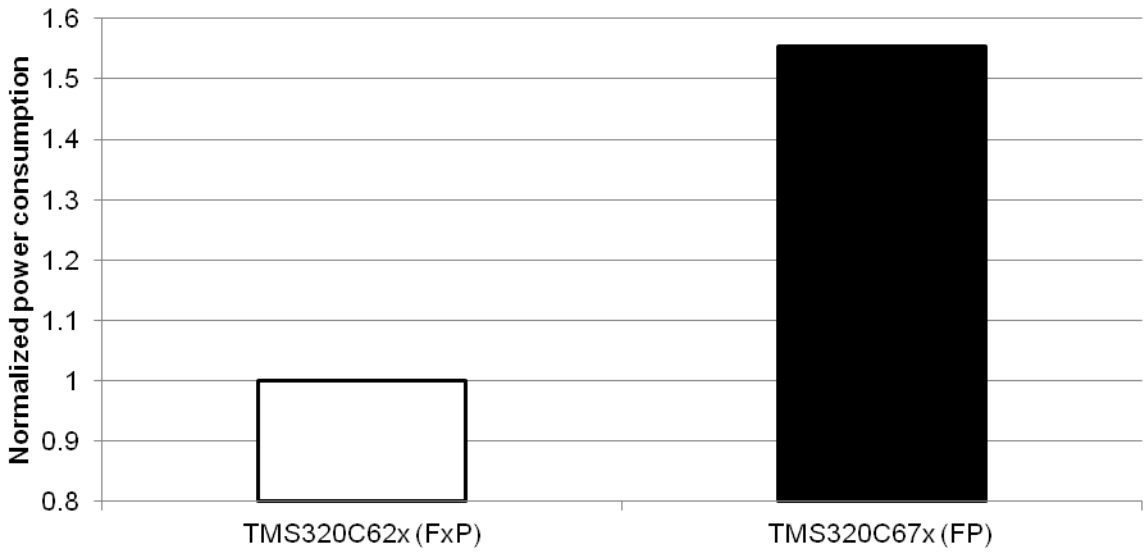


Figure 1.1: Normalized power consumption of two Texas Instruments' processors

[9]) do not provide hardware support for FP arithmetic due to its high area and power overhead. For example, Figure 1.1 shows the power consumption of a Texas Instrument FP DSP normalized to that of a similar FxP DSP [10]. In this case, providing hardware support for FP arithmetic increases the power consumption of the DSP by more than 50%. Moreover, the long latency of FP operations can also reduce the performance of some applications.

Other low-power processors such as the ARM-Cortex A9 provide an option of including vector FP (VFP) and/or NEON™ FP co-processor units. Both VFP and NEON consume significant die area (shown in Figure 1.2), and can considerably increase the leakage energy consumption of the processor. Moreover, the communication between the core and the co-processor and clocking the pipelines of co-processors may greatly increase the dynamic energy consumption of the processor. Finally, embedded ARM processors may or may not have VFP/NEON units. Thus, programmers often prefer FxP codes or FP software emulation to ensure that applications are not limited to platforms with NEON/VFP units.

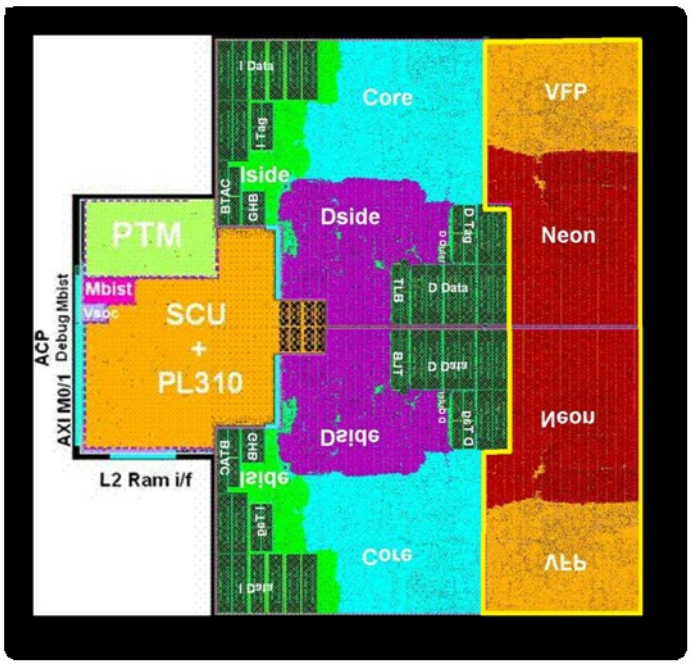


Figure 1.2: Layout of an ARM® Cortex A9 processor with VFP and NEON™ co-processors [90].

For architectures that do not provide hardware support for FP arithmetic, FP applications and algorithms need to be converted to use FxP arithmetic. This FP to FxP code conversion is a tedious and time-consuming process that can require a significant portion of the software design time (up to 30%) [11]. Moreover, the FxP code typically requires additional instructions to avoid overflows and underflows during computations. These additional instructions can also reduce the performance and energy efficiency of embedded architectures. Consequently, a significant portion of the overall design time is spent in converting FP code to FxP code and ensuring that both signal-to-noise ratio (SNR) and computational performance of the converted application meets the design requirements.

In the past, software for embedded processors was designed by a limited number of programmers that were well-versed with the underlying architecture. Dedicated software design teams

were employed for embedded software development. However, modern embedded processors are often programmed by non-specialist programmers (e.g. Android and iOS application development). It is thus essential to ensure that improvements in power and energy efficiency of these architectures do not come at the cost of increased programming complexity.

1.2 RESEARCH OVERVIEW

The research described in this dissertation focuses on improving power efficiency (IPC/Watt) of embedded processors without increasing their programming complexity. It proposes techniques for improving compute efficiency of both low-power digital signal processors and low-power GPUs for embedded systems.

Since FP arithmetic is becoming increasingly common in many embedded applications, this research proposes approaches to reduce the power overhead of FP units and the performance impact of the long latencies of FP operations. Specifically, this research proposes a novel approach for reducing the lengths of dependence chains in FP ADD and FP fused multiply add (FMA) operations. The research demonstrates that the proposed approach for handling dependence chains of FP operations can improve both performance and power efficiency in embedded processors [12, 13], out-of-order (OOO) processors [14] and hardware accelerators [15].

To reduce the area and power overhead of FP hardware in low-power embedded processors, this research proposes virtual-FPUs that re-use existing FxP units of the processor to provide architectural support for FP operations [16]. For applications that perform a high percentage of dependent FMA and dependent ADD operations, virtual-FPUs can provide a performance improvement over dedicated FPUs at a negligible area and power cost.

Integrated GPUs are becoming increasingly common in embedded processor architectures. GPUs typically have high power consumption due to the large amount of hardware resources required to support thousands of hardware threads. Due to the limited power budget of embedded processors, these integrated GPUs are thus power-constrained, as well. This research proposes techniques for improving the power efficiency of GPU architectures for GPGPU applications. GPUs typically have deep pipelines that allow them to efficiently share hardware resources between different threads and manage resource conflicts. These deep pipelines increase the read-after-write (RAW) latencies of instructions and can significantly deteriorate the performance of many GPGPU applications. Incorporating a data forwarding network (DFN) in GPUs can add considerable power overhead, due to a large number of forwarding paths.

This research proposes a low-overhead compiler-directed data forwarding network that can reduce the RAW latencies of a large percentage of dynamic instructions without incurring the high power overhead of a traditional DFN [17]. A further reduction in RAW latency is achieved by employing the proposed FMA unit in the GPU. The proposed approach also enables lower GPU power consumption by reducing the operating voltage of the execution units without increasing FP RAW latencies.

To further improve the power efficiency of GPUs, this research proposes to utilize the FP pipeline in GPUs to accelerate dependent integer instructions through the use of composite instructions [18]. These composite instructions are identified by the compiler and are formed by combing two dependent instructions. Further performance and power improvements are achieved by exploiting the computational redundancy in multiple threads executing on the GPU [18]. Computational redundancy exists when the operand values for a single-instruction multiple-threads

(SIMT) instruction are identical across all the co-issued threads. These computationally redundant instructions are dynamically detected and executed on a low-power scalar pipeline.

Finally, a bit-sliced GPU architecture is proposed to improve instruction throughput and reduce power consumption [18]. The bit-sliced architecture allows more efficient usage of register file (RF) and execution bandwidths by allowing instructions whose operands require 16 or fewer bits to be issued simultaneously to half bit-width execution slices. This significantly increases instruction throughput in GPUs while reducing the power consumed by the RF and the execution units.

1.3 RESEARCH CONTRIBUTIONS

This dissertation proposes a novel hardware FMA unit design that significantly improves the performance of FP applications and also considerably reduces the area and power overhead of FPUs. The proposed FMA design exploits common FP data trends to improve FP pipeline utilization and reduce the lengths of FP dependence chains. Due to reduced power consumption and improved performance, the proposed design considerably improves the power efficiency of the baseline DSP processor (up to 40%). FMA units derived from the proposed design can also be utilized in OoO processors and hardware accelerators for scientific applications to provide significant performance and energy improvements.

To ameliorate the hardware overhead of FPUs this dissertation proposes virtual-FPUs that allow performance very close to dedicated FPUs at a negligible area and power overhead over FxP processors. These virtual-FPUs simplify the FP execution pipeline by only supporting truncation (instead of all IEEE-754 rounding modes) and flushing subnormals to zero. Virtual-FPUs can

either be used instead of dedicated FPUs or for avoiding access to FP co-processors to save leakage and dynamic power. Dedicated FP co-processors can thus be limited to cases where exactly rounded results and/or support for subnormals is required.

To improve the performance of power-constrained integrated GPUs in embedded architectures, this dissertation proposes compiler directed forwarding (CDF). CDF reduces the effective RAW latencies of instructions in GPUs without incurring the power overhead of a regular DFN. CDF combined with the proposed FMA unit also allows a significant reduction in GPU peak-power consumption by re-using the single precision (SP) FMA units for double precision (DP) operations and reducing the operating voltage of the execution units with negligible performance penalty. This allows a geometric mean performance improvement of 23% without increasing the power budget of the GPU.

Finally, this dissertation proposes to improve the power efficiency of GPUs by exploiting common instruction and data patterns in GPGPU applications. The proposed techniques improve instruction throughput while reducing switching activity (and hence dynamic power) in the GPU. These techniques include re-using the FMA units in GPUs to accelerate execution of integer instructions, dynamically detecting and handling computationally redundant instructions and exploiting the low bit-width of RF operands to improve GPU power efficiency using a bit-sliced architecture. The proposed techniques improve the power-efficiency of the GPU by 25%.

1.4 DISSERTATION OUTLINE

The remained of this dissertation is organized as follows: Chapter 2 provides a background of floating-point representations, DSPs and GPUs, and surveys related work. Chapter 3 discusses

some alternatives to dedicated FP hardware and their impact on programming complexity, performance and energy consumption. Chapter 4 describes and evaluates the proposed FP FMA design. Chapter 5 describes the virtual-FPUs and their impact on power, performance and energy consumption. Chapter 6 proposes techniques to improve the performance of power-constrained GPUs by reducing their effective pipeline RAW latencies. Chapter 7 proposes approaches to improve the power efficiency of GPU architectures. Chapter 8 summarizes the dissertation and Chapter 9 describes some opportunities for future research.

2. BACKGROUND & RELATED WORK

This section provides a background on digital signal processors (DSPs), graphics processing units (GPUs), floating-point (FP) representations, rounding modes and the differences between FP and fixed-point (FxP) arithmetic. Previous work related to low-power FP approaches and energy-efficient GPU architectures is also discussed.

2.1 DIGITAL SIGNAL PROCESSORS

DSPs are typically statically-scheduled, very long instruction word (VLIW) processors that are often employed in embedded architectures. Static scheduling allows DSPs to significantly reduce the hardware complexity of the architecture resulting in reduced power consumption. The baseline DSP architecture simulated in this dissertation is similar to a Texas Instrument TMS320C67X series processor [15]. The processor is assumed to run at a 1GHz clock and has a single-instruction multiple-data (SIMD) unit with eight lanes.

For the FxP processor, each SIMD lane consists of an integer ALU and an integer fused multiply-add (FMA) unit. The SIMD lanes of the FP processor include the integer ALU and an FP FMA unit that can also execute integer multiply-add instructions. The data transfers between the vector register file and the scalar register file occur through the level-1 (L1) scratchpad memory. The FP units in the baseline processor only support single-precision operands. All integer execution units are 32-bit wide. Operations that can be executed by the SIMD unit are identified at compile time and appropriate load and store instructions for transferring data between the vector and scalar register files are added by the compiler. All operations are statically scheduled by the compiler.

2.2 GRAPHICS PROCESSING UNITS

GPUs are massively parallel processors that offer a large pool of execution units, high-bandwidth memories, and thousands of hardware threads. Although originally designed for graphics applications, the high compute power and memory bandwidth of GPUs is also being used to accelerate general purpose (GPGPU) applications that can efficiently utilize GPU resources.

The peak compute performance of a GPU is a function of its compute resources and their operating frequency. Consequently, the peak compute performance of GPUs has traditionally been improved by increasing the number of compute resources and/or their frequency. However, modern GPUs are typically power-constrained (e.g., a GeForce® 8800 consumes 280W [19]) and cannot afford to increase compute resources or frequency within their power and thermal budgets[20]. Moreover, as power supply voltage scaling diminishes, the energy cost per instructions must be decreased to achieve performance improvements in future processors [21].

The compute resources of GPUs are organized into groups, such as streaming multiprocessors (SMs) in NVIDIA® GPUs. Each SM is composed of a single-instruction multiple-thread (SIMT) pipeline that consists of execution units, main register file (MRF), fetch, decode and scheduling (FDS) logic, and memories. For compute-intensive applications more than 85% of the total GPU dynamic power may be consumed by the SMs [22]. The FDS, MRF and execution units can contribute to more than 45% of an SM's dynamic power [22]. Reducing the power consumption of these architectural components is thus imperative for a power efficient GPU architecture. This dissertation focuses on these power-hungry components of an SM.

2.3 FLOATING-POINT REPRESENTATIONS

FP representations allow numbers over a large range to be represented with the same worst-case relative representation error. The large range of FP representation is achieved by encoding the number to contain separate exponent and significand bits. The IEEE-754-2008 FP Standard specifies that all FP numbers shall be represented as:

$$(-1) \times sign \times significand \times 2^{(exponent\text{-bias})},$$

where *sign* is a single bit that indicates the sign of the number and bias is a constant used to ensure that the exponent values are symmetric around zero. Most embedded processing applications only requires the single precision FP format that contains an 8-bit exponent field, 23 bits for the significand and 1 sign bit. The bias value for the IEEE single-precision FP format is 127.

2.3.1 NORMALIZED FP NUMBERS

Normalized FP numbers assume a hidden '1' in the most significant bit of the significand (making the total width of the significand 24 bits). Normalized numbers have an exponent range from 1 to 254 (i.e., from -126 to 127 after applying the bias). Typically, most of the FP numbers encountered in real-world applications are normalized. Consequently, hardware floating-point units (FPUs) are generally optimized for normalized FP numbers.

2.3.2 SUBNORMALS

Subnormals are the smallest non-zero numbers that can be expressed in the FP format. Subnormals do not have a hidden '1' for the MSB of the significand and use a reserved exponent value of zero. Subnormals occur relatively infrequently in real-world data. Consequently, some architectures employ software traps to handle subnormal computations or flush subnormals to zero [23].

2.3.3 FLOATING-POINT ARITHMETIC

Since the FP format is an encoded representation, arithmetic operations on FP operands need to check each bit-field to determine the represented number. Arithmetic operations are often dependent upon the different bit-fields of the representation. For example, subnormal numbers requires different handling than normalized numbers. Moreover, a FP format also has reserved representations for special cases such as ±Infinity and Not-A-Number (NaN). Arithmetic operations on these FP numbers also require different handling than normalized numbers and subnormals. Finally, even for normalized numbers, the arithmetic operations often involve several steps that depend upon the signs, exponents or significands of the numbers. Consequently, FP arithmetic is considerably more complicated than FxP arithmetic.

2.4 FLOATING-POINT TECHNIQUES IN EMBEDDED SYSTEMS

Low-power digital signal processors (DSPs) that provide hardware support for FP arithmetic generally customize the FP hardware for a particular application or application domain to reduce the associated overheads. Some approaches for providing hardware support for FP arithmetic in DSPs include *block floating-point*, *light weight floating-point,* and *fractured floating-point*.

Block floating-point (BFP) has been used for hardware implementations of digital filters and fast Fourier transforms [24, 25, 26]. In BFP arithmetic, a single exponent (block exponent) is used for all elements in a block of data. For performing arithmetic operations on elements of the same block, the hardware generally does not need to account for the block exponent. Consequently, all arithmetic operations are performed on the significands using FxP hardware. Once the FxP hardware produces the results, the exponent field is used to scale the results to obtain their actual values. The elements within the block are normalized to avoid underflows during

subsequent operations. Normalization removes all leading zeros (the most significant bits (MSBs) with value zero) from the largest element in the block. Since all elements in the block share a single block exponent, they are all normalized by the same amount. Consequently, all elements in the block other than the largest element may still have some leading zeros after normalization. In some applications, however, a predetermined number of zeros may be left in the largest data element to prevent overflow.

The primary advantage of BFP arithmetic is that all operations can be performed with FxP hardware, instead of FP hardware. BFP thus allows a wide range of numbers with much less hardware overhead than dedicated FPUs. BFP is particularly suited to digital filters and fast Fourier transforms where the coefficients and twiddle factors are fixed and known in advance, making it easier to avoid overflow. However, even for adaptive digital filters, like least mean square (LMS) filters, where the filter taps change in response to the output of the filter, efficient grouping and scaling of data elements in a single block can become an arduous task. A naive non-optimal grouping and scaling of data elements can cause a significant loss in signal-to-noise ratio (SNR). Consequently, application of BFP arithmetic for more complicated DSP algorithms can be particularly challenging. Mitra *et al.* propose a BFP implementation of the LMS filter [24], where different scaling values are selected for the filter coefficients and filter inputs.

The conversion of algorithms to efficiently utilize BFP arithmetic can increase the software design time of applications. Moreover, this conversion typically requires additional instructions to be added to the code. These instructions are used to find the largest element in a block and normalize the elements of the block. If overflow checks are also performed in software, conditional branches need to be added to the code. These conditional branches can result in further

performance reduction. If hardware support is provided to deal with overflows and underflows, the area overhead approaches that of FP hardware [27].

Light-weight floating-point reduces the area and power overhead by modifying the way in which FP numbers are represented and how operations are calculated. In other words, light-weight FP does not comply with the IEEE-754 floating-point standard [28]. One modification is to use a customized FP format with smaller exponent and/or significand bitwidths. This in turn reduces the size of adders, shifters and multipliers required in the data path. Light-weight FP is particularly advantageous for various digital signal processing applications that do not require a significand precision and exponent range as high as specified in the IEEE-754 Floating-point Standard. Gafar *et al.* [29] and Fang *et al.,* [30] propose to optimize the significand precisions and exponent ranges for various embedded applications including discrete cosine transforms, FIR filters, ray tracing, and speech recognition. Gafar *et al.* determine the precision of each variable separately according to the sensitivity of the final result to the precision of that variable, while Fang *et al.* investigate the case where all the variables have the same bitwidths. Their study for the DCT algorithm in video compression suggests that 8 bits for the significand and 5 bits for the exponent ensure sufficiently accurate results.

Although light-weight FP can greatly reduce the area and power consumption of FPUs, the non-compliance to IEEE-754 Floating-point Standard can limit the architecture to a limited number of applications. Modern software defined radio (SDR) architectures are likely to support a wide range of applications that may have drastically different dynamic ranges and precision requirements. Non-compliance with IEEE-754 floating-point standard will increase the software design time due to additional simulations required to determine the impact of reduced precision

and range on the application performance. Moreover, for applications such as multiple-input multiple-output (MIMO) wireless communications, matrix decompositions like singular value decomposition (SVD) may produce different results for different precisions. For MIMO approaches that require matrix decompositions at both the base-station and the mobile terminal, different decomposed matrices at the two ends can drastically increase the transmission error rates.

Although software emulation can be used to perform FP arithmetic operations on FxP processors, it leads to significant performance and energy overheads [16]. Hockert *et al.,* propose a fractured FPU approach that adds custom instructions to the MicroBlaze™ soft core processor to accelerate software emulation [31]. They propose different levels of acceleration by speeding up different portions of the SoftFloat subroutines that emulate FP arithmetic operations. Their approach can result in a considerable reduction in the area overhead, but is likely to increase the energy consumption of FP operations.

2.5 FLOATING-POINT TO FIXED-POINT CODE CONVERTERS

There have been several published works on automated FP to FxP code conversion of DSP algorithms [32, 11, 29]. These converters typically perform dynamic range and precision analyses on the variables used in FP code. Kedding *et al.* propose a FxP design and simulation environment (FRIDGE) to reduce the FP to FxP conversion cost [32]. Their approach requires the programmer to annotate the FP code with some information that can be used to reason about the FxP format requirements of variables. These annotations may initialize constants or specify the FxP formats for some variables. Another set of global constraints is specified to ensure that the over-

all precision of the results of the algorithm are guaranteed. Once these annotations have been specified, their algorithm uses an interpolative approach to estimate the precision required for other variables in the code. If the global constraints fail at any point, a more intricate algorithm redesign is required to reduce the algorithms precision requirements. Kedding *et al.* also introduce FxP data types to reduce the programming complexity associated with conversions between different formats. The FxP data types also reduce the programming complexity by implicitly performing quantization, wrap-around or saturation during computations.

Menard *et al.* propose an approach for FP to FxP code conversion that maximizes the accuracy of linear time-invariant algorithms while minimizing the size and execution time of the code. Their approach estimates the dynamic range of the algorithm using a combination of FP simulations with a representative input data set and a mathematical analysis of the algorithm. Once the analysis is completed, the dynamic ranges of different variables are annotated in the control flow graph. Code generation is then performed according to the different data-types provided by the DSP architecture to minimize the number of shift operations while estimating the impact of different word-lengths on the SNR of the application.

In general, most of these approaches have similar shortcomings. First, all of these approaches require the dynamic range of inputs to be either known or estimated in advance using analytical methods or statistical simulations. The analytical methods can overestimate the dynamic range requirements and the methods based on statistical simulation cannot guarantee overflow protection [11]. Consequently, these approaches may reduce the SNR or incur unnecessary shift operations. While a reduction in SNR may require an algorithm redesign, an increase in the number of shift operations can increase the execution time and energy consumption of the code.

2.6 HARDWARE ACCELERATORS

Sergyienko *et al.* propose an FPGA-based hardware accelerator for QR decomposition employed in MIMO wireless communications [33]. They synthesize an array processor and map a QR decomposition algorithm based on Given's rotations onto the array processing units. Their approach employs 32-bit FxP data-path with eight PUs for a 9 × 9 matrix. The processing units internally use functional units based on the co-ordinate rotation digital computer (CORDIC) for implementing the algorithm.

Karkooti *et al.* propose a matrix inversion kernel for MIMO communications that utilizes a systolic array architecture. For arithmetic operations, they use a reduced-precision FP format with 14 significand bits, six exponent bits, and one sign bit. Eilert *et al.* propose an FP matrix inversion core for a 4 x 4 complex values matrix [34]. They design a complete FP multiply accumulate unit with reduced precisions of 16 and 20 bits.

Luethi *et al.* propose an ASIC-based approach for QR decomposition for MIMO wireless systems [35]. They employ FxP arithmetic in their design, but take an approach similar to BFP to properly scale columns of the matrix to maintain proper precision. Ahmadsaid *et al.* propose an FPGA-based systolic architecture for SVD [36]. They implemented CORDIC-based processing elements for the systolic array.

2.7 ENERGY-EFFICIENT TECHNIQUES FOR GPUS AND OUT-OF-ORDER PROCESSORS

GPUs typically employ large register files to maintain the context for thousands of active threads. Due to their size and banked organization, register files consume a significant portion of

the total GPU power [22]. Gebhart *et al.* propose register file caches to reduce the register file accesses and smaller and low power register file structures managed at compile time [37, 38]. With their approach, most of the register file accesses can be re-directed to smaller register file structures with considerably less power consumption. While the research presented in this dissertation also introduces small register files in the GPU, the purpose of these register files is not limited to reducing register file accesses. They also help to considerably improve the performance of the GPU and reduce the energy consumption of execution and fetch-decode and schedule (FDS) stages.

Kim *et al.* propose a macro-op scheduling technique for out of order (OOO) processors to reduce the cycle time constraints from the scheduling logic [39]. Dependent integer instructions are dynamically detected and scheduled as a group. Each instruction within the macro-op is still issued separately. In contrast, the research presented in this dissertation forms composite instructions that are identified at compile time. These composite instructions can reduce the total number of fetched and decoded instructions. Moreover, they utilize the FP pipeline in GPUs to accelerate the execution of integer instructions. This results in reduced execution unit energy consumption and a considerable performance improvement.

Brooks *et al.* propose operation packing for OOO processors. They employ a sub-word parallel approach for improving the performance of reduced bit-width operands in OOO processors [40]. Their approach uses a bit-sliced ALU to concurrently execute two instructions. Both instructions must perform the same arithmetic operation and cannot have any data dependencies. The number of instructions that can take advantage of their approach thus depends upon the number of dependent instructions in the instruction window. While a large instruction window can increase

the number of bit-sliced instructions, it is likely to significantly increase the power consumption of the processor. Instead of only bit-slicing the ALU, this research proposes a sliced GPU architecture that divides the register file, the schedulers and the execution units into two slices. Slicing the schedulers and the register file along with the execution units allows more efficient use of these critical GPU resources and considerably improves GPU performance.

My proposed GPU architecture can execute two sliced instructions from different threads simultaneously. Moreover, the absence of data dependencies between different threads, which is typical in many GPGPU applications, allows more opportunities for performance improvements. Moreover, the scheduling logic in an OOO processor is often timing critical [39]. Efficiently finding instructions that can be executed in a sub-word parallel form is likely to increase the critical path of the processor. GPUs however, schedule instructions at a much lower clock rate. A single scheduled warp issues over four cycles before the next warp is scheduled. This additional scheduling time allows GPUs to potentially make more complex scheduling decisions without impacting performance.

Hameed *et al.* utilize customizable processors to implement application-specific functional units to improve the performance of CPUs [41]. Instead of adding customized functional units in GPUs, I utilize the resources of the FP execution units to efficiently map common integer instructions. Ergin *et al.,* utilize the reduced bit-width operations to reduce register file pressure in OOO processors. Our approaches, however, strive to improve instruction throughput in multi-threaded GPUs [42].

3. ALTERNATIVES TO FLOATING-POINT HARDWARE

This section describes some alternatives to floating-point (FP) hardware and their impact on the programming complexity, performance, power and energy consumption of the processor. These alternatives are typically employed in low-power processors that do not provide hardware support for FP arithmetic (e.g. Texas Instruments TMS320C64x [7], Sandbridge Sandblaster DSP [8], ARM Ardbeg [9]). While the lack of FP support reduces power dissipation and chip area, it can significantly increase the programming complexity of these processors. The most common approaches to performing FP arithmetic in FxP processors include conversion of FP code into fixed-point (FxP) code and software emulated FP arithmetic.

3.1 FIXED-POINT CODE CONVERSION

For FP to FxP code conversion, a baseline FP code is typically implemented first as a reference. Once the FP implementation is completed, a representative data set is used to estimate the range of numbers in the data set. This is used to guide the selection of the FxP format and FxP code implementation. The results for FxP implementation are compared with the FP implementation to ensure that the final output of the FxP implementation has satisfactory accuracy (e.g. by determining the signal to noise ratio (SNR)). FxP codes are thus tailored to the data-range and the error tolerance of the application.

3.1.1 FIXED-POINT REPRESENTATION

An FxP number can be considered as a scaled integer where the scaling factor is a negative power of two. An FxP representation assigns a fixed number of bits per word for representing the integer and fractional parts of a number [43]. The integer word length (IWL) represents the num-

```
float SquareRootFloat(float
number) {
 long i;
 float x, y;
 const float f=1.5F;
 x=number*0.5F;
 y=number;
 i=*(long *)&y;
 i=0x5f3759df-(i>>1);
 y=*(float *)&i;
 y=y*(f-(x*y*y));
 y=y*(f-(x*y*y));
 return number*y;
}
```

(a)

```
#define step(shift) \
if((0x40000000l>>shift)+root<=value){\
 value-=(0x40000000l>>shift)+root;\
 root=(root>>1)|(0x40000000l>>shift);\
}else{\
 root=root>>1;\
 }

fp14 fpSqrt2(fp14 value){
 long root=0;
 for (int i=0; i<=30; i+=2)
  step( i);
 if(root < value) ++root;
 root <<= 7;
 return root;
}
```

(b)

Figure 3.1: Square root routines: (a) Floating-point [44], (b) Fixed-point [93]

ber of bits used to represent the integer part. The fractional word length (FWL) represents the number of bits used to represent the fractional part of the number. An *IWL*:*FWL* FxP format utilizes *IWL* most significant bits (MSBs) for storing the integer part and *FWL* least significant bits (LSBs) for storing the fractional part of the number. For a two's complement representation the total word length (WL) of a FxP representation is equal to *WL=FWL+IWL+1*.

The range (*R*) and quantization step (*Q*) represent two important properties of a FxP representation. *R* determines the total representable range of numbers while *Q* determines the maximum possible representation error (quantization noise). *R* and *Q* of a given FxP representation can be determined as shown in Equations 1 and 2 respectively.

$$R = [-2^{IWL}, 2^{IWL}) \tag{1}$$

$$Q = 2^{-FWL} \tag{2}$$

3.1.2 FIXED-POINT ARITHMETIC RULES

Addition, subtraction and comparison

Two FxP numbers can be added, subtracted and compared by a normal integer unit if they have the same *IWL*. Otherwise the number with the smaller *IWL* needs to be scaled (by right shifting it) until both numbers have the same *IWL*. However, this scaling may result in a loss of accuracy as the least significant bits (LSBs) of the FxP number are shifted out.

Multiplication

The two's complement integer multiplication of two words yields a result of $2WL+1$ bits. The IWL of the results is given by:

$$IWL_{result} = IWL_a + IWL_b + 1$$

However, in many processors, integer multiplication only returns the least significant WL bits. An overflow is signalled if the discarded MSBs of the result are non-zero. FxP multiplications, however, require access to the MSBs of the result. There are three different approaches that can be taken to implementing FxP multiplications:

I. If the upper word of the result is accessible in the architecture using specific instructions (e.g. digital signal processor (DSP) architectures such as the Texas Instrument TMS320C64), the source code can be modified to directly access the MSBs to obtain the FxP results

II. The source operands can be scaled (right shifted) before a multiplication. However, this can greatly increase the computation error due to the loss of LSBs in the source operands.

III. A software routine/macro can be implemented to compute the MSBs of the result.

This work assumes that the MSBs are accessible in the FxP processor architecture. This results in less overhead for FxP multiplication than the other approaches.

Division

Two FxP numbers can be divided using a normal integer divider. The *IWL* of the result (IWL_{result}) is given as:

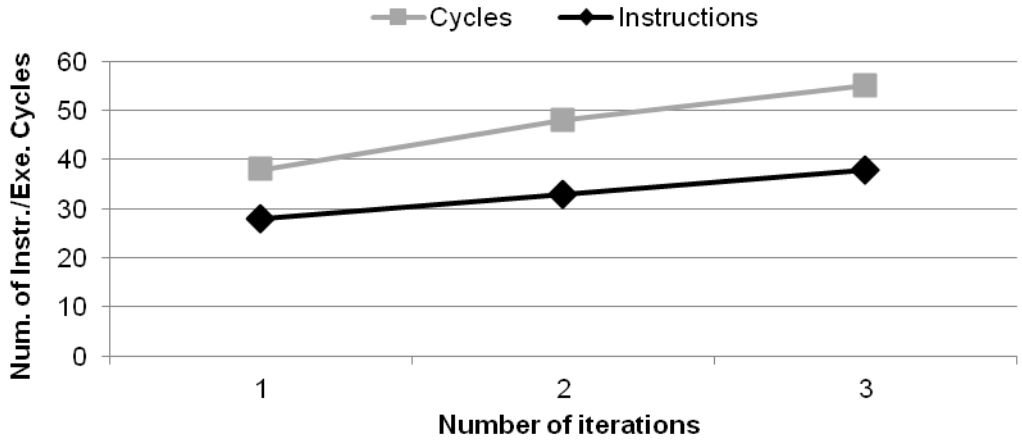
$$IWL_{result} = WL - 1 + IWL_{num} - IWL_{den}$$

where IWL_{num} and IWL_{den} are the *IWLs* of the numerator and denominator, respectively. The division operation can result in a high loss of accuracy if the source operands are not carefully scaled. For example, if IWL_{den} is small, then the accuracy of the result is poor (few fractional bits) and if $IWL_{num} > IWL_{den}$, the result may not be able to be represented within *WL* bits. FxP division is typically implemented by scaling (left shifting) the numerator before division. This increases the number of fractional bits (and thus the accuracy) available in the result. However, the programmer must ensure that the integer part of the expected quotient can fit within IWL_{result} bits.

Square-root

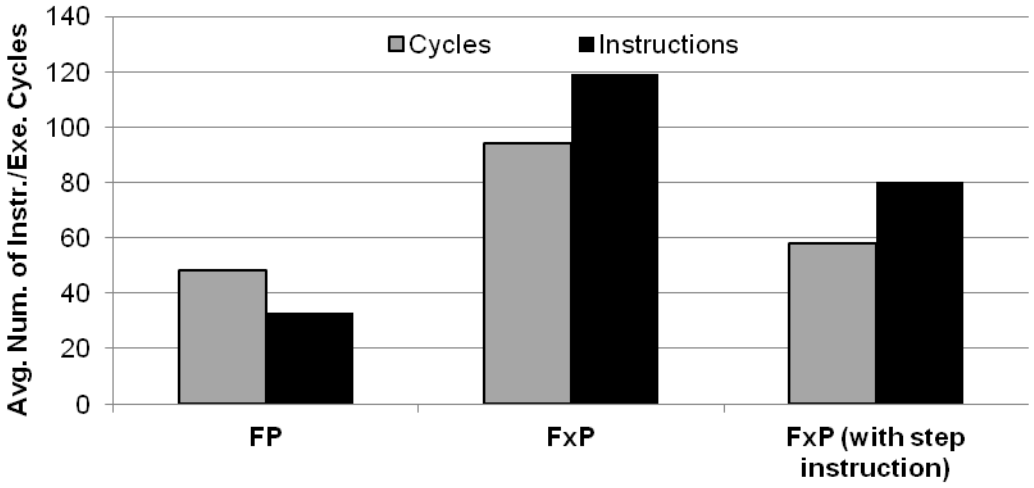
Both FxP and FP square root operations are implemented as software routines in this work. The implementations are shown in Figure 3.1(a) and (b) for FP and FxP numbers respectively. Both routines compute square root iteratively. The FP square root routine performs two iterations (two iterations provide sufficient accuracy for graphics applications) [44]. The FxP routine calculates the square root of a FxP number by determining the integer square root of the integer value represented by the FxP number. The integer square root result is then scaled according to the scaling factor of the FxP representation. The FxP square root implementation is less accurate than the FP implementation. However, it is significantly faster than other FxP square root algo-

rithms that increase the accuracy of the result at the cost of much longer execution times. For a conservative performance comparison of FP and FxP implementations, all of the FxP codes developed for this research use the fast FxP routine and assume that the accuracy provided by the FxP implementation is sufficient for the application requirements.



Mean Error	3.8×10^{-3}	2.0×10^{-4}	0.0

(a)



Mean Error	2.0×10^{-6}	3.8×10^{-3}	3.8×10^{-3}

(b)

Figure 3.2: Total number of execution cycles and dynamic instructions for square root routines: (a) Floating-point, (b) FP and FxP square root comparison

Some embedded architectures provide custom instructions to improve performance (execution time) of FxP square root operations. These instructions provide hardware support for executing the *step* macro (shown in Figure 3.1(b)) with a single instruction. The performance and accuracy impact of FP and FxP square root routines (with and without the *step* instruction) is analyzed below.

Figure 3.2 shows a comparison of different square-root routines for FxP and FP DSPs similar to TMS320C64x and TMS320C67x respectively. Figure 3.2(a) shows the number of dynamic instructions executed in computing a square root of a number using the FP square root routine. The number of instructions increases with an increase in the number of iterations. The total number of iterations (and thus the number of executed instructions) depends upon the required result accuracy. Figure 3.2(a) also shows the mean relative error for different numbers of iterations. The relative error is computed by calculating the square root of pseudo-randomly generated single precision FP numbers using the software routine (shown in Figure 3.1(a)) and the standard math library in C. The pseudo-random FP numbers are generated using the *rand* function in the standard C library. As shown in Figure 3.2(a), the relative error decreases considerably as the number of iterations increases. Beyond two iterations no errors are observed in the results of square root operations. For most embedded systems, the accuracy of the result obtained with two iterations is sufficient. The FP square-root routine takes about 48 cycles and executes approximately 33 instructions with two iterations.

Figure 3.2(b) shows the number of execution cycles and the number of dynamic instructions executed during the FP square root routine (with 2 iterations) and the number instructions and cycles consumed by the FxP routine. For FxP square root implementation, I show results for both

the software only approach and with hardware support for the *step* macro. The horizontal axis is also marked with the mean relative error observed for each of the three approaches.

A key observation from Figure 3.2(b) is that even with just two iterations, the FP square root routine exhibits considerably lower relative error than the FxP routines. Although more complex FxP square routines can be employed to improve the accuracy of computed results, they take considerably more cycles and dynamic instructions. The FP square root routine also takes fewer cycles and dynamic instructions than the FxP routines. Although hardware support for the *step* macro considerably improves the performance of the FxP square root routine, it still executes more dynamic instructions and consumes more cycles than the FP routine. The research presented in this dissertation assumes that the processor has hardware support for the *step* macro to allow a conservative performance comparison between FP and FxP processors.

3.2 FIXED-POINT PROGRAMMING ISSUES

FP to FxP conversion is often tedious and error prone, particularly for more complicated signal and image processing algorithms. Some of the main contributing factors for the increased programming complexity include (i) amplified quantization noise; (ii) rounding errors; (iii) trade-offs between range and accuracy of FxP representations; (iv) assembly-level programming; and (v) the impact of changing problem size on the SNR of the FxP code. These are described in more detail below.

Quantization noise refers to the error introduced in the actual value of the number when it is represented by a finite number of bits. While both FP and FxP representations of a number can introduce quantization noise, the relative quantization error in an FP representation is typically

considerably lower than in an FxP representation. This quantization noise can significantly impact the fidelity and stability of DSP algorithms [2],[3]. From a programming perspective, a software developer needs to understand the impact of quantization noise on the algorithm. This typically involves simulations of the FxP versions of the code to determine the SNR in the computed results and may also require algorithmic modifications, such as adding separate input dither signals [4], to make the FxP code more resilient to quantization noise. Consequently, embedded software design teams generally include DSP algorithm specialists to perform algorithm analysis and modifications based on the error trends in the FxP implementations.

Rounding-errors are introduced in FxP arithmetic due to the limited bit-width available for FxP computations. Although, FP computations also result in rounding errors, the relative rounding error in FP operations is typically less the rounding error in FxP operations. Many embedded processors (e.g., Texas Instruments' TMS320C64X) provide long accumulators to reduce the impact of rounding-errors during accumulation operations [5]. While these accumulators are suitable for multiply-and-add based operations, algorithms that do not perform such dependent multiply-and-add operations can still suffer from the high rounding errors. Moreover, these accumulators often require programming in assembly language or highly sophisticated compilers to utilize them properly.

Arithmetic operations on FxP numbers need to be guarded against *overflows and underflows*. This typically requires hardware or software support for saturating or wrap-around arithmetic [6]. If saturating arithmetic is not supported in hardware, the programmer needs to add code for magnitude comparisons and saturations. While saturation may be suitable for some applications, for kernels such as matrix inversions, saturation can lead to unacceptably large errors in the in-

verted matrix. In such cases, the programmer needs to ensure that the magnitude of inputs is limited to a specified range to ensure that no overflows or underflows occur during computations. If the input range cannot be estimated at design time, the programmer needs to include additional code to handle overflows and underflows during computations.

An FxP representation also involves a *trade-off between the range of representation and the representation accuracy*. The FxP representation selected for a particular algorithm should provide enough fractional bits to guarantee sufficient accuracy in results. Moreover, the representation also needs to allow sufficient integer bits to cover the dynamic range of the algorithm. Since an increase in the fractional bits, without increasing the total number of bits in the representation, results in a reduced dynamic range, a programmer needs to carefully explore different representations to determine the representation that meets all the requirements of the application [7].

FxP implementations are also dependent upon the *problem sizes*. For example, the number of points for fast Fourier transforms (FFTs) used in orthogonal frequency division multiplexing (OFDM) modulation may vary from 256 (IEEE-802.16e [8]) to 8192 (DVB [9]). FxP code written for one problem size may not be suitable for other sizes. This is because increasing the problem size often requires more protection against overflows and may increase the required dynamic range of the algorithm. Moreover, rounding errors also typically increase as the problem size is increased. The accommodation for the increased dynamic range in the FxP representation may result in a significant loss of precision, which can further increase rounding errors. Handling such scenarios may require more complicated modifications, such as block floating-point (BFP) or dynamically changing the FxP representation during execution to deal with the high dynamic range [10]. Consequently, the software designer may need to develop different versions of the

```
0  for (i=0; i<n; i++)
1   acc +=  a[i]*a[i];
2       b = b/acc;
```

(a)

```
0  for (i=0; i<n; i++)
1     acc += (a[i]*a[i]) >> FRACBITS;
2  b = (b << FRACBITS)/acc;
```

(b)

```
0  for (i=0; i<n; i++)
1     $reg0 += mul_acc(a[i],a[i]);
2  $reg0 = shr($reg0,FRACBITS);
3  b = div_accreg0(b << FRACBITS);
```

(c)

```
0  blk_exp = find_blk_exp(a);
1  for (i=0; i<n; i++)
2  scaled[i] =  a[i] >> blk_exp;
3  for (i=0; i<n; i++)
4  acc +=  scaled[i]*scaled[i];
5  b = b << blk_exp;
6  b = b/acc;
```

(d)

Figure 3.3: Floating-point code and equivalent fixed-point codes. (a) FP code, (b) FxP code, (c) FxP code with long accumulators, (d) BFP code

FxP code according to the problem size and the error tolerance of the application.

Finally, the programming complexity of FxP implementations is also greater than the complexity for floating-point implementations due to the many instructions included in the instruction set architecture (ISA) for performing the same FxP operation. For example, the SHARC DSP ISA specifies both FP and FxP multiplication instructions [11]. While the ISA only includes a single FP multiply instruction, in order to properly handle different FxP code requirements, several instructions are specified for integer multiplications. These include instructions to specify whether the inputs and outputs are signed, unsigned, fractional or rounded. Moreover, different instructions are required to handle different sources and destinations, such as register file operands or the accumulator registers. For operations that utilize the long accumulators, the ISA specifies different instructions to indicate whether the operands should be rounded or saturated before

computations or whether the result needs to be rounded or saturated before writing to the destination register. Finally, for implementing more complex algorithms without introducing a high quantization noise, the ISA specifies different instructions for manipulating the multiplier accumulator registers.

Examples of FP to FxP and BFP code conversion are illustrated in Figure 3.3. Figure 3.3(a) shows an FP code snippet that multiplies and accumulates the product in variable *acc*. Another variable *b* is then divided by *acc*. Such code sequences are commonly used for computing vector norms in many FP-intensive applications that deal with matrices.

3.2.1 FIXED-POINT IMPLEMENTATION

Figure 3.3(b) shows a FxP implementation of the FP code. The primary changes involve inserting additional scaling operations for each multiplication and division operation. However, these scaling operations can result in a significant loss of accuracy and overflow/underflows. For instance, if all the elements in the array *a* are extremely small, the left shift instruction at line 1 can flush the result to zero. This can lead to a divide by zero exception at line *2*. Moreover, scaling *b* before the division instruction can result in a loss of MSBs in *b*, leading to incorrect results. These conditions can be avoided by implementing the shift operations according to the magnitude of numbers. However, this can increase the instruction count and execution time of algorithms due to the addition of data-dependent conditional branches. Consequently, most FxP implementations require that the programmer should be aware of the dynamic range and the error tolerance of the application. This allows them to prevent overflows/underflows while supporting a dynamic range and accuracy that is suitable for the requirements of the application.

Fixed-point implementation with long accumulators

An example of a FxP code utilizing long accumulators is shown in Figure 3.3(c). Although the long accumulators reduce the number of scaling operations and overflows/underflows during accumulation, the use of these accumulators typically requires assembly-level programming to access the accumulator registers. Moreover, the different types of rounding modes available for these accumulators (e.g. saturation, round to nearest, and warp around) increases the programming complexity, since the programmer must chose that appropriate instruction according to the application requirements. Moreover, the instructions other than multiply-add and adds cannot take advantage of the long accumulator and still require scaling operations (e.g., line 3 in Figure 3.3(c)).

Block floating-point implementation

Figure 3.3(d) shows an example BFP implementation of the code snippet shown in Figure 3.3(a). BFP utilizes a single exponent for a block of numbers. The exponent (block exponent) is set after determining the number with the largest magnitude in the block of numbers. All numbers are shifted left according to the block exponent to remove leading zeros. Arithmetic operations on the block of numbers are then performed similar to normal integer operations. BFP often needs hardware support to efficiently find the block exponent without requiring software routines. However, the benefit of BFP is only visible if many arithmetic operations can be performed on the block without requiring re-scaling.

For the code-snippet shown in Figure 3.3(d), the block exponent is set according to the magnitudes of numbers in array *a*. The array *a* is likely to be loaded into a vector register where the magnitude comparison is performed to determine the block exponent. The elements of array *a*

are then shifted according to the block exponent in line *2*. The accumulation is performed on the shifted numbers.

For the division operation in line *6, b* is scaled to avoid underflow. The left shift amount required for scaling needs to ensure that there is no loss of MSBs in *b*. The example code uses the same block exponent to shift *b*. However, this requires the programmer to ensure that using the same block exponent for *b* will not lead to unacceptable errors in the result. Otherwise, additional instructions need to be added to determine the block exponent suitable for *b*.

3.3 SOFTWARE EMULATION

FP arithmetic can also be emulated using software libraries such as SoftFloat [12]. The arithmetic operations performed using these libraries are generally considerably slower than both FP and FxP implementations in hardware. This is because the software routine for an FP operation requires many bit-level manipulations as well as several data dependent branches to adequately handle the different types of FP numbers, such as normalized, subnormals and special cases (NaN and ±Inf). Consequently, software emulation typically results in reduced performance and higher energy. I utilise the SoftFloat library for energy and performance comparisons [45].

Table 3.1: Baseline processor configuration and operation latencies

Resource	Units	Operation	Latency (cycles)
Scalar integer ALUs	4	Integer ADD	1
Scalar integer MUL/DIV	2/0	Integer MUL	2
Scalar FP units	0/2	Integer fused MUL-ADD	3
SIMD vector unit width	8	FP ADD/MUL	3
		FP fused MUL-ADD	4
		Integer/FP DIV	12

Table 3.2: Common FP kernels in embedded computing

Benchmark	Dimensions/Size
Finite impulse response filter (FIR)	128-tap
Autocorrelation (AUTO)	64-tap
Fast Fourier transform (FFT)	1024-point
QR decomposition (QRD)	8 x 8 matrix
SVD decomposition (SVD)	8 x 8 matrix
Cholesky decomposition (CHL)	8 x 8 matrix
Matrix inversion using QRD (MIQ)	8 x 8 matrix
Matrix inversion using SVD (MIS)	8 x 8 matrix

3.4 FIXED-POINT VERSUS FLOATING-POINT ARITHMETIC

This research shows that although FP arithmetic consumes more energy than FxP arithmetic per instruction, in general, the additional instructions in the code associated with FxP arithmetic operations, such as scaling operations, can increase the overall energy consumption relative to FP code.

3.4.1 SIMULATION INFRASTRUCTURE

In order to investigate the performance and energy impact of FP hardware in high-performance embedded DSP architectures, I simulated FxP and FP implementations of key signal processing kernels utilized in modern wireless communication, computer vision, and media processing algorithms. I used the Trimaran simulation and compilation infrastructure to generate execution statistics for the comparison of FxP and FP implementations [13]. For energy estimation, I integrated McPAT [14] with Trimaran. All energy numbers are based on 32nm low-standby-power (LTSP) technology model. A summary of the baseline processor configuration is tabulated in Table 3.1.

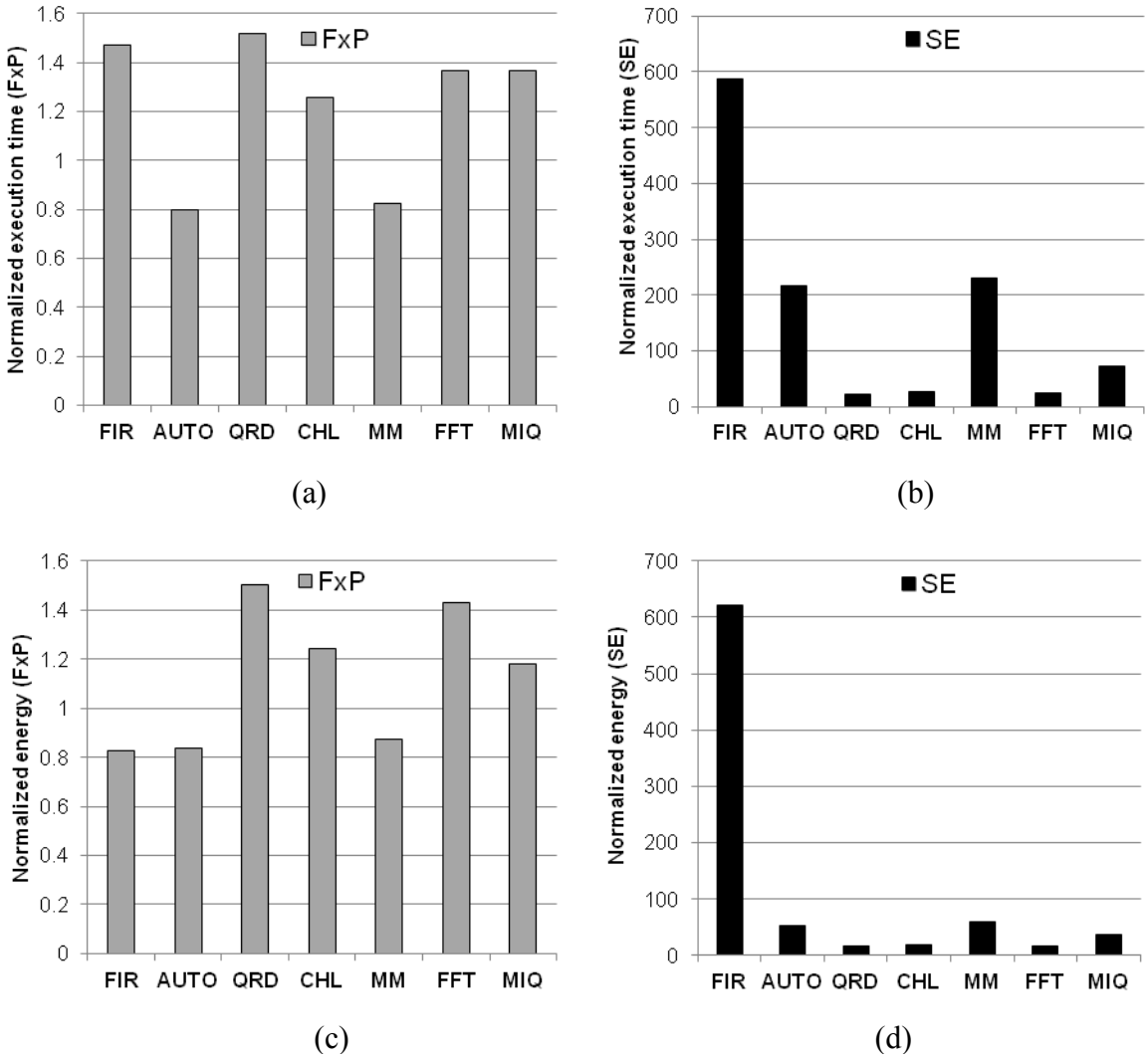


Figure 3.4: Normalized execution time and energy consumption of benchmarks using FxP and software emulated FP (SE). All results are normalized to those of the FP implementations. (a) Execution time (FxP), (b) Execution time (SE), (c) Energy consumption (FxP), (d) Energy consumption (SE)

The kernels used for evaluation of FP, software emulated FP (SE) and FxP arithmetic in this dissertation are listed in Table 3.2. These kernels are typically utilized in wireless communications, media processing and computer vision applications. FxP codes may require additional instructions to protect against overflows and underflows during computations and to performing

saturation or rounding. However, in order to limit the overhead of additional instructions, the FxP benchmarks developed for this research only include the scaling operations. This allows a conservative performance comparison of FxP implementation with FP and SE implementations.

3.4.2 PERFORMANCE AND ENERGY EVALUATION

Figure 3.4(a-d) show the execution time and energy consumption of the two alternatives to hardware support for FP arithmetic. Both execution times and energy consumption of FxP and SE implementations are normalized to the corresponding FP implementations. The FxP and SE are simulated for the baseline FxP processor while the FP implementations are simulated for the baseline FP processor. For kernels that have a relatively low instruction overhead for FxP scaling operations, such as AUTO and MM, the FxP implementation can result in about 20% reduction in execution time (compared to FP). This is because of the higher latency of FP operations. Both of these benchmarks involve a high percentage of chained multiply-accumulate operations that are impacted most by the FP operation latency.

The FxP implementation of the FIR benchmark exhibits considerably lower performance than the FP implementation. Although FIR also performs a significant number of chained multiply-accumulate operations, the high instruction level parallelism of the algorithm is able to hide their latency. The FxP implementation of FIR results in a performance reduction because of the additional scaling operations. Although, the benchmark requires only a few scaling operations, these operations add data dependencies in the code, resulting in longer instruction schedules being generated by the compiler. The relative energy cost of these scaling operations is lower than their relative performance impact. Consequently, the FIR benchmark shows a performance reduction

with its FxP implementation but an improvement in energy consumption due to the reduced energy consumption of the FxP operations.

The FxP implementations of matrix decomposition algorithms (i.e., QRD, CHL, and MIQ) and the FFT have considerably more scaling operations. Therefore, they exhibit better performance with FP arithmetic. Moreover, both QRD and CHL also require software routines for FxP square root operations, which significantly increase the execution times of their FxP implementations.

A key observation from Figure 3.4 is that the additional instructions in FxP benchmarks increase the energy consumption of all the pipeline stages (i.e., instruction fetch, decode, execution, and writeback stages). On the other hand, replacing FxP operations with equivalent FP operations mainly increases the energy in the execute stage. Moreover, for algorithms that experience a performance loss, the increased execution times also increase the leakage energy consumed by the entire processor.

From an energy consumption perspective, the additional instruction overhead in the more complex benchmarks, such as QRD, CHL and FFT, results in the FxP benchmarks using more energy. On the other hand, benchmarks that require relatively few scaling operations, like FIR, autocorrelation and matrix multiplication, experience an increase in energy with FP implementations due to the additional energy cost of FP operations.

The SE implementations result in a significant deterioration in performance and energy consumption. This is because unlike FxP, which introduces a few scaling operations in the code, SE implementations include time-consuming functions that implement FP arithmetic operations us-

ing bit-level operations. On average, the SE implementations can increase the execution time and energy consumption by more than 100X compared to FP implementations.

In terms of area, hardware support for FP arithmetic in both SIMD and scalar units requires 13% more area than the baseline FxP processor.

4. HYBRID BFP-FP FMA UNIT

In this chapter, I propose a novel FP fused multiply-add (FMA) unit design that exploits the characteristics of common operations in digital signal processing (DSP) algorithms to significantly improve performance and reduce energy consumption. The novel contributions presented in this chapter include:

i) Design and evaluation of a high-throughput FMA (HFMA) unit with block floating-point (BFP) support.

ii) Dynamic-range-based dynamic voltage and frequency scaling (DVFS) for DSP kernels with hard execution deadlines.

iii) Accuracy comparison of the proposed HFMA unit with a conventional FMA unit.

4.1 PERFORMANCE ISSUES OF FLOATING-POINT ARITHMETIC

As shown by the performance trends in Figure 3.4, FP implementations can result in performance degradation as high as 20% for kernels with a significant number of chained multiply-add operations. Figure 4.1(a) shows an example loop involving such chained multiply-accumulate operations. The loop multiplies each entry in the array *a* by itself, and adds all the products into the variable *sum*.

For a DSP processor using a conventional FMA unit (cf. Figure 4.2), the operations performed in this loop are scheduled as shown in Figure 4.1(b). The first part of the loop involves the multiplication of array elements. Since the multiplications of array elements do not have any data dependencies, the compiler schedules them using a single vector multiplication (*vec_mul*) in cycle 0. The vector operation is executed by the SIMD vector unit. Although the actual scheduled code

requires additional instructions to load data to vector register files, I do not show the additional instructions for clarity. Once the vector multiplication has written the results to the vector register file (cycle 2), the scalar FP units (FPUs) are used to add the elements in the product array. Due to the three-cycle latency of the FP addition operation, even when the multiplication is performed using the vector units, the scalar addition of the product vector can result in an overall latency of 24 cycles (7 FP additions x 3-cycle FP addition latency + 3-cycle vector FP multipli-

```
for (i=0;i<8;i++)
   sum += a[i]*a[i];
```

(a)

Cycle	Instruction
0	Vx_mul(a,a,prod)
1	
2	
3	add(prod[0],prod[1],sum)
4	
5	
6	add(sum,prod[2],sum)
7	
8	
…	…
…	…
21	add(sum,prod[7],sum)
22	
23	

(b)

Cycle	Instruction
0	mulAndAcc(a[0],a[0])
1	mulAndAcc(a[1],a[1])
2	mulAndAcc(a[2],a[2])
3	mulAndAcc(a[3],a[3])
4	mulAndAcc(a[4],a[4])
5	mulAndAcc(a[5],a[5])
6	mulAndAcc(a[6],a[6])
7	mulAndAcc(a[7],a[7])
8	
9	
10	normAcc(sum)
11	

Scheduling delay

(c)

Figure 4.1: Scheduling for chained multiply-accumulate operations. (a) Code snippet performing chained multiply-add operations, (b) Baseline instruction scheduling, (c) Instruction scheduling with the proposed HFMA unit

cation latency = 24 cycles). This is because for each FP addition operation, both operands must be available before the addition operation can be issued. This data dependency between addition operations causes scheduling stalls and a significant performance impact. Moreover, for an actual scheduled code, additional cycles are required for data transfers between the SIMD vector units and the scalar pipeline. These can further reduce the performance of chained multiply-accumulate operations.

To reduce the performance penalty due to dependent multiply-accumulate operations, I propose a high throughput FMA (HFMA) unit that exploits the limited dynamic range of many embedded applications. My approach allows the compiler to schedule dependent multiply-accumulate instructions back to back in successive clock cycles. This improves the FP pipeline utilization and thus the throughput of FP operations. The proposed design uses a single *block-exponent* for chained multiply-accumulate operations and avoids the normalization of intermediate results of FP multiply-accumulate after each addition/subtraction. *The key component that allows back-to-back operations is the pipeline flip-flops that store the unnormalized result. The multiplier results are aligned and added to the unnormalized result stored in the pipeline flip-flops.* Section 4.6 compares my approach with the standard BFP approach in more detail.

As shown in Figure 4.1(c), with my approach, each multiply-accumulate (mulAndACC) instruction needs only the two multiplication operands. Since the data-dependencies between two consecutive multiply-add operations are resolved in time, the multiply-accumulate instructions can be issued in consecutive cycles, keeping the multiplier pipeline completely utilized. Unlike the conventional design shown in Figure 4.2, the multiply-accumulate results are accumulated in the FPU. Once all the multiply-accumulate instructions have been issued (cycles 0-7), the final

instruction normalizes the accumulated result and converts it to the IEEE-754 single-precision (SP) FP format. With our approach all operations in the loop are scheduled on a scalar FP unit. The normalization instruction needs to wait for two cycles (cycles 8 and 9) to allow the last multiplication to be completed and added to the accumulator. The overall latency for the loop execution with our approach is 11 cycles.

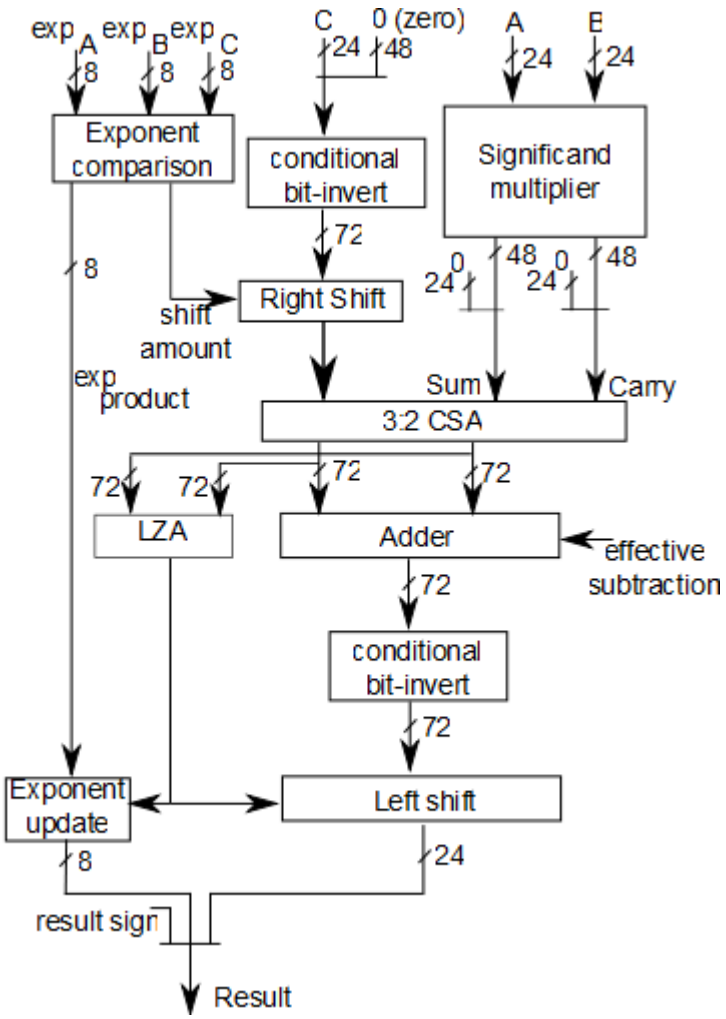


Figure 4.2: Baseline single-precision FMA unit [61]

4.2 BASELINE FMA DESIGN

To compare the proposed HFMA unit with a conventional FMA unit, I briefly describe my baseline FMA design [17]. Figure 4.2 shows the baseline single-precision FMA design for $A \times B + C$ operations. The design improves the accuracy of a multiply-add result by avoiding the rounding of the intermediate multiplication result. The significand of the operand C is right-shifted, added to the 48-bit significand product and saved in carry-save format. The carry-save

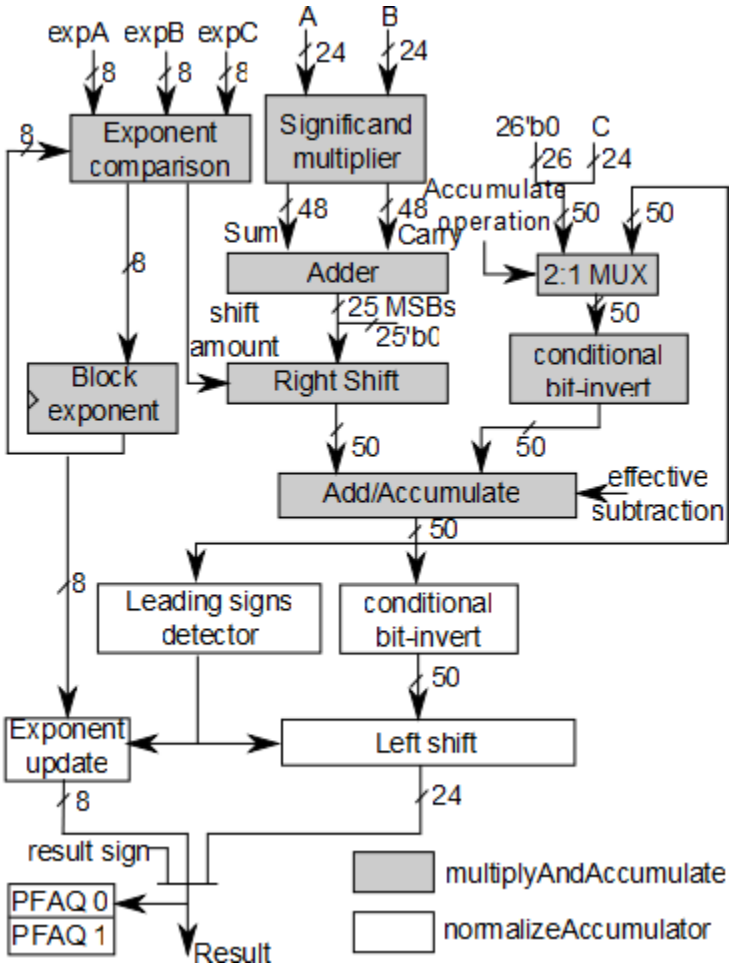


Figure 4.3: Proposed HFMA unit

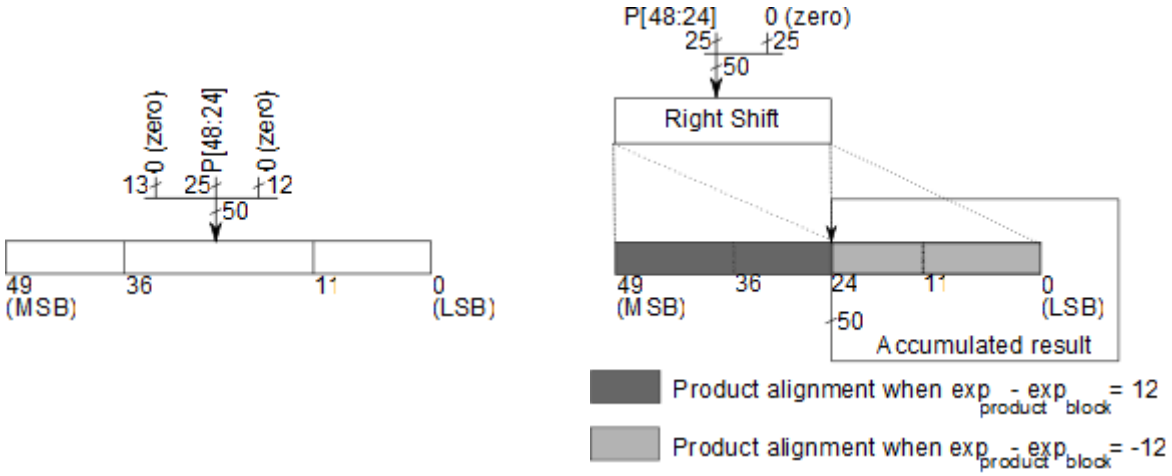
outputs of the *carry-save adder* (CSA) are then added by a compound adder. The adder inputs are also used by the *leading-zero anticipator* (LZA) to determine the left-shift amount for normalization. The normalization stage uses the LZA output to shift the result of the adder and may shift an additional place because the LZA output may be incorrect by one bit position.

The baseline design is optimized to reduce the latency of FP operations. This is achieved by utilizing the LZA to reduce the delay in determining the shift amount required for normalization and right-shifting the operand C in parallel with significand multiplication to shorten the critical path. Both the baseline and our proposed design implement only the round-to-zero rounding mode and flush all sub-normal FP numbers to zero.

4.3 PROPOSED HFMA DESIGN

Figure 4.3 shows the overall design of the proposed HFMA unit. Instead of minimizing delay, the design is optimized to maximize the throughput of the most common DSP operations, i.e., the chained multiply-accumulate operations. It achieves high throughput by internally providing BFP support for chained-multiply accumulate operation at a small area cost. All other FP operations are still supported with the same latencies as the baseline design.

The proposed design uses a 50-bit adder to add the 25-bit product and the 24-bit addend C for normal multiply-add operations ($A \times B + C$). In normal multiply-add operations, the addend C is aligned with the least significant 24 bits of the adder (Add/Accumulate in Figure 4.3(b)). The product is right-shifted such that it aligns with C according to the difference between the exponents of the product and the operand C. This requires a maximum shift of 49 bits. Instead of using the 48-bit result of product, the design uses the top 25 bits to reduce area, power, and delay.



(a) (b)

Figure 4.4: Alignment of product with accumulator for multiply-accumulate operations

This ensures the multiplication result is still precise up to the IEEE-754 single precision of 24 bits but reduces the width of the right-shifter and the datapath elements in later stages.

Chained multiply-accumulate operations use a single block-exponent for aligning the product and the accumulated result. This block-exponent can either be set by the programmer or by the exponent of the result of the first multiply operation in a chained multiply-add sequence. If the accumulator overflows, the block exponent is incremented and the accumulated result is right shifted by one.

For chained multiply-accumulate operations when using the exponent of the first product as the block exponent, the first product is aligned to the adder inputs [36:12], as shown in Figure 4.4(a). The block exponent is set to the exponent of the unnormalized product. For each successive multiplication after the first in chained multiply accumulate operations, the exponent of the product $A \times B$, ($\exp_{product} = \exp_A + \exp_B$), is compared with the block-exponent ($\exp_{block}$).

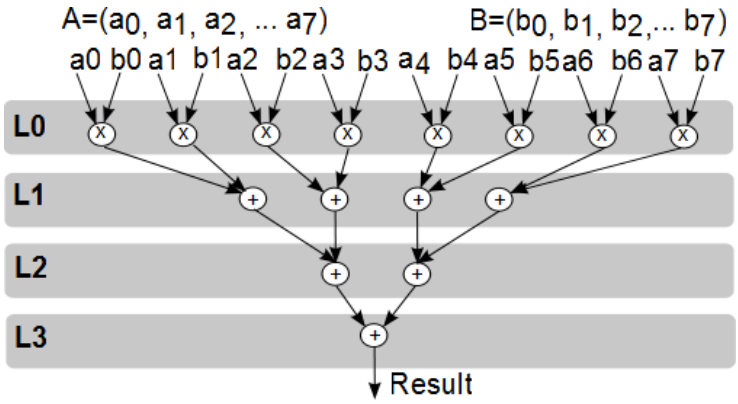
If $\exp_{prod}$ is within a range of ± 12 with respect to the block exponent (i.e.$|\exp_{product} - \exp_{block}| \leq 12$), the product can be added to the accumulator without losing any accuracy. As shown in Figure 4.4b, if $\exp_{product} - \exp_{\text{block}} = -12$, the product is right-shifted such that it aligns with the adder inputs [24:0]. For an exponent difference of +12, the product does not require any right shifts and uses the adder inputs [49:24].

If the absolute value of the exponent difference is greater than 12, the product cannot be added to the accumulator without losing MSBs or LSBs. In this case, the current result in the accumulator is allowed to proceed to the next stage in the HFMA for normalization. The block exponent is updated to the result of the product exponent. The result of the accumulator is normalized and stored in a 2-entry pending floating-point accumulate queue (PFAQ). If both entries in the PFAQ become valid, the processor is stalled and the two entries are added using normal floating-point addition. The result is again stored in the PFAQ and the other entry is invalidated. Once all the chained multiply-accumulate operations have completed, the result in the PFAQ is added to the final result in the accumulator after its normalization. These final additions proceed as normal FP add operations. Thus, only the sets of operands that have a large exponent difference with the accumulator exponent have to tolerate the cost of complete FP addition latency.

In the worst case scenario, all chained FP operations require the complete FP addition latency. However, in most practical situations, a relatively small number of FP accumulate opera-

Table 4.1: Synthesis results for proposed and baseline fused multiply-add units

Design	**Area (μm^2)**	**Power (mW)**
Baseline FMA	18394.0	9.625
Proposed HFMA	12315.6	5.610



(a)

Baseline	Cycle	Proposed
L0: FP multiply	0	L0: BFP multiply
	1	
	2	L1: BFP align & add
L1: FP add	3	L2: BFP add
	4	L3: BFP add
	5	L3: BFP md & norm.
L2: FP add	6	
	7	
	8	
L3: FP add	9	
	10	
	11	

(b)

Figure 4.5: Data-flow graph (DFG) and instruction scheduling of a vector dot-product instruction: (a) DFG, (b) Scheduling.

tions have a dynamic range larger than ± 12. Since the data dependent latency of operations cannot be determined during static scheduling at compile-time, the compiler assumes that no operation will require the additional stalls for static-scheduling. This allows a significant improvement in the FP multiply–accumulate performance for the common case.

I synthesized both a baseline FP FMA unit and our proposed HFMA unit using Synopsys® Design Compiler® and the TSMC 65-nm standard cell library at nominal operating conditions.

Table 4.1 shows the synthesis results for the designs along with power estimates obtained from Synopsys Power Compiler™. Both FMA designs were pipelined for a four-cycle latency at 1 GHz clock frequency.

The proposed approach strives to reduce the area by improving the throughput of FP operations instead of their latency. The area reduction is achieved by utilizing a leading-sign detector instead of the LZA, reducing the bit-width of the adder and left shifter to 50 bits (instead of 72 bits as in the baseline FP FMA) and removing the CSA from the design. Although, FP accumulators have been proposed by Vangal *et al.,*[18] and Luo *et al.,*[19], both involve considerably more logic to perform accumulation. *I reduce the overhead by only providing accumulation support for a limited dynamic-range of operands and reverting to standard floating-point fused multiply-add operations for high dynamic-range operands.*

4.4 DOT-PRODUCT INSTRUCTION

Many SIMD architectures support a dot-product instruction that computes the element-wise product of two vectors and adds the elements in the result vector to produce the accumulated product [46]. The proposed HFMA unit can also be employed to significantly reduce the latency of dot-product instructions. Figure 4.5(a) shows a data flow graph (DFG) of a vector dot-product instruction. The DFG consists of four levels indicated by L0, L1, L2 and L3. The two source vectors, A and B, are multiplied during L0 and the product vector is accumulated during L1, L2 and L3. Figure 4.5(b) shows the scheduling of dot-product instruction using a baseline vector FP unit composed of regular FMA units and the proposed vector FP unit composed of HFMA units. For the baseline case, each DFG level takes three cycles to complete. This is because each level

is dependent upon the results of previous level and both FP multiplies and adds takes 3 cycles. The overall latency of the vector dot-product instruction in the baseline case is 12 cycles (the result of the last FP add in Figure 4.5(b) is available after three cycles).

The scheduling of the dot-product instruction using our proposed HFMA unit is also shown in Figure 4.5(b). Overall, our approach can provide the final result of the dot-product in 7 cycles. This is achieved by relying on BFP-based operations and internally forwarding the results between different FP units without normalization and rounding. Specifically, instead of performing a FP multiply operation in cycle 0, the unit performs a BFP multiplication. This essentially means that the two significands are multiplied but the product is not normalized or rounded. In cycle 2, all the product significands are aligned to a single block-exponent. If the difference between the product exponents and the block-exponent is between [-12:12], the remaining steps in the DFG can be performed using BFP arithmetic. If the exponent difference is beyond this range, the dot-product operations are scheduled similar to the baseline case. However, for more than 95% of FP operations the unit can perform a low-latency BFP-based dot-product.

In cycles 3 and 4, the fixed-point (FxP) accumulated result is forwarded between the FMA units. The result continues to be accumulated without being converted to the standard IEEE-754 FP format. This allows the unit to perform the accumulation steps L1, L2 and L3 in three cycles (instead of the 9 cycles required in the baseline case). The adder widths in the FMA units are increased by one or two bits to ensure that there is not overflow during BFP additions. Once the accumulation steps have been completed, the final accumulated result is normalized and rounded in cycle 5 to convert it to the standard IEEE single-precision format. Overall, my approach produces the result of a dot-product in 7 cycles.

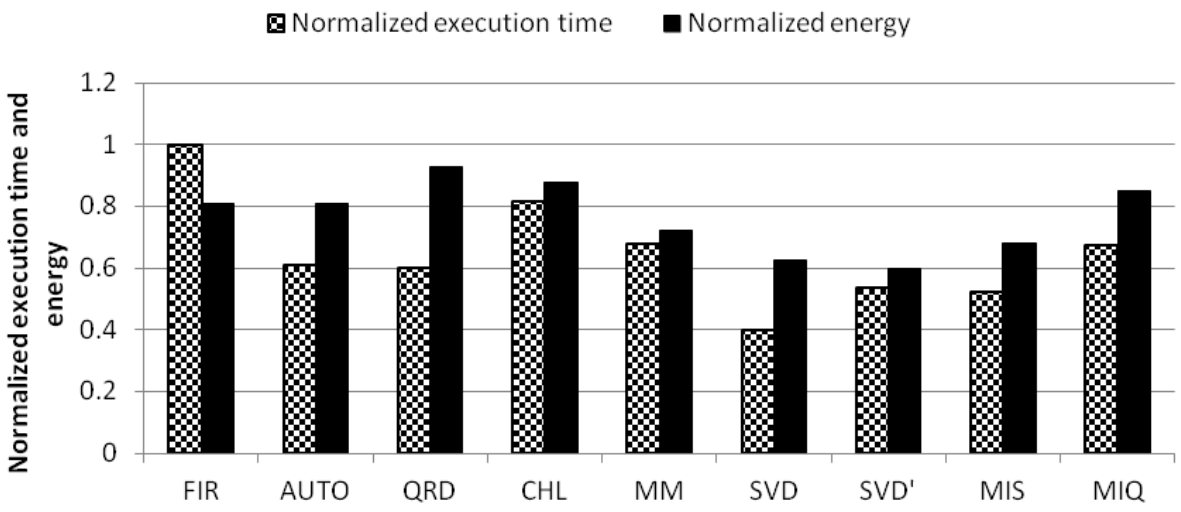


Figure 4.6: Performance and energy impact of proposed HFMA unit. The results are normalized to the performance and energy of the baseline floating-point unit that does not provide BFP support for accumulate operations.

4.5 EVALUATION

To investigate the performance and energy impact of our HFMA unit, I added *multiplyAndAccumulate*, and *normalizeAccumulator* instructions to the compiler and simulator infrastructure and modified the benchmarks to utilize these custom instructions. Figure 4.3 shows parts of the data path that are active during the execution of each of these instructions. I assume the baseline processor already employs our proposed low-power multiply-accumulate design. Thus, the only impact of our design is on the energy of chained multiply-accumulate operations. These operations utilize the BFP approach for accumulation. Thus, for these operations, only the final accumulation result will be normalized. Based on the synthesis and power estimation results, the accumulate operation will consume 17% less energy than our design's normal multiply-add operation.

Figure 4.6 shows the normalized execution time and energy consumption of the kernels with my proposed HFMA unit. The results are normalized to the execution time and energy of the FP unit that does not provide BFP support for accumulate operations. These results assume the input dynamic range of the algorithm was estimated beforehand and the programmer explicitly set the block-exponent during accumulate operations to avoid any cases that would require additional stalls. In practical situations, this is typically achieved by limiting the dynamic range of the input to ensure that the dynamic range remains limited during computations [20].

The performance of most benchmarks exhibits a significant improvement when BFP accumulation is supported in the FMA unit. The SVD benchmark exhibits the greatest performance improvement; nearly 60% shorter execution time. All kernels except FIR show significant performance improvements; the least performance improvement reduces execution time by 20% for Cholesky. Compared to FxP implementations, the execution time reduction can be as high as 60% (for QRD). The benchmarks that exhibit performance degradation with the baseline FMA unit also show more than 20% reduction in execution time compared to FxP implementations. The reason for the high speedups is that matrix-based kernels need to perform several operations such as column rotations and vector norms that require accumulation of the product of columns and rows of matrices. The FIR benchmark does not show any speedup because the FP FMA latency was effectively hidden through software pipelining in the baseline FP implementation.

All benchmarks show considerable energy reduction. This is because the accumulate operations do not utilize the normalization stage of the FMA unit resulting in energy savings. The energy reductions can be as high as 30% for kernels employing SVD. Although, the BFP accumulation approach only reduces energy by 17% per accumulate operation, the additional energy

savings come from reduced instruction counts, because the compiler requires fewer vector register move operations, as well as a significant reduction in memory operations to exchange data between the SIMD vector unit and the scalar register file. This is because most of the accumulate operations can now be scheduled on the scalar FP units (FPUs); in the baseline case the multiplication of operands was performed using the vector unit while the sequential additions were then performed using the scalar FPUs.

Compared to the FxP implementations, my approach results in a maximum energy reduction of 38% (for QRD). For FP implementations that exhibited a higher energy consumption than the FxP implementations, my approach results in energy reductions of 3% (for FIR and autocorr) and 18% (for matrix multiplication).

If the baseline FMA unit is modified to include a 32 × 32-bit multiplier for supporting 32-bit integer multiplication in the FMA unit, the overall area overhead over a FxP processor (with 32 × 32-bit multipliers) is 74%. This assumes the FPUs are used in both the scalar and vector pipelines. However, with hardware support for FP arithmetic, high dynamic-range numbers can utilize the FP format. Consequently, the integer data-path may be reduced to 24 or 16-bits. A 24 × 24-bit multiplier will suffice in this case. If the multiplier size is not increased in the baseline FMA, the overall area overhead reduces to 47%. *With our proposed HFMA, the area overheads are 27% and 13% respectively for the two cases (with a 32 x 32-bit multiplier and with a 24 x 24-bit multiplier)*.

4.6 DISCUSSION

For matrix-decomposition and inversion algorithms, the original matrix is updated during each

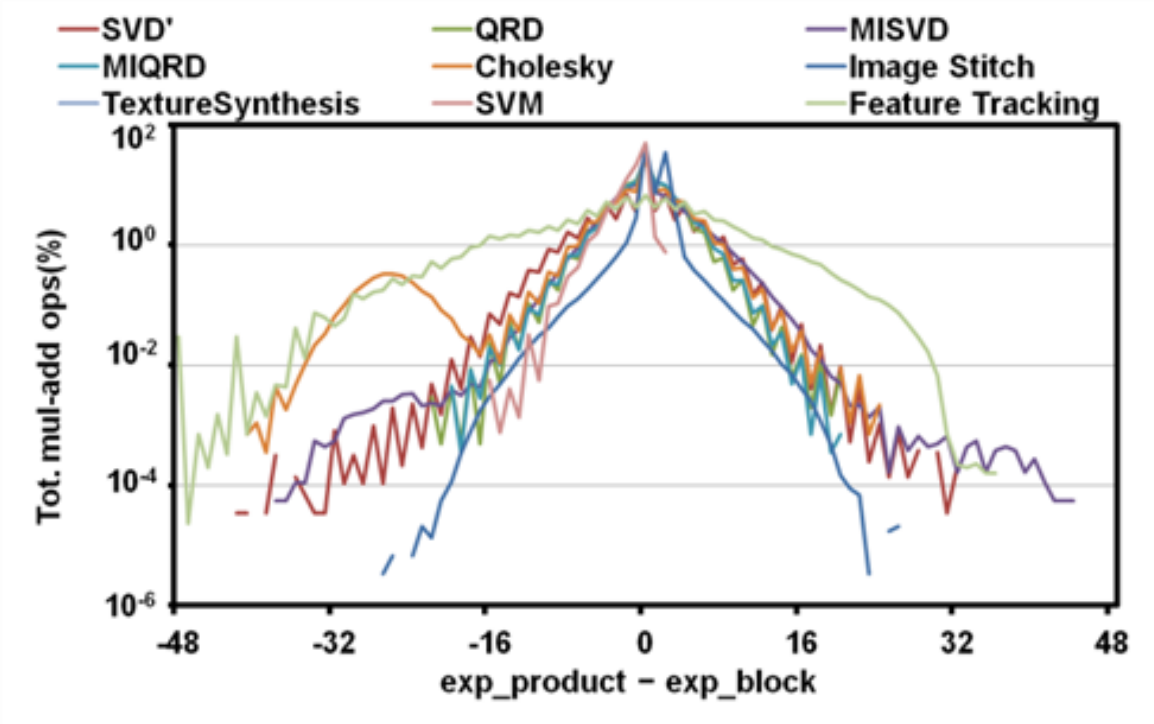


Figure 4.7: Exponent difference of multiplication result with the block-exponent used for accumulation

iteration. This can make it particularly difficult to estimate a block-exponent that is suitable for multiply-accumulate operations without specifically reading the entries in the rows or columns of the matrix before computations to determine a suitable scaling factor. Even in this case, there may not be a block-scaling factor suitable for all the computations, leading to a loss in precision. On the other hand, for DSP kernels such as the FIR filter and the FFT, estimating the dynamic ranges of operations during computations is considerable less complex.

Providing only BFP arithmetic support, as is done in the ARM-Ardbeg architecture [10], requires a significant programming effort to write algorithms that can efficiently utilize the BFP hardware. It is also likely to increase the instruction overhead due to the additional instructions required for comparing numbers for setting a proper block exponent and to deal with possible

Figure 4.8: Performance impact of stalls

overflows and underflows. For operations that cannot exploit the BFP support, the programmer still needs to resort to FxP arithmetic. Moreover, BFP can also lead to precision issues if a block–exponent is selected that results in a loss in precision of some operands in the same block.

Although my approach employs BFP to maximize the throughput of FP operations, it avoids the traditional limitations of BFP. The block-scaling factor can be selected by the hardware without requiring specific instructions to find a suitable scaling factor. The loss of precision of BFP is avoided by reverting to normal FP arithmetic during run-time. This also ensures that there is no increase in programming complexity due to implementing parts of an algorithm using FxP arithmetic or transforming the algorithms to make them more amenable to BFP arithmetic [11].

My approach provides BFP support for the most common FP operation in DSP kernels, thus achieving significant speedups. Due to the reduced data-path width and the unrounded result feedback for accumulate operations, my design relaxes some constraints specified by the IEEE-

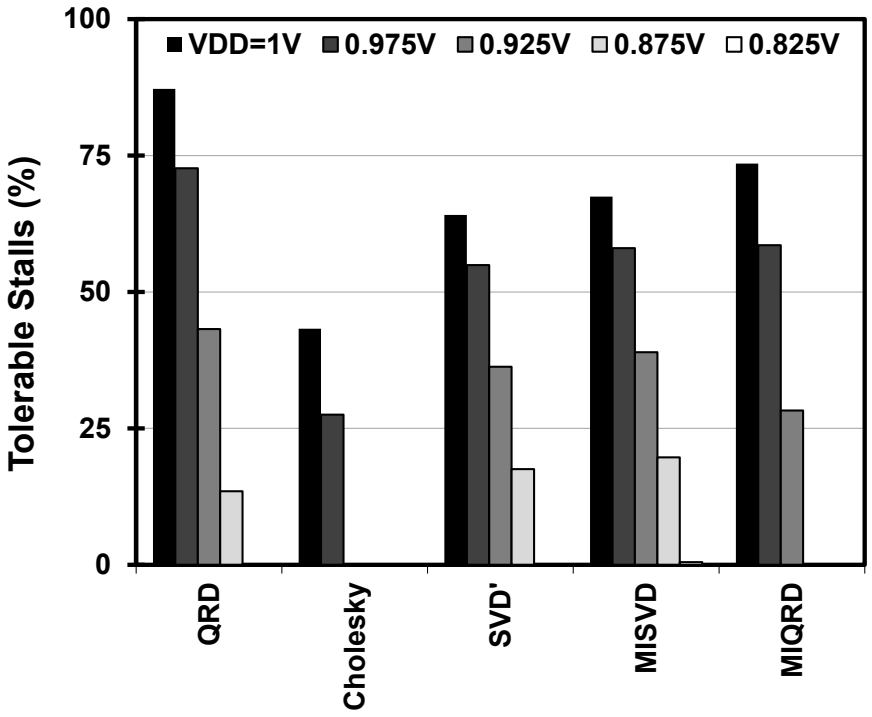


Figure 4.9: Operating voltage vs. % of execution stalls

754 standard. Specifically, the IEEE-754 standard requires unrounded multiplication results for higher accuracy in MAC operations; this additional accuracy is unlikely to be required in many embedded architectures.

To test the impact of dynamic range on the performance of kernels, I generated 1000 pseudo-random Rayleigh flat-fading matrices of dimensions 8 x 8. The Rayleigh flat-fading channel matrices are used to simulate MIMO channels in wireless communication simulations. The matrices were generated through MATLAB and are used as inputs to the matrix-decomposition and -inversion kernels.

Figure 4.7 shows the distribution of the difference in *product-exponent* ($\exp_{product} = \exp_A + \exp_B$) and accumulator block-exponent for all the multiply-accumulate operations encountered

during simulations. Typically, less than 1% of multiply-accumulate operations have operands with a high dynamic range ($|\exp_{product} - \exp_{block}| > 12$), thus requiring additional stall cycles. Since our multiply-add unit still allows normal FP multiply and add operations, operations other than multiply-accumulate are not impacted by our approach.

Figure 4.8 shows the performance impact of execution stalls due to high-dynamic range multiply-accumulate operations. In all cases, there is a very limited impact on the execution time of the kernel with my BFP-accumulation approach. However, DSP kernels sometimes have real-time deadlines that may be violated with the variable execution times. If hard deadlines are required, the performance improvement achieved with our approach can be converted to energy reduction through DVFS.

This allows the kernels to assume the worst-case execution time of the baseline FMA unit for the purpose of deterministic execution latency. The operating voltage can be adjusted according to the number of stalls caused by high dynamic-range operations. Specifically, if the number of stalls increases beyond a specified threshold, the operating voltage can be increased to ensure the real-time deadlines are still met.

Figure 4.9 shows that the execution time of the kernels increases as the operating voltage of the DSP decreases. Consequently, the number of stall cycles the kernel can tolerate while ensuring a worst-case execution time of the baseline FP fused multiply-add decreases. For SVD', MIS, and QRD -- even when the voltage is scaled to 0.875V -- more than 10% of the total multiply-accumulate operations can cause stalls without violating the real-time execution deadline.

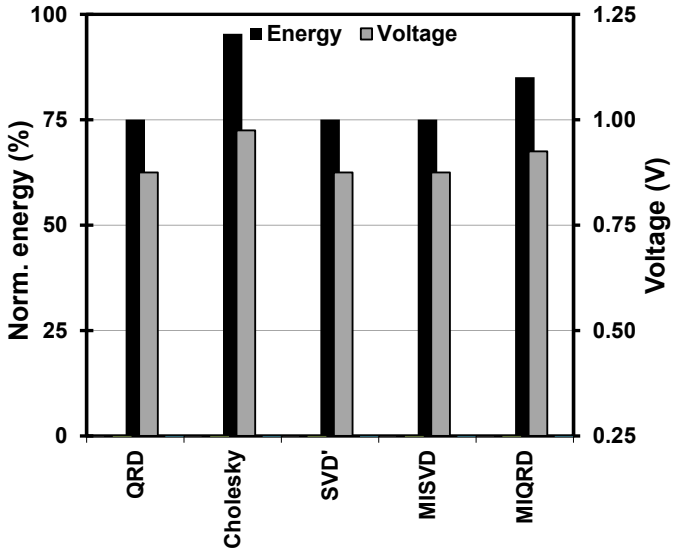


Figure 4.10: Energy reduction achieved through DVFS

However, the actual number of stall-causing accumulate operations experienced with our input data set of random Rayleigh matrices was less than .001% for these cases. This indicates that even if more operations have high dynamic-range operands, the real-time execution deadlines would still be met. However, in case more than 10% operations cause stalls for some matrix, the operating voltage can be dynamically increased to accommodate the increase in stall cycles.

Overall, significant power and energy savings can be achieved by dynamically scaling the voltage according to the number of stall-causing high-dynamic range accumulate operations. In most cases the core can be scaled to a considerably lower voltage without violating the worst-case execution latency of the baseline floating-point unit. Figure 4.10 shows the energy reduction achieved through dynamic range-based DVFS along with the operating voltage used for each benchmark. Energy reductions as high as 25% can be achieved while assuming more than 10% stall-causing high dynamic-range accumulate operations.

4.7 ACCURACY ANALYSIS

Since the proposed HFMA unit avoids rounding and normalization for a large percentage of dependent FP operations, the results produced by the HFMA unit are not bit-compatible with the results of the baseline FMA unit. Moreover, if the exponent range of FMA operands is greater than the exponent range covered by the HFMA unit, the results may also be less accurate. However, in this section it is shown that for real application data, very few operands suffer from this loss of accuracy. Moreover, this loss of accuracy can also be avoided by reverting to normal FMA operations for such operands. However, for most applications the loss of accuracy is not significant. In fact, by avoiding unnecessary roundings and normalizations, the HFMA unit typically produces more accurate results than the FMA unit.

In this section, I compare the accuracy of the results produced by the HFMA and baseline FMA units. For this accuracy comparison, I assume that both the baseline single-precision (SP) FMA unit and the SP HFMA unit use a 75-bit adder for accumulation (instead of the 50-bit adder utilized in Figure 4.3). The 50-bit adder employed in Figure 4.3 allowed the design to achieve higher energy efficiency (because of the narrower datapath) at the cost of reduced accuracy. This design was suitable for low-power embedded applications that do not require very high accuracy. For accuracy sensitive applications (e.g. scientific computing), the proposed HFMA unit can easily be extended to include a 75-bit adder instead of the 50-bit adder. An HFMA design that utilizes a 75-bit adder is described in Section 6.4.3. For accuracy comparison of double-precision (DP) operations, the internal modules of the FMA unit are proportionately scaled.

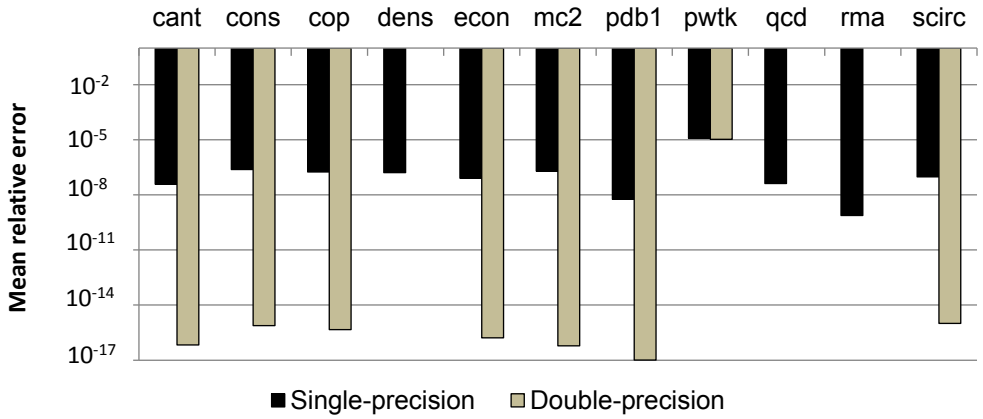


Figure 4.11: Mean relative error in the results using the proposed HFMA unit.

Figure 4.11 shows the mean relative error of HFMA results compared to IEEE-754 compliant FMA operations. Because scientific computing applications are likely to have a much wider dynamic range than image processing kernels, I utilized sparse matrix-vector multiplication kernels for accuracy analysis. The sparse matrices used for studying the accuracy impact are the same as used by Bell *et al.* [47]. The mean relative error in SP computations compares the results obtained using SP HFMA unit with the results of an SP FMA unit, while the error in DP results is determined by comparing the results of DP HFMA with the results obtained from a DP FMA unit. The results of SP and DP HFMA units are obtained using C code written for software emulation of the proposed unit. Although the HFMA approach results in some loss of accuracy for high dynamic range operations, the overall error in most cases is negligible.

For both SP and DP operations, the results achieved using our approach have a maximum mean relative error of the order of 10^{-5}. In all other cases, the mean relative error is even lower. In particular, with DP implementation, most kernels either do not result in any loss of accuracy (dens, qcd, rma) or the loss of accuracy is very low (mean relative error less than 1×10^{-15}).

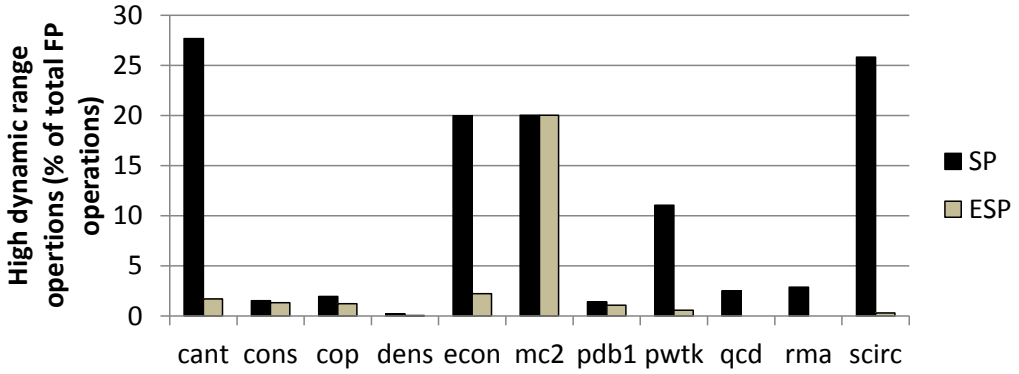


Figure 4.12: High dynamic range operations that may result in loss of accuracy (% of total FP operations)

The SP HFMA unit can also be enhanced to support DP operations (Section 6.5). For such an implementation the SP HFMA units are enhanced with wider shifters and adders to support the higher precision and dynamic range of DP operands. However, these wider resources can also be used to reduce the impact of high dynamic range operands. Specifically, the DP adders and shifters are required to be 163 bits wide while the SP adders and shifters are typically 75 bits wide. To extend the dynamic range coverage of SP operations in our approach, these wider shifters and adders can be utilized to implement extended single-precision (ESP) FMA/ADD operations. With 163 bits, our approach can cover a dynamic range of ±57 with respect to the block exponent. When ESP is utilized, the number of high dynamic range operations that can potentially reduce the accuracy of the results decreases greatly. Figure 4.12 shows the percentage of FP operations that resulted in a loss of accuracy using the SP operations and the ESP HFMA units. With the exception of mc2, for all kernels the percentage of SP FP operations that resulted in a possible loss of accuracy reduced to less than 2%.

The mean relative error shown in Figure 4.11 is primarily due to the bit-incompatibility of HFMA results with FMA results. Consequently, the percentage of high dynamic range operations per benchmark (Figure 4.12) does not have a strong correlation with its mean relative error. Although the HFMA results are not bit-compatible with FMA results, they are more accurate. To show this, I compared the accuracy of SP FMA and SP HFMA after executing multiple iterations of dependent FMA operations. The FMA dependence chains lengths in sparse-matrix vector multiplication benchmarks (Figure 4.11) are variable and do not clearly show the impact of error accumulation due to long dependence chains. To show this impact, I performed an experiment that compares the accuracy of HFMA and FMA results for different dependence chains lengths (iterations). The accuracy comparison is shown in Figure 4.13. The baseline FMA unit utilizes round-to-nearest (RN) rounding mode while the HFMA unit only supports round-to-zero (RZ)

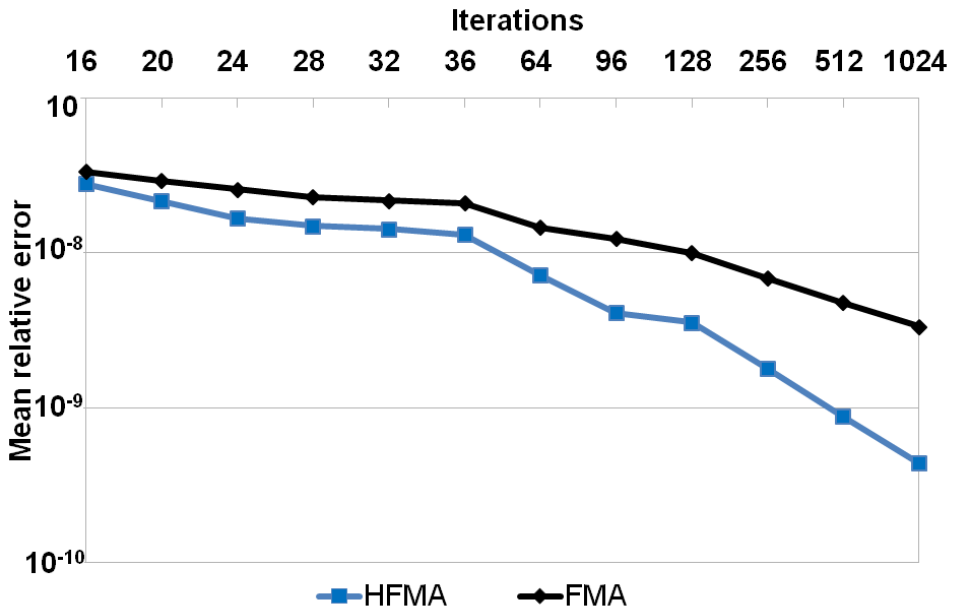


Figure 4.13: Accuracy comparison of SP FMA and SP HFMA implementations for different numbers of iterations

rounding. FMA operations are performed for multiple iterations and the results of our proposed HFMA units and the baseline FMA unit are compared with DP FMA results to determine the accuracy of the two approaches. The multiplication operands for FMA are generated pseudo-randomly (assuming a normal distribution with mean 0 and standard deviation 1) while the addition operand is the result of the previous FMA operation.

As the number of iterations is increased, the rounding error in the FMA result increases, resulting in a loss of accuracy in the results. Since the proposed HFMA unit, avoids unnecessary roundings and normalizations between dependent FMA operations, the results produces by our proposed HFMA unit are more accurate than the baseline FMA. The accuracy difference between the HFMA unit and the baseline FMA unit increases as the number of iterations are increased due to increased rounding error accumulation.

The proposed HFMA unit is able to achieve a higher accuracy while only employing the RZ mode rounding. Thus, the HFMA unit not only avoids the area, power and energy overheads of rounding and normalization, it also achieves a higher accuracy.

5. VIRTUAL FLOATING-POINT UNITS

Chapter 4 proposed a floating-point (FP) fused multiply-add (FMA) unit that allowed low latency execution of dependent multiply-add instructions. The approach utilized a hybrid block floating-point (BFP-FP) approach for such dependent operations. Instead of dedicated FP FMA units, the fixed-point (FxP) units in the processor can also be utilized to implement the hybrid BFP-FP approach for dependent multiply-add operations. Even without dedicated FP units the processor can achieve performance improvements for kernels that perform high percentages of multiply-add operations (e.g. matrix and vector operations). This allows the implementation of most common FP operations on the FxP units already present in the processor at a low performance and power cost. Many embedded applications, such as computer vision and signal processing algorithms, perform extensive matrix operations. Since dot-products are required for most matrix-based computations, I also propose a hybrid virtual BFP-FP approach for performing low-latency vector dot-product operations using the FxP single instruction multiple data (SIMD) vector units of the processor.

In this chapter, I propose techniques to provide architectural support for FP arithmetic that can be efficiently employed in low-power embedded processors. Unlike previous low-power FP approaches, such as light weight FP (LWFP) and block floating-point (BFP), our approach does not increase the programming complexity of the processors. Moreover, it can be applied to a broader set of applications than LWFP and BFP, which are limited to a specific precision (LWFP) or dynamic range (BFP). The novel contributions described in this chapter include:

i) Low-overhead support for FP arithmetic using virtual floating-point units (VFPUs) that con-

sume less chip area and power/energy than conventional FPUs. These VFPUs re-use the FxP execution resources of processors to provide architectural support for FP operations (Section 5.1).

ii) Performance and energy optimization of virtual FP (VFP) operations by dynamically switching between BFP and FP arithmetic based on the application characteristics at run-time (Section 5.3).

iii) Low-latency VFP dot-product instructions exploiting BFP arithmetic and internally forwarding the results accumulated by FxP units (Section 5.4).

In order to reduce the area and power overhead of supporting FP arithmetic, I propose to re-use the FxP units that are already in a processor. In my proposed approach, the FxP hardware resources of the processor are chained together to form a virtual FPU emulating an FP pipeline. My proposed approach breaks up a single FP operation into a sequence of FxP operations (micro-operations) that are performed using the FxP execution resources of the processor. The processor is enhanced with control logic and special modules to detect FP operations at the decode stage and orchestrate the data movement between FxP units to execute the micro-operations corresponding to the FP operation. Since the FxP units are also used by FxP instructions, the compiler is responsible for ensuring that these units are free when FP operations are scheduled to FxP units. I assume that support is added only for single-precision (32-bit) FP operations because these are more common in embedded systems, but the techniques can also be extended to double-precision operations.

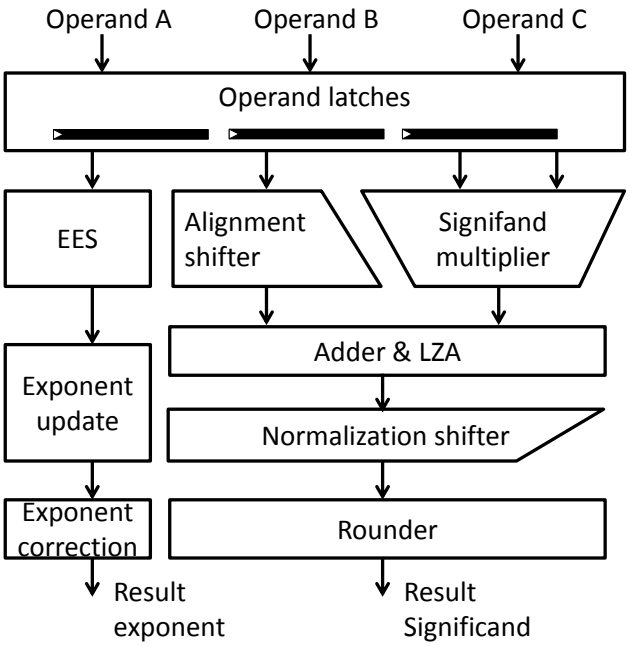


Figure 5.1: Typical FP FMA unit.

Figure 5.1 shows the data path of a typical FP FMA unit [48]. The significands of the two multiplication operands (*A* and *B*) are passed to the significand multiplier to compute the product. The EES (exponent comparison, sign and exception handling) module compares the exponents of the three operands to determine the product exponent and the alignment shift needed to add

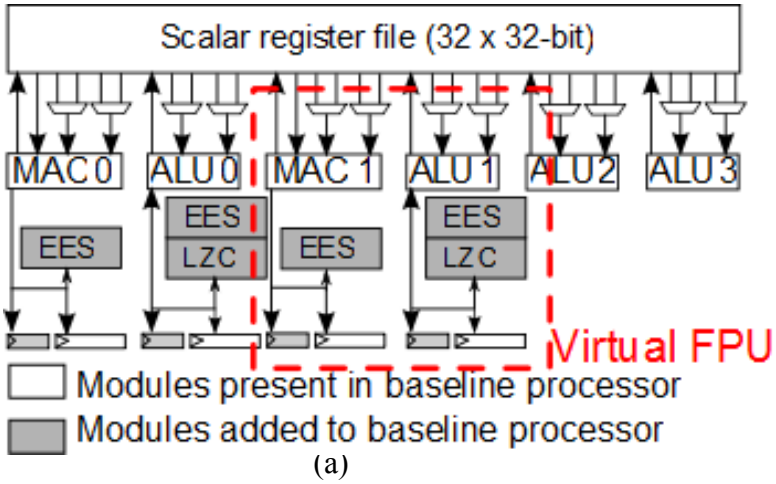
Table 5.1: Micro-operations and the corresponding resource used for VFMA operations.

Micro-operations	**Resource utilized**
Exponent/ Exception/Sign handling (EES)	Custom logic
Significand multiplier (SMUL)	FxP MAC
Alignment shifter (ASFT)	FxP Shifter
Adder (ADD)	FxP ALU
Leading zero count (LZC)	Custom logic
Normalization shifter (NSFT)	FxP Shifter
Exponent update (EU)	Custom logic
Rounder (RND)	FxP ALU
Exponent correction (EC)	Custom logic

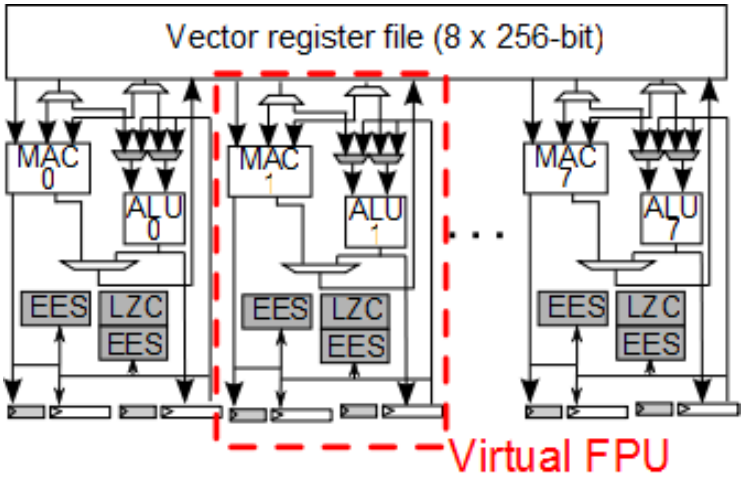
the addend operand (*C*) to the product. It also determines the effective operation type (add or subtract) according to the sign bits of the operands and checks for exceptions. The alignment shift of *C* is performed in parallel with the multiplication.

The product and the aligned *C* are utilized in the "Adder & LZA" block to compute the un-rounded result of $A \times B + C$. The LZA unit performs leading zero anticipation to estimate the left-shift amount required to normalize the result. In the next step, the left shift for normalization is performed and the exponent is updated based on the shift amount. In the final stage, the result is rounded. The FMA unit is also capable of performing FP fused multiply-subtract, FP addition (ADD), subtraction (SUB), and multiplication (MUL) operations.

Table 5.1 shows the different micro-operations that a virtual FMA (VFMA) operation is decomposed into for execution on FxP resources. The resource required for each micro-operation is also shown. While certain FxP resources, such as the multiplier, ALU and shifters, can be re-used for some micro-operations, other parts of the FP operations, such as the leading zero anticipation (LZA), exponent update (EU), effective operation and sign handling (SH), and exception handling (EXH) may benefit from custom logic. Although these micro-operations can be scheduled to use existing FxP units, they are likely to take several execution cycles resulting in considerable performance reduction. However, the area overhead of supporting these micro-operations using custom logic is negligible while significant performance improvement can be achieved by a single-cycle execution of these operations using dedicated hardware. Other FP operations such multiply-sub, add, and multiply can be implemented using a subset of these micro-operations.



(a)



(b)

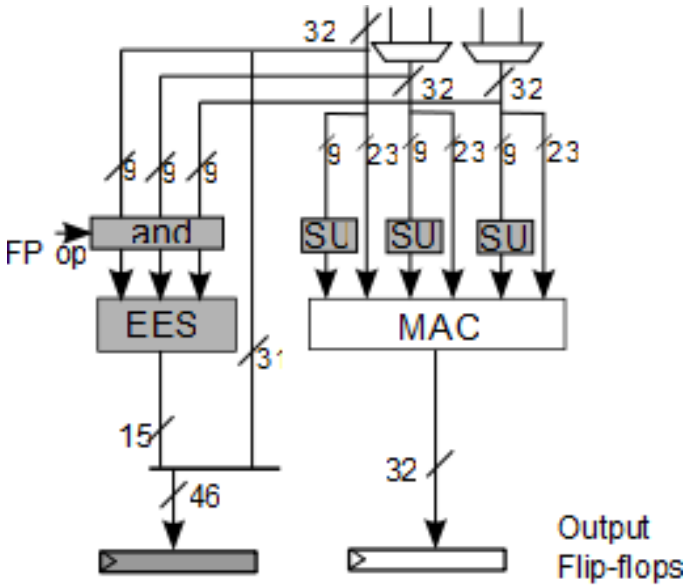
Figure 5.2: Baseline FxP processor (a) scalar and (b) vector units with the additional modules for FP operations.

For most embedded applications, round-to-zero (RZ) rounding mode can be employed without a significant impact on the SNR of the applications results [30]. For VFP operations, the RND and EU micro-operations (shown in Table 5.1) can be eliminated if only the RZ mode rounding is employed. For the remainder of the paper, I assume that the VFP operations only implement the RZ mode rounding. Moreover, instead of the LZA, I assume the design implements custom logic for a leading zero counter (LZC) module. The LZC has lower area and power consumption

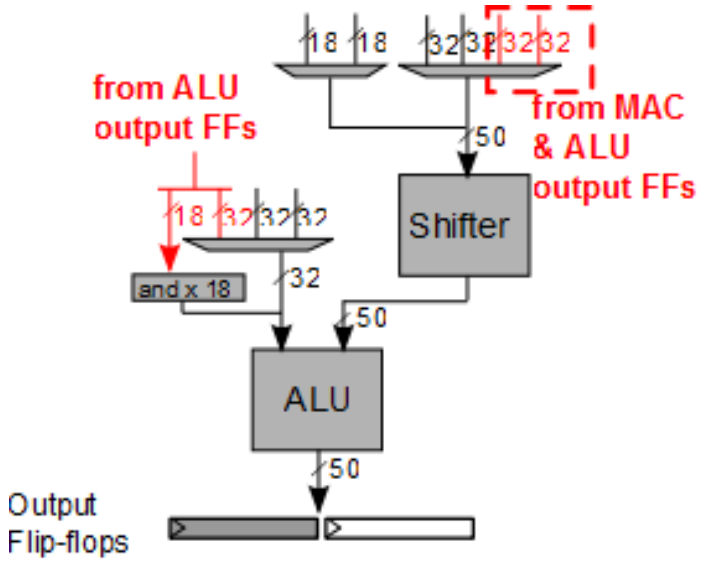
than an LZA, but increases the latency of VFP operations by one cycle. The design only provides architectural support for single-precision FP operations. Moreover, all subnormals are flushed to zero, similar to [12].

Figure 5.2(a) and (b) show the baseline FxP processor architecture along with the additional modules added to support virtual FP (VFP) scalar and vector operations, respectively. For both the scalar and vector pipelines, each FxP MAC unit is augmented with SH, EXH and EU modules to allow FP MUL and FP fused multiply-add (FMA) operations to perform sign-handling, exception checking, and exponent update in parallel with the significand multiplication. The three modules are collectively shown as the EES module in Figure 5.2(a) and (b) for clarity.

EES and LZC modules are added to two of the four available scalar FxP ALUs. This allows the processor to execute two scalar FMA operations in parallel, similar to the baseline FP processor. However, unlike the baseline FP processor, the proposed VFPUs re-use the same ALU for ASFT, ADD and NSFT micro-operations and thus cannot issue FMA operations every cycle due to structural hazards. Extra structural hazards could arise from the use of the same FxP units for both FxP and FP operations. However, these are avoided by resource-aware static instruction scheduling by the compiler in this paper. For most kernels, the performance loss due to reduced throughput of FP operations is fairly small compared to the associated area and power reductions of the processor. This is because most FP kernels involve dependent FMA chains where the dependent FMA operation cannot issue until the previous operation has completed. Moreover, I will also propose an approach that eliminates the structural hazards for most FP operations and provides a significant performance improvement for dependent FMA operations. All vector MACs and ALUs are also enhanced with custom modules to perform VFP operations, similar to



(a)



(b)

Figure 5.3: FxP arithmetic unit modifications. (a) FxP MAC unit, (b) FxP ALU.

the scalar resources (Figure 5.2 (a)).

Compared to software emulation of FP operations, my approach provides significantly higher performance and energy efficiency. This is because the software emulation of FP operations requires many bit-level manipulations, which are implemented much more efficiently in hardware than in software. Moreover, the use of custom modules further improves the performance of VFPUs compared to software emulation. Finally, the data dependent branches present in the software routines used for emulation of FP arithmetic also result in non-deterministic latencies for FP code and make the code unsuitable for parallel implementation on SIMD vector units.

5.1 IMPLEMENTATION DETAILS

5.1.1 FIXED-POINT EXECUTION UNIT MODIFICATIONS

Figure 5.3 shows how the modifications illustrated in Figure 5.2(b) and (c) are incorporated with the existing FxP MAC units and ALUs. For the MAC unit (Figure 5.3(a)), the FP instructions only require a 24-bit×24-bit multiplier, while the normal FxP MAC performs 32-bit×32-bit multiplication. The FP operands read from the register file contain 23 bits of significand. For FP operations, the significand update unit (SU) sets the hidden bit of the FP operands to one and the eight most significant bits of MAC's inputs (corresponding to FP exponent and sign bits) to zero. The bits representing the sign and exponents of the FP numbers are used by the EES to (1) determine the type of operation (addition or subtraction) according to the signs of the numbers, (2) handle special cases, such as NaN and Inf (exponent = FF_H), (3) determine if any operand is a subnormal and flush it to zero (exponent = 00_H), and (4) add the exponents of the multiplication operands. The inputs to the EES module are set to zero for non-FP operations to avoid unnecessary switching activity that can result in higher power consumption.

For the 2-stage FxP MAC unit, additional pipeline flip-flops are added at each stage to accommodate intermediate results for FP operations. Specifically, my approach requires an additional set of 47 flip-flops (8-bit result exponent + 1-bit result sign + 1-bit operation (add/subtract) + 8-bit addend exponent + 24-bit addend significand + 5-bit alignment shift amount) at each stage of the MAC pipeline for intermediate FP results.

The baseline processor employs an ALU with an inline shifter, similar to ARM processors [46]. Each ALU operand can be read from one of two read ports of the scalar register file. This requires a 32-bit 2-to-1 multiplexer (MUX) for the two operands. For the baseline vector pipeline, each operand can only be read from a single dedicated read port. Note that the baseline architecture thus does not require any MUX for the operands of vector operations.

The modifications required for the shifters and ALUs are illustrated in Figure 5.3(b). In order to maintain IEEE-precision in the results, the bit width of each adder and shifter is increased from 32 bits to 50 bits. I synthesized the inline shifter and ALU with bit-widths of 32 and 50 bits to see the impact of increased bit-width on area and power consumption. Both bit-widths were synthesized at the same baseline clock frequency of 1GHz. My results show an increase in area and power of 13% and 6%, respectively, when adder and shifter bit-widths are increased from 32 to 50 bits.

The intermediate results of micro-operations need to be read by the ALU. These intermediate results can either be read from the output flip-flops of the MAC unit or the ALU. For the scalar pipeline, a 32-bit 4-to-1 MUX is required at the shifter input instead of a 32-bit 2-to-1 MUX to allow operands to be read from the pipeline flip-flops.

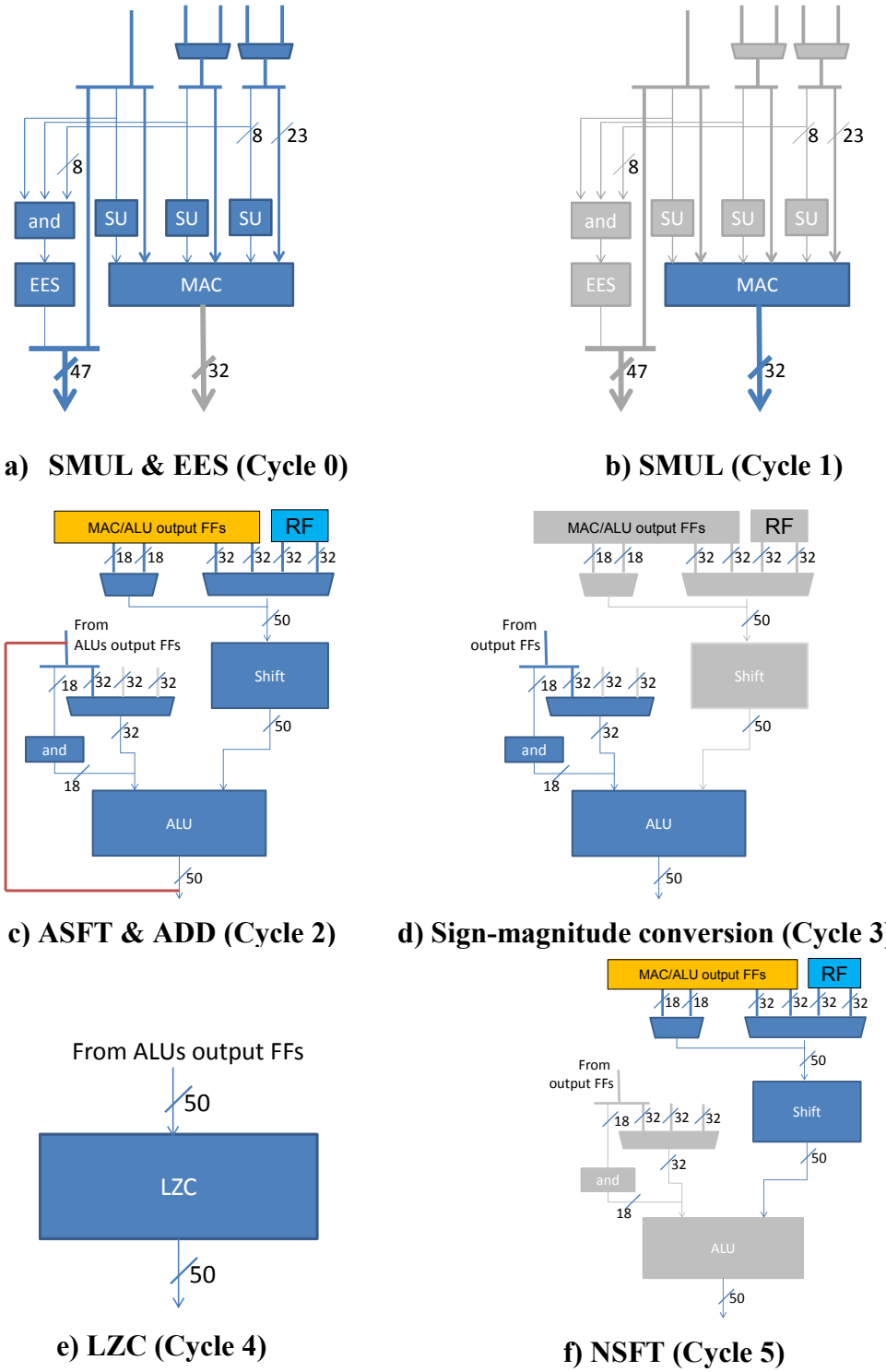


Figure 5.4: Scheduling of micro-operations for the VFMA operations on FxP units and custom logic.

The 18 most significant bits (MSBs), *i.e.* bits 49 to 32 of the shifter input, can only be read from the MAC or ALU outputs and thus require 2-to-1 MUXes. These MUXes are also used for operand isolation (for FxP operations) to prevent unnecessary switching activity in the logic [49]. Finally, an additional set of 27 flip-flops (8-bit result exponent + 1-bit result sign + 18-bit MSBs of addition result) is added to the output of the ALU to accommodate the intermediate results of FP operations.

The 32 LSBs of the addition result are stored in output flip-flops of the ALU that are also used for FxP operations. FP ADD and SUB instructions are issued directly to the ALU and do not use the multiplier. Thus, the exponent comparison required to determine the shift amount for the alignment of operands is performed within the ALU. The shifter in the ALU is enhanced to include an EES module to determine the shift amount for FP ADD and SUB instructions.

5.2 MICRO-OPERATION SCHEDULING

The scheduling of virtual FP micro-operations is performed using a combined hardware/compiler approach. Specifically, the compiler is responsible for preventing structural hazards during the execution of micro-operations. It ensures that the resources that would be required by the FP micro-operations are not scheduled for any other instruction during the cycles that the resources will be in use by the micro-operations. The compiler thus performs scheduling according to the resource requirements of micro-operations every time an FP instruction is encountered in the code. *The micro-operations are, however, not included in the compiled code.* When an FP instruction is decoded during execution, the control logic added to the processor ensures that appropriate MUX selection inputs are generated to allow pipelined processing of mi-

cro-operations. Although the compiler could add micro-operations in the instruction sequence, this would increase the energy consumption and reduce the performance of the code due to the need to fetch and decode multiple instructions for a single FP operation.

Figure 5.4 shows the scheduling of VFMA micro-operations on FxP execution units and custom logic. The resource used and the data-path within the resource are high-lighted in black. The execution resources that are not actively involved in a given cycle are grayed-out. The cycle-by-cycle VFMA micro-operations are:

Cycle 0: The operands are read from the scalar/vector register file. As shown in Figure 5.4(a), the *FPop* signal is raised once the operation is decoded as an FP operation. The SU modules set the hidden bit in the FP operands and clears bits 24 to 31 of the MAC's input operands (corresponding to the exponent and sign bits). The exponent and sign bits are sent to EES for exception checking, sign handling, and exponent update. EES also generates the ASFT shift amount (difference between product exponent and addend exponent). At the end of cycle 0, the intermediate result is written to the pipeline flip-flops. The intermediate result contains 47 bits (8-bit result exponent + 1-bit result sign + 1-bit operation (ADD/SUB) + 8-bit addend exponent + 24-bit addend significand + 5-bit alignment shift amount). If any of the operands is subnormal, the appropriate operand (product and/or the addend significand) is flushed to zero. Since the FxP multiplication takes two cycles, the multiplication result is not available until the end of cycle 1.

Cycle 1: The significand multiplication is completed as shown in Figure 5.4(b). The 25 MSBs of the product are stored in the MAC unit's output pipeline flip-flops. The 25-bit product ensures that the result is precise up to the IEEE-754 specified precision of 24 bits for the single precision significand after a one-bit normalization shift.

Cycle 2: The shifter gets the operands from the MAC unit's output flip-flops and performs an alignment right-shift according to the shift amount computed by EES in cycle 0. The shifted result is forwarded to the ALU to perform addition. At the end of cycle 2, 59 bits (8-bit result exponent + 1-bit result sign + 50-bit addition result) are written to the ALU's output flip-flops.

Cycle 3: The ALUs output is forwarded back to its input and the MSB of the addition result is checked to see if it is negative. If the addition result is negative the ALU is used to convert it to sign-magnitude form. This requires a bit-inversion and an addition operation. The ALU result (the sign-magnitude form of the addition result) is stored in the ALU output flip-flops. The process of BFP-FP accumulation is described in Section 4.

Cycle 4: the sign-magnitude form of the result, stored in cycle 3, is read and is used by the LZC to determine the left shift amount required for normalization of the result. The 6-bit shift amount determined by the LZC is stored back in the ALU output flip-flops along with the 50-bit addition result, the 8-bit result exponent, and the 1-bit result sign.

Cycle 5: The shift amount, stored in cycle 4, is once again forwarded back to the ALU to left shift the result of the addition for normalization. The EES module in the shifter is used to correct the result exponent according to the shift amount. The final result is stored in the compiler-assigned scalar/vector destination register.

VFP MUL and VFP ADD/SUB operations are scheduled similar to the FMA operation. Specifically, VFP MUL operations schedule the micro-operations shown in cycles 0, 1, 2, 4 and 5. The VFP ADD/SUB schedules the micro-operations shown in cycles 2, 3, 4 and 5. However, VFP ADD/SUB requires an additional ALU access for the EES unit to determine the alignment shift amount required for addition. The VFP MUL and VFP ADD operations thus take 5 cycles

each. Cycles 2, 3 and 5 use the same ALU for executing the micro-operations. Although this limits the throughput of VFP operations (since FP operations cannot be scheduled every cycle due to a structural hazard in the ALU), in Section 5.3, I present an approach that significantly reduces the throughput loss due to this hazard.

Because some FxP resources are scheduled for both FxP and VFP instructions, resource contention can result in a performance reduction. Since there are four ALUs in the scalar pipeline, the two ALUs that have been enhanced to perform VFP operations are only used for FxP instructions if both of the other ALUs are busy in the current cycle. This prioritizes the FxP instructions to execute on the ordinary ALUs and allows the VFP instructions to use the enhanced ALUs with reduced resource contention.

The *FP dot-product* instruction computes the product of two source vectors and adds the elements in the product vector using a binary tree with three addition stages. The baseline FP SIMD unit takes 12 cycles to compute the dot-product of two 8-element vectors. Because my VFP MUL and VFP ADD operations take 4 and 5 cycles respectively, the latency of a VFP dot-product is 19 cycles (vector multiplication (4 cycles) and 3 addition stages (5 cycle each)).

My synthesis and simulation results show that the VFPUs can reduce the area and power consumption of the processor by 24% and 31%, respectively. However, it can increase the energy consumption and execution time of some kernels due to higher operation latencies and structural hazards. VFP MUL, ADD, and SUB operations produce results with the same accuracy as an IEEE-754 compliant FMA unit with RZ rounding (for normalized FP numbers only, subnormals are flushed to zero). The accuracy of FMA operations is, however, less since the design only utilizes the 25 MSBs of the product. The accuracy for VFMA operations is the same as

the case where FP MUL and FP ADD are performed separately (instead of with an FMA) with the results of FP MUL and FP ADD rounded to zero.

5.3 HYBRID VIRTUAL BFP-FP ACCUMULATION

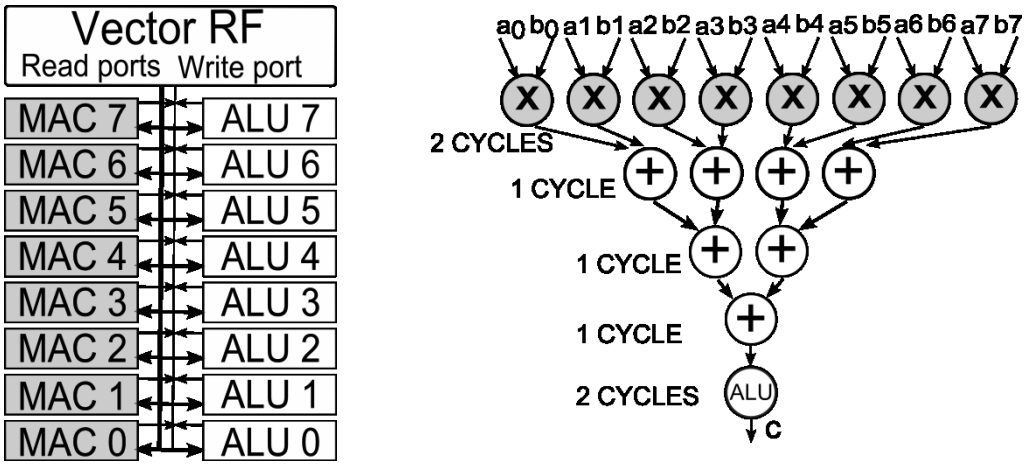
The primary reasons for the reduction in performance and energy efficiency of VFPUs compared to hardware FPUs are the increases in the operation latencies and the throughput reduction due to structural hazards that arise from the re-use of resources for VFP operations. In this section, I propose techniques to reduce the structural hazards and improve the performance and energy consumption of VFPUs.

FP intensive embedded applications typically perform a significant number of dependent FMA operations. The long latency of dependent VFMA operations coupled with resource constraints reduces the throughput of dependent VFMA operations. Thus, I utilize a BFP-based accumulation approach (Chapter 4) that alleviates the VFMA resource congestion and reduces the effective latency of dependent VFMA operations.

Since our VFP design increases the bit-widths of the ALU and the shifter to accommodate 50-bit intermediate operands in VFP operations, the design can use the 50-bit resources to employ a BFP-VFP approach for dependent VFMA operations. *Specifically, I propose to decouple the alignment right shift from the exponent of the accumulated result by aligning all operands to a single block exponent*. This is similar to the HFMA techniques proposed in Chapter 4.Moreover, the accumulated result (the unnormalized and unrounded adder result) stored in the pipeline flip-flops is read by the dependent VFMA instruction in the next cycle. This allows dependent VFMA operations to be issued in successive cycles because they do not need to wait for the ac-

cumulated result to be aligned. The block exponent for a series of dependent VFMA operations is set equal to the product exponent of the first VFMA operation by the hardware and is used to determine the alignment shift amount in cycle 0.

The 50-bit shifter and adder allow us to accumulate products with exponent differences in the range [-12, 12] with respect to the block exponent without requiring normalization of the intermediate addition results. This is because the 25-bit product can be aligned to bits [36:12] of the adder leaving 12 bits each in the most significant and least significant parts of the 50-bit accumulator for misaligned VFMA operands and one bit for the sign. The exponent difference is determined in cycle 0 by the EES module. If the exponent difference in a dependent BFP-VFMA operation is outside the range [-12,12], the processor is stalled and the previous accumulated result is allowed to proceed to normalization. *Essentially,* my *approach speculatively assumes that product exponents of dependent BFP-VFMAs will differ from the block exponent by no more than ±12 and stalls the processors if this condition is violated to revert to normal VFMA for the violating operation to ensure high accuracy*. This approach reduces the energy and performance cost from normalizing and rounding intermediate results and results in higher-throughput VFP operations. The high throughput is achieved because during BFP-VFMA operations, the ALU is required only once. The ALU accesses for cycles 2, 4 and 5 in Figure 5.4 are eliminated until the entire accumulation has been completed and the final accumulated result is normalized and rounded.



(a) (b)

Figure 5.5: (a) Vector FxP unit, (b) Scheduling of a dot-product instruction with hybrid BFP-FP accumulation.

5.3.1 OPERATION SCHEDULING:

For my hybrid BFP-VFP based implementation, cycles 0 to 2 are executed similar to the VFMA instruction. The accumulated result produced at the end of cycle 2 is read by the dependent VFMA operation in the next cycle. This is highlighted in Figure 5.4(c). The sign-magnitude conversion, LZC and the normalization steps (cycles 3, 4 and 5) are executed once the entire accumulation is completed.

High dynamic range operations ($|\exp_{product} - \exp_{block}| > 12$) are determined during the first cycle of multiplication by the EES module. For such operations, the current product cannot be accumulated without causing a loss of precision. In this case, the processor is stalled and the current accumulated value is normalized. The normalized result is then added to the high dynamic range operand and the unnormalized and unrounded result of their addition is used to reset the block exponent and the accumulated value.

5.4 HYBRID VIRTUAL BFP-FP VECTOR DOT-PRODUCT

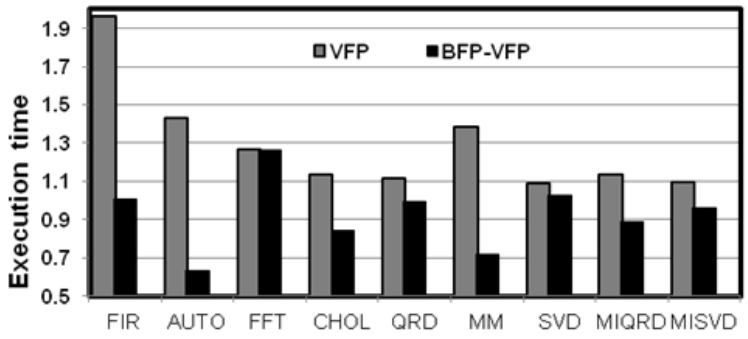
I also reduce the latency of virtual vector dot-product instructions by exploiting the hybrid BFP-VFP accumulation approach. Specifically, the design performs the addition of elements using BFP arithmetic to considerably reduce the overall latency and the resource congestion that arise from ALU re-use.

Figure 5.5(a) shows the organization of the FxP vector unit employed in the baseline processor. The vector unit consists of 8 FxP MACs and 8 ALUs. One MAC or one ALU instruction can be issued to the vector unit in a cycle. The MACs and ALUs share register file read/write ports. Figure 5.5(b) shows the scheduling of micro-operations for the VFP dot-product instruction using the FxP vector unit's resources. Element-wise multiplication is performed in the first two cycles using the MAC units in the vector unit. The product is written to output flip-flops and read in the subsequent cycle by the ALUs. The ALUs perform the accumulation of the product vector. However, since the accumulation in our BFP-based approach is performed as a fixed-point addition, the ALUs in the vector unit internally forward the data to perform each stage of addition in the tree in a single cycle (Figure 5.5). Specifically, once the MAC units have computed the product vector, ALUs 0 to 3 read the product elements and perform alignment and addition. The alignment is performed according to a single block-exponent for all the elements of the product vector. The addition result is forwarded to ALUs 4 and 5 to perform the second stage of addition. Their results are passed to ALU 6 which performs the final addition. In the next 2 cycles, ALU 7 performs normalization and rounding of the result. In order to avoid overflow during additions, the ALU bit widths of ALUs 4 and 5 are increased by one bit and that of ALU6 is increased by two bits.

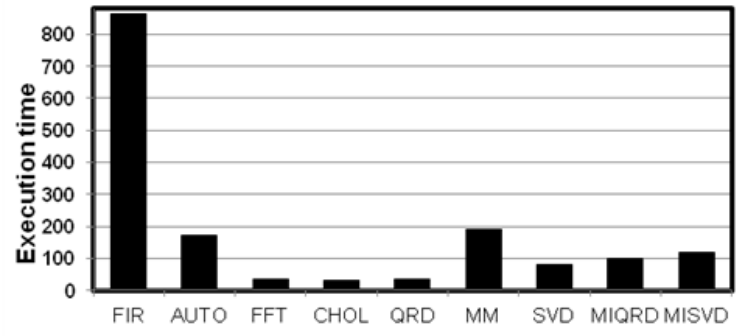
Table 5.2: Area and power overhead of VFP relative to FxP.

	Area overhead	Power overhead
Fixed-point processor	0.0%	0.0%
Floating-point processor	29.1%	33.5%
Virtual FP processor	5.6%	2.8%
EES & LZC modules	*1.0%*	*0.6%*
Pipeline flip-flops	*2.2%*	*0.5%*
50-bit ALUs	*1.4%*	*0.8%*
MUXes/Isolation gates	*1.0%*	*0.9%*

The VBFP-based dot-product scheduling reduces the latency of an eight-element dot-product instruction to 7 cycles compared to the hardware FP's latency of 12 cycles. The case of high dy-



(a)



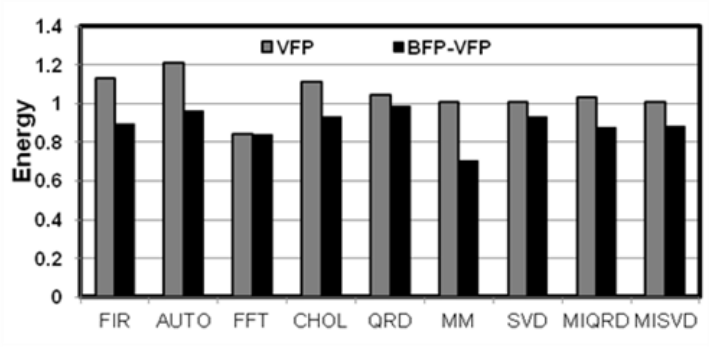
(b)

Figure 5.6: Normalized execution time. (a) VFP, VFP-BFP, (b) SFP. Results are shown relative to baseline FP processor.

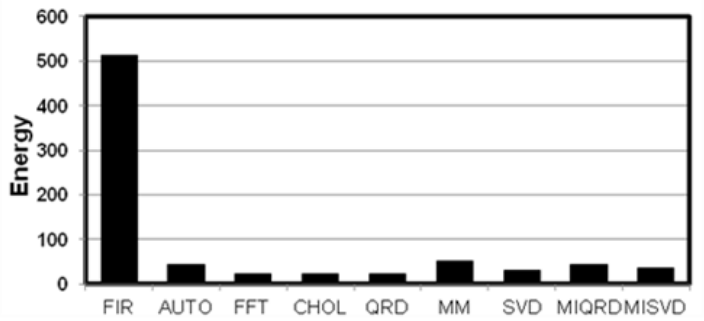
namic range operands in the dot-product is determined during multiplication. In such a case, normal virtual FP operations are utilized to ensure that there is no accuracy loss in the result. This decision is made during runtime according to the difference between product exponents and the block exponent. The block exponent is set equal to the exponent of the product computed by MAC0 or it can be set statically by the programmer.

5.5 EVALUATION

The baseline processor used in this dissertation has a configuration similar to Texas Instruments' TMS320C67 series processors with an additional single-instruction-multiple-data (SIMD) vector unit [50]. Table 5.2 summarizes the area and power overheads of the baseline FP and the VFP processors over the baseline FxP processor. All area and power estimates are based on Verilog descriptions of the FMA unit, ALU, shifter, MUXs and custom logic modules employed in our approach. These Verilog descriptions are synthesized using a TSMC's 65nm standard cell library and Synopsys® Design Compiler. The hardware overheads estimated by synthesis are integrated into McPAT to estimate the overall impact on area and power consumption of the processor. This includes the area and power consumption of on-chip memories and the execution core. The processor area increases by about 29% when dedicated FP units are employed. On the other hand, the VFP approach reduces the area overhead to 5.6% by re-using the FxP resources of the processor. Dedicated hardware FP units in the processor can increase the power consumption by 33%, yet with our approach the power consumption of the processor increases by about 2.8%.



(a)



(b)

Figure 5.7: Normalized energy. (a) VFP, VFP-BFP, (b) SFP. Results are shown relative to baseline FP processor.

The area and power reductions achieved by our approach, relative to dedicated FP units, come at the cost of higher energy consumption and reduced performance. However, by employing our proposed hybrid BFP-VFP based approach, our VFP units can achieve a higher performance and lower energy consumption than dedicated FP units. The performance impact of our proposed approaches (VFP and BFP-VFP) is plotted in Figure 5.6(a). For comparison, I also show the performance of software emulated FP (SFP) in Figure 5.6 (b). Figure 5.7(a) and (b) show the normalized energy consumption of our proposed approaches and SFP respectively. All results shown in Figure 5.6 and Figure 5.7 are normalized to the baseline FP processor with hardware support for FP operations. For comparison, I have also shown the performance and energy con-

sumption of SFP using the *softfloat* emulation library [45]. All the performance and energy numbers are normalized to the corresponding results of the baseline FP processor. SFP results in much higher execution times and energy consumption than VFP. With VFPUs, the performance loss, relative to the baseline FP processor, can be as high as 90% (FIR). However, by employing my hybrid BFP-VFP approach, the worst case performance loss is reduced to 30% for FFT. For all benchmarks other than FFT, my BFP-VFP approach provides significant performance improvements over the baseline FP processor. The lack of performance improvement in the FFT benchmark is because the kernel is coded such that it only employs FP MUL and FP ADD operations. A different FFT algorithm that utilizes FMA operations can provide performance improvements for the FFT benchmarks as well. However, I did not investigate FFT implementations that utilize FMA operations. All benchmarks also show much more energy savings with my proposed approach than with dedicated FP units. Overall, my BFP-VFP approach provides performance and energy improvements as high as 30%, along with significantly lower area and power consumption.

5.6 CONCLUSION

FP arithmetic is becoming increasingly prevalent in low-power mobile devices. Providing low-overhead FP arithmetic support for these devices can greatly reduce their programming complexity and improve their performance, power, and energy efficiency. I proposed novel low-overhead approaches to provide architectural support for FP arithmetic. The proposed approaches can greatly reduce the area and power consumption of the processor while also providing significant performance improvements for common FP kernels employed in embedded computing. Com-

pared to a processor with dedicated FP hardware, a processor with my approaches can reduce the area and power consumption by 24% and 31%, respectively. The proposed approached also provide nearly 30% higher performance and energy efficiency improvements than the baseline FP processor.

6. REDUCED EFFECTIVE PIPELINE LATENCIES FOR GPUS

To efficiently support extensive multithreading, GPUs typically share register files, shared memories and L1 data, constant and texture caches across multiple threads. GPU hardware provides mechanisms to detect and avoid contention in these shared resources. For example, operands read from and written to register files are buffered to ensure that register file bank conflicts do not translate into pipeline stalls [51]. Similarly, shared memory accesses are checked for bank conflicts and reads form the same location are grouped into broadcast messages to reduce pipeline stalls [52]. This requires additional logic to check the shared memory access addresses, detect all the broadcast/bank-conflict cases and efficiently handle them. Similarly, concurrent cache accesses need to be coalesced if possible, to reduce the number of coalescing stalls [53]. Accesses to texture caches also need special handling to determine the texture address and perform any required data conversion (e.g. integer (INT) to floating-point (FP) or vice-versa).

The additional hardware stages required to detect and avoid resource contention increase the pipeline depth of the GPU architecture. Since GPUs do not employ traditional data forwarding networks (DFNs), the increase in pipeline stages directly impacts the read-after-write (RAW) latency of instructions. For example, most instructions in the recent NVIDIA GPU architectures have a RAW latency of 24 cycles [54].

GPUs strive to hide memory and pipeline latencies by interleaving the execution of instructions from different threads. If enough threads are present and are active (not blocked due to memory stalls), GPUs can completely hide the pipeline latency. However, for many GPGPU applications, the number of active threads during execution (or during specific execution phases) is

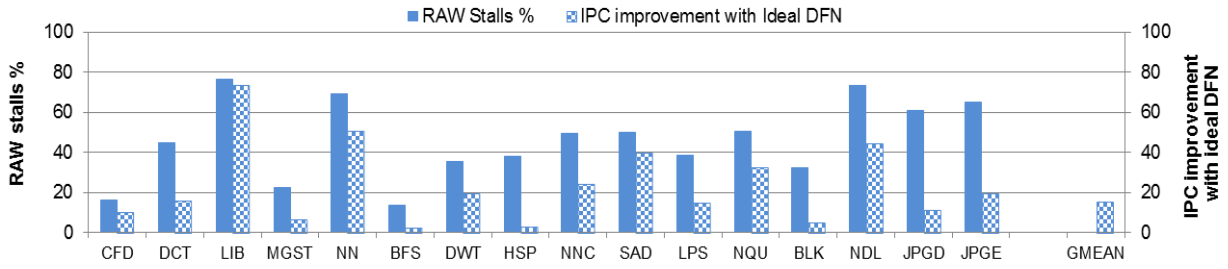


Figure 6.1: Percentage of stalls caused by RAW dependencies and the IPC improvement with an ideal DFN (single-precision floating-point and integer benchmarks)

not enough to hide RAW latencies.

The ideal DFN allows data-forwarding from any of the post-execution stages to any pre-execution stage that occurs after the register file read stage. Figure 6.1 shows the percentage of total cycles that a thread was stalled due to RAW dependencies along with the percentage IPC improvement that can be achieved in the presence of an ideal DFN. It shows that a DFN can considerably reduce RAW dependency stalls and improve average performance by 15% as a result.

Although an ideal DFN can significantly improve the performance of many GPGPU applications, it can be very power hungry. For an NVIDIA® Fermi™ GPU, with two execution pipelines (integer ALU and floating-point (FP)) per GPU core [55], the DFN can increase peak GPU power consumption by nearly 16% (see Section 6.1 for detailed modeling). Consequently, the DFN is extremely difficult to implement in power-constrained GPUs due to its high power consumption (e.g., a GeForce® 8800 consumes 280W [56]). Thus, I propose a compiler-directed forwarding (CDF) approach that provides a performance improvement similar to the ideal DFN but at a considerably lower power cost.

The single precision (SP) and double precision (DP) floating-point units (FPUs) in GPUs are typically implemented with FP fused multiply-add (FMA) units [3]. These FMA units provide

architectural support for addition, subtraction, multiplication, multiply-add, and multiply-subtract operations. Integer multiplications also utilize the FP FMA units to avoid the power and area cost of separate integer multiplier implementations. Assuming data-forwarding is supported in the GPU, an SP FMA operation requires 7 cycles to produce its result and forward it to the dependent instruction [57].

I extend the HFMA architecture proposed in Chapter 4 to efficiently utilize it in GPUs. I show that my proposed HFMA architecture can considerably improve the performance of FP-intensive applications without any significant change in the area and power consumption of the FPU. For DP applications, I utilize the SP FMA units to achieve the same throughput as Fermi but with reduced peak-power consumption.

Although the throughput of many kernels can be increased with more streaming multiprocessors (SMs), power constraints limit the increase in the number of SMs. Since execution units consume a major portion of peak GPU power [58], I also propose a novel technique to reduce the power consumption of GPUs without significantly impacting performance. The key contributions described in this chapter are:

Low-power compiler-directed data forwarding to reduce the impact of RAW latencies: The proposed approach employs a small forwarding buffer (FB) along with compiler modifications to achieve performance improvements similar to a DFN with much less lower power (Section 6.2).

High-throughput FMA units to improve the performance of compute-intensive FP applications: The proposed high-throughput SP and DP FMA units can reduce the effective latency of FP operations to a single cycle with an optional operating mode at the cost of non-IEEE compli-

ant, but more accurate, results (Sections 6.4 and 6.5).

Peak power reduction without performance impact: I demonstrate that the operating voltage of execution units can be decreased without increasing the effective latency of operations through voltage scaling. The allowed voltage reduction for the execution units can lower the peak GPU power consumption without decreasing the performance of kernels (Section 6.6).

Higher performance under a power constraint: The reduction in peak power consumption allows us to activate more SMs under a power constraint. This leads to significant performance improvements for all kernels examined (Section 6.6).

6.1 BACKGROUND

In this chapter, I assume a GPU similar to NVIDIA GTX480 to establish the baseline GPU performance and power consumption. This section provides a brief overview of the baseline architecture.

The baseline GPU architecture consists of 15 SMs, each of which is composed of 32 CUDA cores (also known as steam processors), 16 load/store (LD/ST) units, four special function units (SFUs), and 48KB L1 and 768KB L2 caches. Each SM can support a maximum of 1,536 threads active during execution. These threads are scheduled in groups of 32 threads called warps. The threads in an SM share a 32K-entry register file (RF). Each CUDA core consists of an FPU and an integer ALU. Integer multiplications use the FP multiplier and are thus executed in the FP pipeline. Instructions from these threads can be issued to the CUDA cores, the LD/ST units, or the special function units (SFUs) every cycle. A double precision (DP) unit is shared by two CUDA cores and performs both 32-bit integer multiplications and DP FP operations [59]. Over-

all, each SM can execute 32 SP operations per cycle. The throughput of DP operations and 32-bit integer multiplication is one-half of the SP throughput. I assume that the SP FPUs and the INT ALUs have execution latencies of 7 cycles and 1 cycle, respectively [57].

Table 6.1: Kernel characteristics and IPC.

Kernel	Registers	Shared memory	Block dimensions			CTAs	Active threads/SM	Execution time (%)	IPC
			x	y	z				
CFD_0	47	0	192	1	1	3	576	85.6	167.1
CFD_1	17	0	192	1	1	8	1536	3.6	244.5
DCT_0	31	2112	8	4	2	8	512	79.5	209.7
DCT_1	17	512	8	8	1	8	512	12.6	353.1
LIB_0	25	0	64	1	1	8	512	32.2	215.2
LIB_1	34	0	64	1	1	8	512	67.8	225.0
$MGST_0$	14	12288	96	1	1	4	384	2.7	94.5
$MGST_1$	14	4096	32	1	1	8	256	5.4	36.6
$MGST_2$	19	4096	32	1	1	8	256	7.1	193.2
$MGST_3$	28	0	208	1	1	5	1040	77.5	42.3
NN_0	12	0	1	1	1	8	8	94.5	9.0
NN_1	21	0	5	5	1	8	200	4.2	195.7
BFS_0	18	0	256	1	1	6	1536	100.0	21.9
DWT_0	13	0	512	1	1	3	1536	75.0	190.2
DWT_1	13	0	8	1	1	3	24	25.0	0.4
NNC_0	12	0	16	1	1	8	128	100.0	61.8
SAD_0	31	32	61	1	1	8	488	85.4	330.9
SAD_1	16	0	32	4	1	8	1024	6.5	89.0
SAD_2	16	0	32	1	1	8	256	8.2	86.4
LPS_0	17	2448	32	4	1	8	1024	100.0	390.2
NQU_0	15	15744	96	1	1	3	288	100.0	12.1
BLK_0	25	0	128	1	1	8	1024	100.0	583.4
NDL_0	30	15844	44	1	1	3	132	48.8	37.5
NDL_1	28	15844	44	1	1	3	132	51.2	37.4
$DJPEG_0$	23	2112	8	4	2	8	512	100.0	211.7
$CJPEG_0$	7	0	8	8	1	8	512	60.9	203.4
$CJPEG_1$	23	2112	8	4	2	8	512	39.1	209.0

6.1.1 GPGPU KERNEL ANALYSIS

The RAW pipeline latencies of the FP and INT pipelines are assumed to be 24 cycles and 18

cycles, respectively [54]. GPUs hide pipeline latencies by scheduling instructions from different threads such that, given enough independent threads, the pipeline latency will be completely hidden. For the Nvidia® Fermi™ architecture, up to 768 threads per SM may be required to completely hide pipeline latency (the maximum number of active threads per SM is 1536). However, the number of threads per SM can be limited by several factors. Table 6.1 lists the main kernels in the applications under study along with the number of active threads per SM in each kernel. For each kernel, the bottleneck limiting the number of active threads is highlighted in grey. The IPC for a kernel with fewer than 768 threads is typically considerably lower than the kernels with more threads. However, for memory intensive kernels (e.g. BFS), the IPC even with 1536 threads is fairly low.

Parallel threads in the application:

GPGPU applications are programmed such that their threads are organized in a hierarchical fashion. At the top of this hierarchy is a *grid* that contains thousands of thread blocks. Threads within a block can synchronize and cooperate by sharing data through some shared memory. Different threads blocks however should be completely independent. The maximum number of threads blocks that can concurrently execute in an SM (co-operative thread arrays (CTAs)) is limited to 8 due to the limited resources of the SM. While these constraints are not critical for graphics applications, several GPGPU applications are limited by the number threads in a block or the total number of thread blocks in the kernel. In Table 6.1, all applications other then CFD and NDL have kernels that are limited by the number of threads per block (see the highlighted block dimensions and CTAs column).

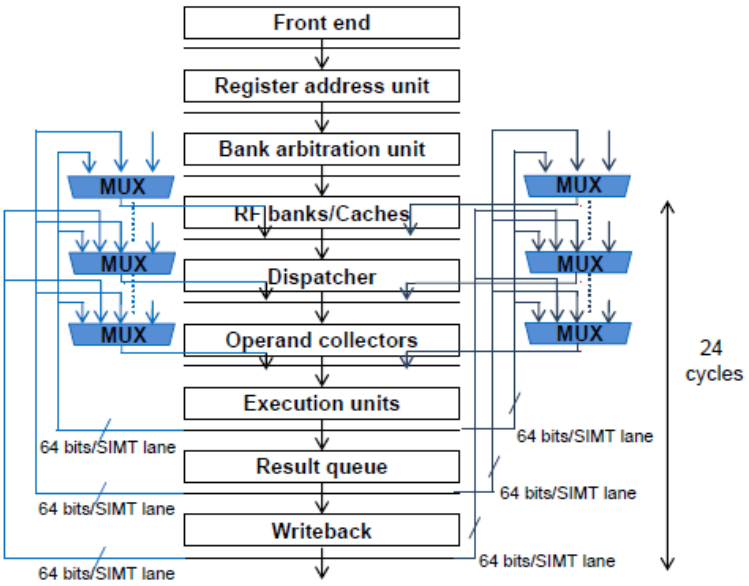


Figure 6.2: Overview of a GPU pipeline and the forwarding paths required in an ideal DFN [51, 55]

Shared resources:

The number of active threads per SM may also be limited due to the size of the shared resources. Each SM shares the register file and shared memory among all the active threads. The number of registers and the amount of shared memory used per thread is determined at compile time and the GPU thread distributor utilizes this information to determine the maximum number of threads that can be distributed to each SM, ensuring enough resources are available per thread. In Table 6.1, CFD_0, $MGST_0$, $MGST_3$, NQU_0, NDL_0 are NDL_1 are limited by shared memory and/or the number of registers.

Memory intensive applications:

For memory intensive kernels (e.g. BFS_0) the RAW pipeline latency may become visible if multiple threads are blocked due to pending memory requests. The round-robin warp scheduler in

GPUs produces a fine-grained interleaving of threads which can create scenarios in which multiple threads stall due to long latency memory operations in a short time window [37]. In such a case, if the effective latency of operations is reduced, the threads that have not stalled because of memory can still execute dependent instructions without waiting for the complete pipeline latency.

6.1.2 COST OF AN IDEAL DATA-FORWARDING NETWORK

For an ideal DFN, forwarding paths are required from each stage after execution to all stages between the register file read and execution. An overview of a GPU SM pipeline along with the forwarding paths required for the DFN is shown in Figure 6.2. Based on publically available NVIDIA resources, I assume there are at least three post-execution stages that require forwarding paths to pre-execution stages [51, 55]. The 32-bit execution result from each of the two execution pipelines (FP and INT) needs to be forwarded to the pre-execution stages. With a 24-cycle RAW latency for the pipeline and three post-execution stages, there are fourteen pre-execution stages to which the result needs to be forwarded. This requires a total of 2×14×3=84 forwarding paths per SM in the baseline GPU. Each of these 84 paths is 32×32=1024-bit wide (32-bit wide datapath per SIMT lane × 32 SIMT lanes). Moreover, the pre-execution stages also need additional multiplexors to select the proper operand source. In order to estimate the power overhead of the forwarding paths, I employ the same wire capacitance model as Gebhart *et al.* [37] (with an operating voltage of 1.1V). I also synthesized hardware descriptions of multiplexors and estimated their power consumption using Synopsys PowerCompiler® to determine the power overhead of the DFN. With a baseline GPU peak power of 250W, a DFN increases the peak power by about 16% (the peak power is 290W with the DFN).

PC	Instruction	Exposed Latency	Instruction with CDF	FB entry written	Exposed latency with CDF
0x0D8	shl %r13, %r11	18	shl %r13, %r11	FB2	18
0x0E0	add %r14, %r13, %r9	18	add %r14, **%FB2**, %r9	FB0	18
0x0E8	cvt.s64 %rd2, %r14	18	cvt.s64 %rd2, **%FB0**	FB1	1
0x0F0	mul %rd3, %r14,4	18	mul %rd3, **%FB0**,4	FB2	1

Figure 6.3: PTX code excerpt from the DCT benchmark

6.2 COMPILER-DIRECTED DATA FORWARDING (CDF)

6.2.1 MOTIVATION AND OVERVIEW

Deep GPU pipelines increase the instruction RAW latencies and limit the performance of many GPGPU applications as demonstrated in Figure 1. While traditional data-forwarding can ameliorate these inefficiencies, the power overhead associated with DFNs limits their implementation in GPUs. As a low-cost alternative, I propose a compiler-directed data forwarding (CDF) technique that can achieve performance comparable to a DFN at a considerably lower power overhead. Moreover, unlike a conventional DFN, CDF also allows the GPU to tolerate even longer RAW latencies without increasing the complexity/power of the forwarding logic. I exploit this to reduce the power consumption of GPUs (Section 6.6).

For CDF, I include a forwarding buffer (FB) within each CUDA core as shown in Figure 6.4(a). The FB temporarily stores the result of each instruction as it completes execution. The FB has dedicated buffers for each active warp (up to 48 warps for Fermi). Instructions from the same warp write results to different entries within the buffer allocated for the warp. Each buffer contains three entries and can store the result of the three most recently executed instructions for each warp. I chose three entries per warp because three entries provided the best power, per-

formance tradeoff.

If an instruction uses operands from the FB entries (if it uses operands which are the results of any of the last three instructions), the compiler replaces the RF register for that instruction with the FB entry number. The entries for each warp in the FB are written in a round-robin fashion. For example, with a FB that has three entries per warp, results from instructions *a*,*b*,*c*, and *d* are written to FB entries 0, 1, 2 and 0, respectively. Since the FB can only hold three entries, they are replaced by the results of new instructions after three instructions. If all the source operands of an instruction are specified to be read from the FB, the instruction only needs to wait for the execution latency (not the complete RAW latency of the pipeline).

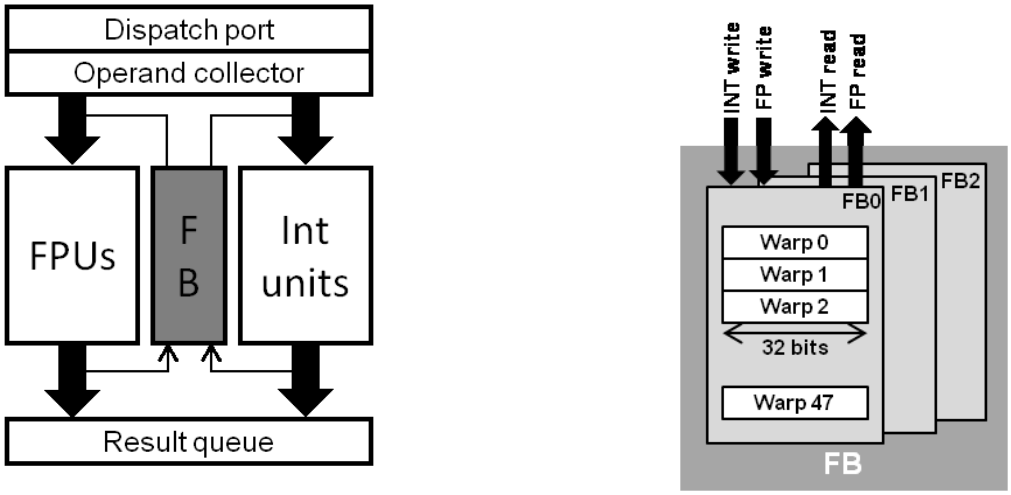
Figure 6.3 shows a PTX code excerpt from the DCT benchmark along with the modified instructions and the exposed latency of the parent instruction for each dependent instruction. As instructions complete execution, their results are temporarily stored in FB entries. The FB has three entries per warp and thus can store the results of the three preceding instructions. In Figure 6.3, the instruction at PC 0x0E0 stores its result in FB entry 0 (FB0) of the corresponding warp. Since the ADD instruction only requires one cycle to compute its result, the dependent CVT instruction can read its only source operand from FB0. The CVT instruction can thus be issued as early as the cycle after the ADD instruction. Similarly, for the MUL instruction, the only non-immediate source operand (%r14) is available in FB0. The exposed latency of the ADD instruction for the CVT and MUL instructions is only a single cycle, instead of the complete 18 cycle RAW latency of the baseline GPU.

NVIDIA's Fermi architecture already implements register scoreboarding for instruction scheduling [60]. The proposed approach can be incorporated within the scoreboarding logic to

allow early scheduling of instructions with forwarded operands. Although the more recent NVIDIA architecture (Kepler) does not implement scoreboarding [60], it utilizes the compiler to indicate the latency of instructions to simplify the instruction scheduling hardware. My approach can also be implemented on a Kepler-based architecture. Specifically, in addition to replacing source operands with FB entries, the complier can also check if all the source operands of an instruction are read from the FB, in which case it can indicate the instruction as an early-schedulable instruction using the mechanism already in place to deal with the latency of other instructions.

6.3 IMPLEMENTATION DETAILS

In order to implement CDF, I added an additional compiler phase to check the dependencies between instructions and substitute source registers with appropriate FB entries. The PTX intermediate representation generated by the NVIDIA compiler is converted to an architecture spe-



(a) (b)

Figure 6.4: Forwarding buffer (FB) (a) within CUDA core and (b) banked architecture

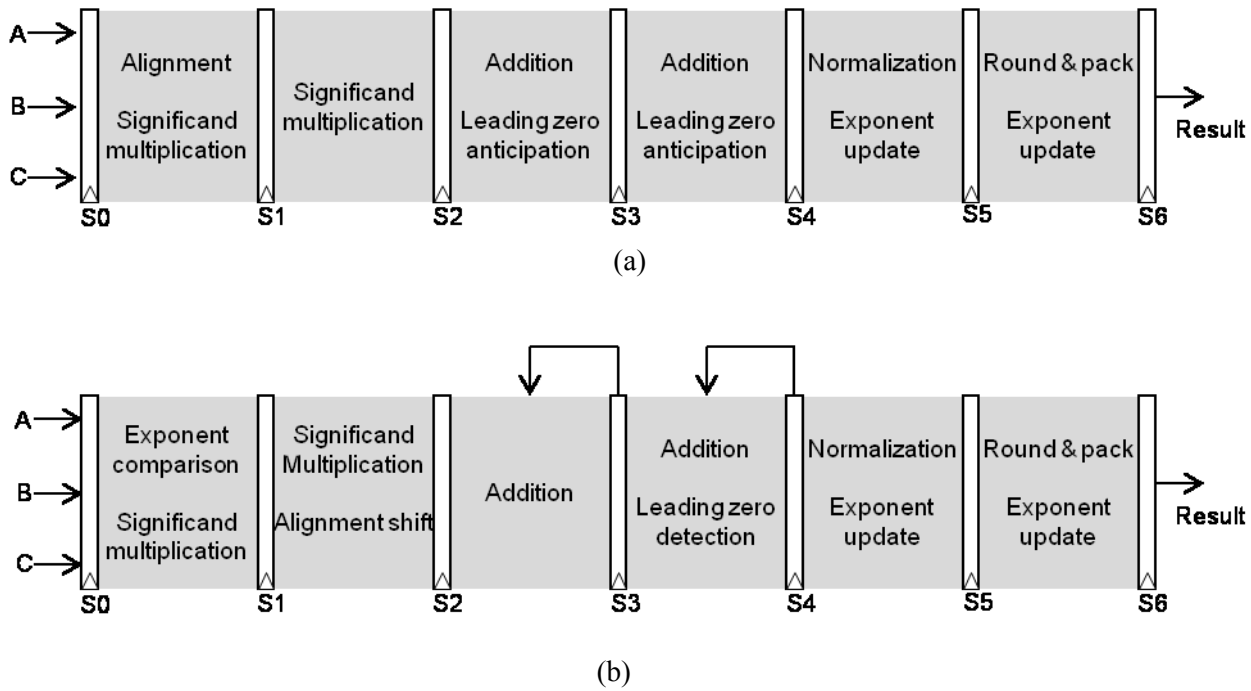


Figure 6.5: Pipeline overview of an FMA unit (a) Baseline, (b) Proposed

cific ISA by the GPU driver. During this conversion, appropriate source registers are replaced with FB entries.

Figure 6.4 shows the FB along with the integer and FP execution pipelines within a CUDA core. The FB is implemented as three SRAM banks (FB0, FB1 and FB2). Each FB bank has two read and write ports. This allows the integer and FP pipelines to independently read and write results to FB banks without any port conflicts. Each FB bank has a dedicated entry for each of the active warps. With a maximum of 48 active warps in Fermi, the FB banks are 48 deep.

During instruction scheduling, the source operands of an instruction are checked. If all the source operands of an instruction are read from the FB or are immediates, the instruction can be scheduled early. If the predecessor instruction was an FP instruction (or an integer multiply instruction), the dependent instruction only needs to wait 7 cycles (instead of 24) before it can be

issued. If the predecessor instruction was executed in the integer ALU, the dependent instruction can be executed in the very next cycle.

Overall, FBs increase the GPU peak power consumption by about 2% (compared to a 16% power increase for a DFN). However, utilizing FB improves the performance of applications by 8% (geometric mean) relative to the baseline GPU. I will exploit this performance increase and latency tolerance to reduce the power consumption of execution units without negatively impacting performance.

6.4 HIGH-THROUGHPUT FMA (HFMA) UNIT

6.4.1 MOTIVATION AND OVERVIEW

Figure 6.5(a) and (b) show a pipeline overview of a typical FMA unit and our proposed unit, respectively. The HFMA unit is similar to the one proposed in Section 4 but allows IEEE-754 precise results by extending the adder bitwidth and also supports different IEEE-754 rounding modes. It is also tailored to a multi-threaded architecture by providing internal buffers to store multiple intermediate FMA/ADD/SUB results. The pipeline stages are marked S0-S6. While the details of the two FMA units are discussed in Sections 6.4.2 and 6.4.3 respectively, it is important to note that in the baseline FMA unit pipeline, results cannot be forwarded to previous pipeline stages within the FMA unit. This is because a dependent FP operation needs the result of the previous operation (after normalization, rounding and packing) before it can perform alignment shift in the first stage of the pipeline. It thus needs to wait 7 cycles before the result from S6 becomes available. This 7-cycle dependence loop exists even if an ideal DFN is employed.

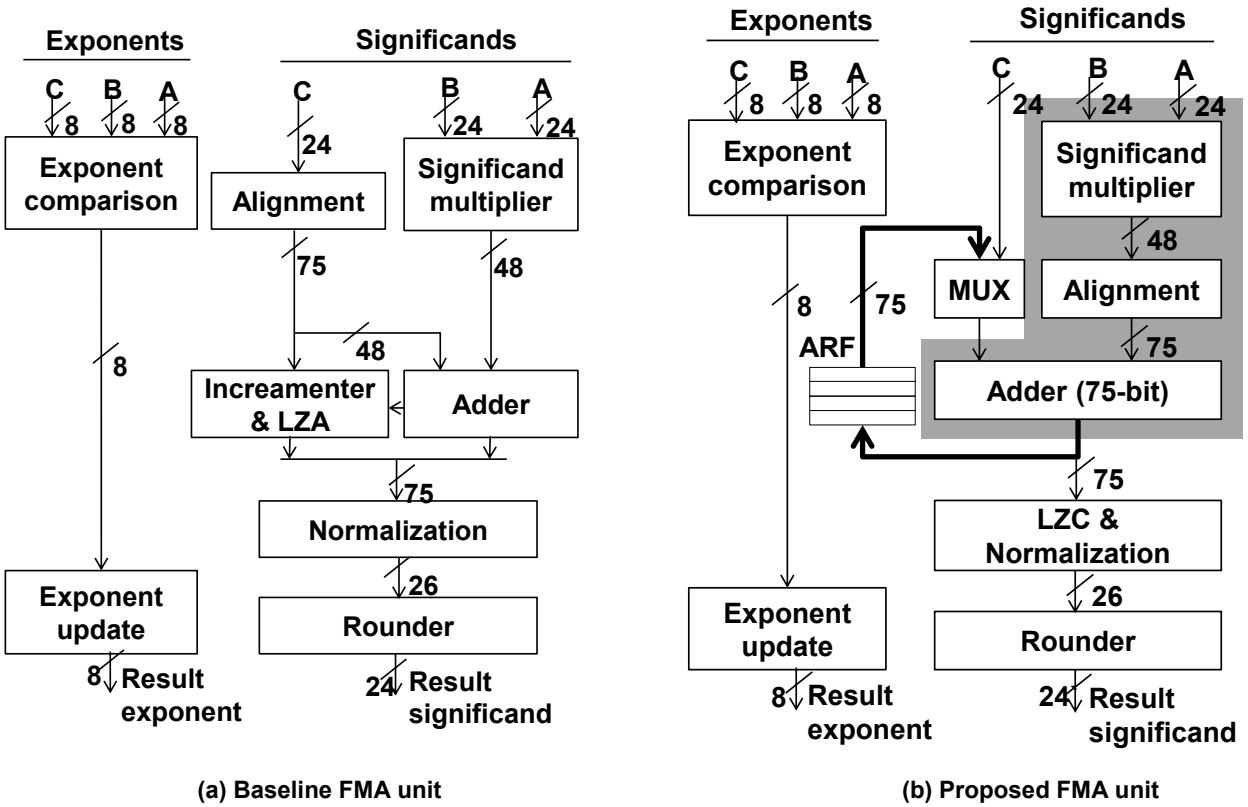


Figure 6.6: Data paths for baseline and proposed SP FMA units.

The proposed FMA unit reduces this dependence loop to a single cycle such that the FMA/ADD/SUB result can be forwarded to the dependent operation the very next cycle. This is achieved by predicting the exponent range of the input operands to be within the interval [-13:13]. If this prediction is correct the result is forwarded immediately. Otherwise, the pipeline stalls to allow the previous result to be computed after S6 and forwarded to the current instruction. The exponent range of [-13:13] allows the internal modules of an FMA unit to be utilized differently (and more efficiently) for dependent FMA/ADD/SUB operations. For all applications examined, our static exponent prediction is correct over 96% of the time.

6.4.2 BASELINE FMA UNIT

The data-path for our baseline FMA design is illustrated in Figure 6.6(a)[48, 61]. The FMA

unit takes three FP operands *A*, *B,* and *C* and performs $A \times B + C$. The significands (*A* and *B*) are multiplied using the significand multiplier in parallel with the alignment shift of operand *C*. The shift amount for aligning operand *C* is determined by comparing the exponents of *A*, *B,* and *C* such that the significand of *C* is aligned with the product ($A \times B$). Alignment ensures that *C* and the product have the same exponent and thus can be added together. For dependent FP operations, this alignment shift amount cannot be determined until the parent operation has completed execution (after stage S6 in Figure 6.5). Once the multiplication is completed, the shifted operand *C* is added to the product. The adder inputs are also used by the leading-zero anticipator (LZA) to estimate the left-shift amount required for the normalization of the result. In the final step, the normalized result is rounded according to the employed rounding mode.

6.4.3 PROPOSED HIGH-THROUGHPUT FMA UNIT

The data-path for our proposed high-throughput FMA (HFMA) unit is illustrated in Figure 6.6(b). It is derived from the FMA unit proposed in Section 4 [12]. The proposed design extends the bit-width of adder to allow IEEE-754 precise FMA results and also supports different IEEE-754 rounding modes. It also included a smaller accumulator register file (ARF) to allow multiple threads to maintain an intermediate FMA/ADD/SUB result for forwarding to dependent instructions. Our approach uses similar modules as the baseline FMA unit, but I design the data path differently to allow data forwarding within the FMA unit. This allows dependent FMA, ADD, and SUB operations to be issued in consecutive cycles, effectively reducing the latency of these operations to a single cycle. In my approach, instead of aligning operand *C* to the product ($A \times B$), I align both the product and the operand *C* to a pre-determined exponent (i.e., block exponent (exp_{blk})) and statically predict that the exponents of the input operands will be within the range

[exp_{blk} -13: exp_{blk} +13].

The adder output in my design is applied back to its input to be added with the subsequent FMA/ADD/SUB operation without normalization and rounding, leading to a single-cycle feedback path. This is highlighted in Figure 6.6(b). Moreover, since all operations are aligned to the pre-determined block exponent, the dependent operation does not need the exponent of the previous FMA/ADD operation for alignment. This allows dependent FMA/ADD operations to be issued in consecutive cycles because they do not need to wait for the normalized and rounded significand from the previous operation before issuing the current operation. The unrounded and unnormalized result is forwarded to the dependent operation using the single-cycle feedback path. The hardware sets the block exponent per warp according the exponent of the result of the first FMA/ADD/SUB operation in a series of dependent FMA/ADD/SUB operations.

The output of the adder is applied back to its input through a 2:1 multiplexer (MUX) shown in Figure 6.6(b). The MUX allows the addend to be either the operand *C* for normal multiply-add operations or the adder output for dependent accumulate operations. Instead of forwarding the adder output back to its input, the adder output is written to and read from an accumulator register file (ARF) supporting one port each for reads and writes. This allows instructions from different warps to be issued in consecutive cycles such that each thread uses a different accumulator register to maintain the current accumulated value. For the maximum number of 48 warps per SM allowed in the baseline GPU, a 48 × 75-bit ARF is required. If only a single warp is active, the addition result can be read from the pipeline flip-flops; otherwise, the result is read from the ARF by indexing it with the warp ID.

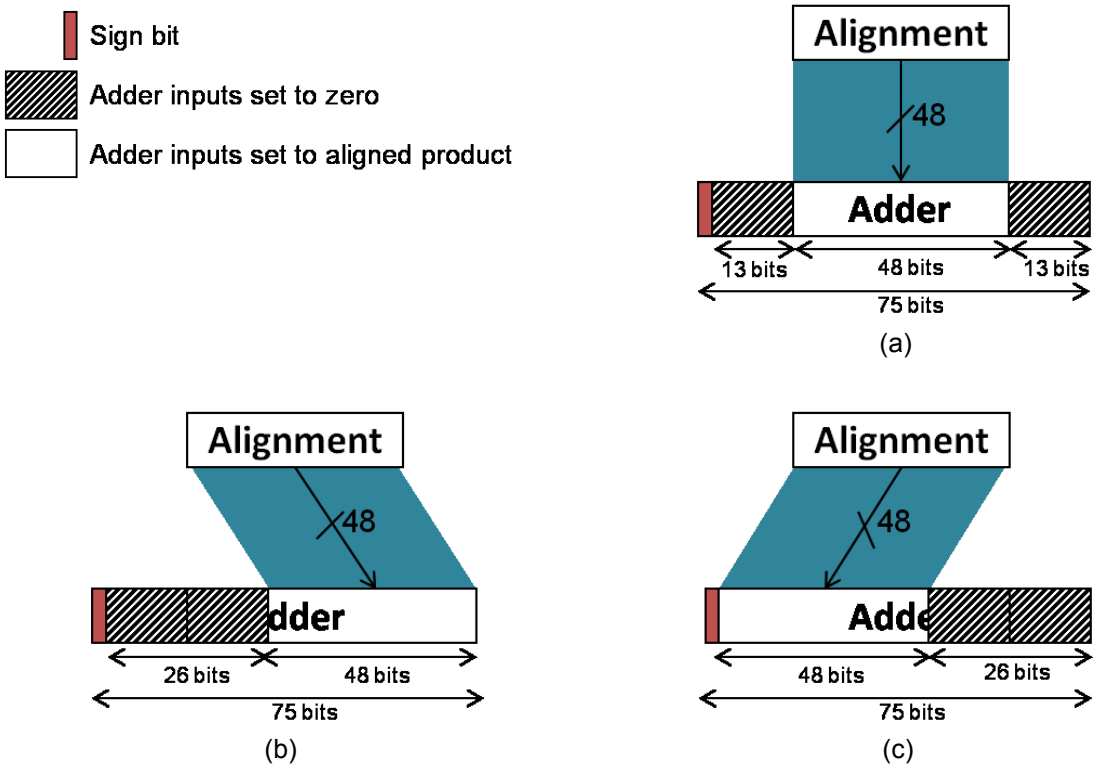


Figure 6.7: Alignment of product with the block exponent: (a) $d = 0$ (b) $d = 13$ (c) $d = -13$.

Exponent range coverage: The SP HFMA unit utilizes a 75-bit adder for adding the addend and the product. The 75-bit adder ensures that the unrounded 48-bit multiplication result can be added to the 24-bit shifted significand C without any loss of accuracy for FMA operations. The 75-bit adder supports MACC and AACC operations within the dynamic range of ±13 with respect to the block exponent. This implies that FP numbers with magnitudes in the range $2^{13+\ expblk}$ to$2^{-13+\ expblk}$ will execute in my approach without experiencing the complete RAW latency of the pipeline. If the difference between the block exponent and the exponent of the product (exp_A + exp_B) is zero, the product is shifted such that it is aligned to bits 60:13 of the adder. For an exponent difference $d = exp_A + exp_B - exp_{blk}$, the product is shifted such that it aligns with adder inputs $(60 + d)$:$(13 + d)$ of the adder. For d greater than 13, the product will lose its most signifi-

cant bits (MSBs) when it is aligned with the block exponent and for *d* less than 13 it will lose its least significant (LSBs), as illustrated in Figure 6.7.

Since integer multiply operations are also executed in the FP pipeline, our HFMA unit also allows dependent integer multiply-and-add (MAD) operations to utilize the same single cycle feedback path as the FP operations. Consequently, the HFMA unit reduces the effective latency of both the FP FMA and integer MAD instructions.

IEEE-754 compliance: Although, my approach still implements FMA operations that conform to the IEEE-754 [28] requirements, for FMA/ADD/SUB operations, my approach may yield different results than those achieved by standard FMA or ADD operations. This is because if all the accumulate operations are have an exponent range within [exp_{blk} -13: exp_{blk} +13] then our approach forwards the unrounded addition result to be used as the addend for the next operation. In conventional IEEE-754 compliant architectures, the result of the addition needs to be rounded and normalized before it can be used as the addend for the dependent operation. However, as long as the exponent range of the FMA/ADD/SUB operands is within ±13, my approach guarantees that the accuracy of the results will not be less than the accuracy achieved through standard IEEE-754-compliant SP operations. Furthermore, my proposed HFMA can achieve higher accuracy than conventional FMA instructions for certain computations because I do not perform intermediate rounding after each operation.

If the exponent range of the operands is outside the range [exp_{blk} -13: exp_{blk} +13], a loss of accuracy can occur because the 75-bit adder cannot completely accommodate the two operands. This loss of accuracy is avoided by reverting back to normal FP-FMA operations to deal with such operands in my proposed HFMA. Thus, only operations with high dynamic range will need

to tolerate the complete latency of the FMA. The exponent range prediction is checked in the first cycle of the FMA pipeline (by the exponent comparison module). If the prediction is determined to be incorrect, the pipeline is stalled until the normalized result from the last stage of FMA execution is available. The stall mechanism uses the logic already in place for stalling due to writeback register bank conflicts or shared memory bank conflicts.

These stalls, however, are extremely rare for real applications (less than 4% of operations). Moreover, to support DP operations (Section 6.5), I extend the sizes of some of the HFMA components. The extended sizes also increase the exponent range covered by our HFMA units for SP operations to [exp_{blk} -56: exp_{blk} +56]. With the extended range, the processor never experienced any SP operations that mispredict in the applications I examined.

6.5 DOUBLE-PRECISION HFMA UNIT

6.5.1 MOTIVATION AND OVERVIEW

High-end GPUs tailored for the scientific computing market typically have hardware support for high-throughput DP arithmetic. However, due to wider bit width than SP arithmetic, supporting DP arithmetic requires much more area and consumes much more power. Meanwhile, the throughput of DP operations is typically less than that of SP operations because the bandwidth of the register file, which also supplies the operands for the DP FMA, is optimized and only sufficient for the peak throughput of SP operations.

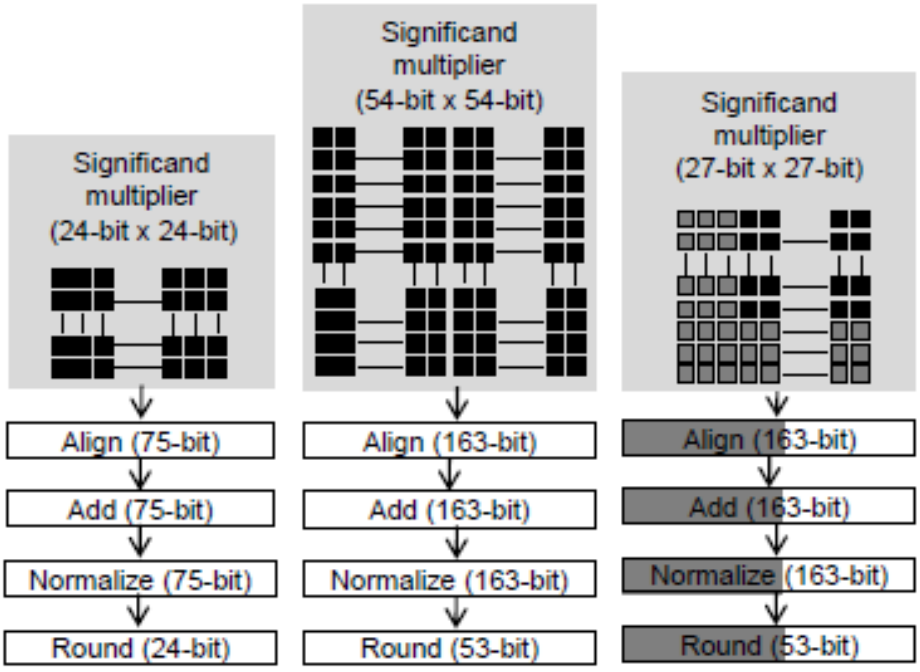
For some GPU architectures, the DP throughput is one-eighth of the SP throughput [52]. The Fermi architecture has DP throughput that is one-half of SP throughput. In this section, I extend our proposed SP HFMA design to implement DP FMA operations that can provide the same throughput as the baseline Fermi architecture. This allows my approach to support DP FMA op-

erations at a very low cost compared to conventional DP FMA operations.

DP FMA units consume considerably more power than SP FMA units. The primary source of the increase in power is the larger DP multiplier. This is because as the data bit-width is increased, the power consumption of the multiplier increases quadratically, while the power consumption of most of the other components of an FMA unit (e.g. adders, shifters) increases linearly. I propose to reduce the peak power consumption of the GPU by enhancing the SP FMA units to allow computation on DP operands without significantly increasing the power cost. The inline multipler, alignment-shifter and adder organization (highlighted in Figure 6.6(b)) in my proposed HFMA unit allows the unit to accumulate the product after shifting it by different amounts, without any additional overhead. I divide the DP operand significands A and B into A_{HI}, A_{LOW}, B_{HI}, and $B_{LOW,}$ respectively. For the 53-bit DP significand, the division of significands into *HI* and *LOW* parts is such that the 27 LSBs are considered *LOW* bits while the 26 MSBs are considered *HI* bits. The DP significand multiplication can then be performed by multiplying and accumulating the products $A_{HI} \times B_{HI}$, $A_{HI} \times B_{LOW}$, $B_{LOW} \times A_{HI,}$ and $A_{LOW} \times B_{LOW}$.

6.5.2 IMPLEMENTATION DETAILS

DP FMA operations require a 53-bit × 53-bit significand multiplication, instead of the 24-bit × 24-bit multiplication used in SP FMA unit. Moreover, the bit width of adders and shifters required for supporting DP FMA operations is also higher than SP FMA units. Based on my synthesis and power estimation results, a DP significand multiplier consumes more than four times more power than its SP counterpart. As shown in Figure 6.6(b), in my proposed SP HFMA design, the product is shifted (alignment shift) and then added to operand C or the accumulated value. The multiplier followed by the shifter allows the design to implement DP significand mul-



Normalized power			
	SP HFMA	**DP HFMA**	**SP HFMA + DP**
Leakage	1	3.7	1.2
Dynamic	1	4.6	1.4

Figure 6.8: Data path bit widths and normalized power consumption of (a) SP HFMA (b) DP HFMA (c) SP HFMA with DP support (SP HFMA + DP).

tiplication as four smaller multiplications and accumulate their results in the accumulator with the proper shift amounts. To accommodate the 53-bit-wide DP significand, I extend the 24-bit-wide SP significand multiplier to 27 bits. Moreover, to support the wider bit width and exponent range of DP arithmetic, the shifter and adder bit widths are also increased appropriately.

The bit widths of the main data-path elements of our SP HFMA, DP HFMA, and SP HFMA with DP support are shown in Figure 6.8. The modifications required in the data-path modules to support DP operations on SP units are highlighted in Figure 6.8(c). The increase in bit width

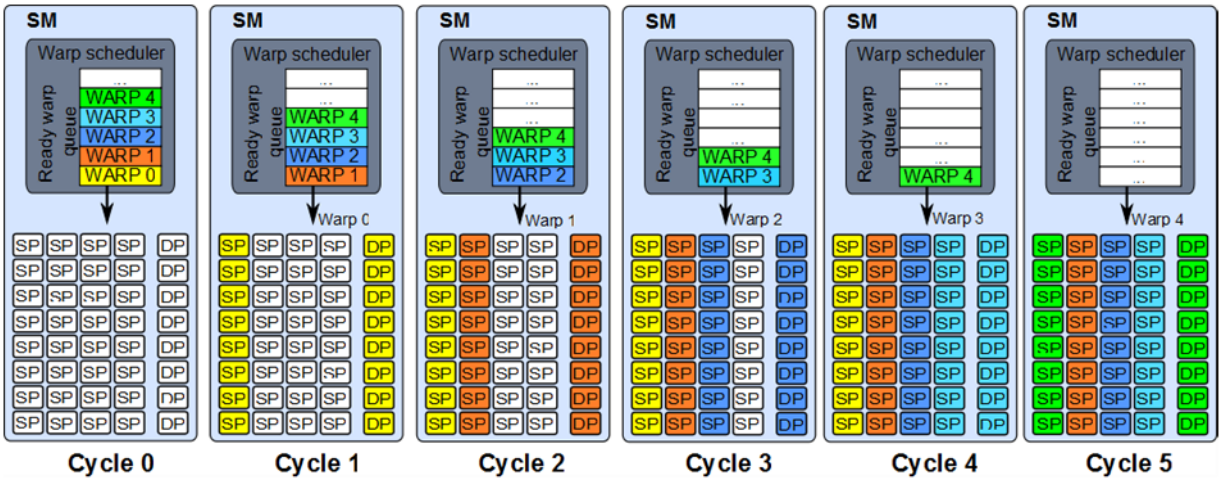


Figure 6.9: Scheduling of DP operations across SP and DP FMA units

does not increase the power consumption for SP operations because the additional logic can be clock-gated [48]. For each DP multiply/FMA type operation, the multiplier is occupied for four cycles. This implies that no new instructions may be issued to the HFMA for 4 cycles.

I extend the shifters and adders of all the SP HFMA units to 163 bits, which enables DP FMA operations on all 32 SP HFMA units. This allows me twice as many FMA units capable of performing DP arithmetic operations each cycle as the baseline Fermi-like architecture (16 DP units). Overall, my approach provides one-half of the throughput of the baseline Fermi architecture (each HFMA can execute new DP instruction every four cycles).

To achieve the same DP throughput as the Fermi architecture, I include eight DP HFMA units (with 53 × 53-bit significand multipliers) and also use the 32 SP HFMA units to perform DP arithmetic. The 8 DP units can together execute eight DP operations every cycle while eight of the 32 SP units can issue DP operations every cycle because each SP HFMA unit can issue DP FMA operation every four cycles. Figure 6.9 shows an example where five warps issue DP op-

erations in consecutive cycles. For clarity, I assume that each warp only consists of 16 threads instead of 32 because the processor can issue only 16 DP operations per cycle from a warp. In such a case, a warp requires only one cycle to completely issue instructions from all of its threads.

In cycle *0* all the SP and DP units are idle and all five wraps are ready to be issued. In cycle *1*, the 16 threads of warp *0* are issued to the FMAs. Eight of the 16 threads execute their DP instruction on the eight dedicated DP units. The remaining eight threads perform DP instructions on the first (left-most) set of SP units. Since the DP units can execute instructions every cycle, in cycle *2*, eight threads of warp 1 are issued to the DP units. However, the first set of SP units uses the multiplier for four cycles and thus cannot execute new instructions until cycle *5*. The remaining eight threads of warp 1 thus execute on the second set of SP units.

Similarly, in cycles *3* and *4,* warps 2 and 3 issue instructions to the DP units and the next set of SP units. At cycle *5*, the first set of SP units is ready to issue new instructions. Eight threads of warp four can therefore issue to these SP units again. The remaining eight threads still execute on the DP units.

The proposed HFMA unit still provides a single-cycle effective latency for DP and SP MACC and AACC operations. With eight DP HFMA units per SM instead of the 16, the proposed architecture achieves the same DP throughput as the baseline architecture. However, it reduces the peak power of the baseline GPU by 7%.

6.6 POWER AND PERFORMANCE IMPROVEMENT THROUGH VOLTAGE SCALING

Execution units (FPUs and ALUs) in GPUs consume a significant portion of the overall power

Table 6.2: GPU simulation parameters

# of SMs	15	# of Memory Channels	6
SM Freq	1.4GHz	Memory Freq	1848MHz (GDDR5)
On-chip Interconnect Freq	0.7GHz	Memory B/W	177.4 GB/s
Warp Size	32	B/W per Memory Module	4 (Bytes/Cycle)
SIMD Width	32	Memory Controller	FR-FCFS
# of Threads per SM	1536	Branch Divergence	Immediate post dominator
# of CTAs per SM	8	Warp Scheduling	Round Robin
# of Registers per SM	32768	Const. Cache Size per SM	8 KB
L1$/L2$ Memory per SM	16KB/768KB	Texture Cache Size per SM	12 KB

and area (up to 35% of peak GPU power [58]). Any reduction in their power consumption can result in a considerable reduction in the peak power requirements of the GPU. The reduced peak power consumption can be exploited to improve the performance under a power constraint. In this section, I first discuss how the power consumption of execution units can be reduced by applying voltage scaling to the execution units connected to a separate voltage island [62, 63]; considering the large chip area and power consumption of the execution units and emerging high-efficiency on-chip voltage regulator technology [64], it is compelling to consider this option Although voltage/frequency scaling typically leads to a reduction in performance, I show that our CDF and HFMA approaches ameliorate the performance loss due to voltage scaling while still providing significant power reduction. I also demonstrate how the performance of GPUs can be increased by exploiting the reduced power consumption of execution units.

6.6.1 INCREASING INSTRUCTION THROUGHPUT:

To reduce the power consumption of execution units, dynamic voltage and frequency scaling (DVFS) can be applied to the CUDA cores in a GPU. I reduce the voltage of the execution units such that the logic delay is doubled. However, the increase in logic delay results in an increase in the latency of instructions. Moreover, due to the frequency reduction, the instruction throughput

Table 6.3: Benchmarks and their acronyms

Name	Acronym	Name	Acronym
CFD Solver	CFD	k-nearest neighbor	NNC
Discrete Cosine Transform	DCT	Sum of Absolute Differences	SAD
LIBOR Monte Carlo	LIB	3D Laplace Solver	LPS
Merge Sort	MGST	N-Queens Solver	NQU
Neural Network	NN	Black Scholes	BLK
Breadth-first Search	BFS	Needleman-Wunsch	NDL
Discrete Wavelet Transform	DWT	JPEG-decode	JPGD
Hot Spot	HSP	JPEG-encode	JPGE

is also reduced. These factors significantly reduce the performance of compute-bound kernels.

For reduced-voltage execution units to maintain the same throughput as the baseline units, I double the number of pipeline stages of execution units. Note that voltage domain crossing is much cheaper than the frequency domain crossing and the dual voltage technique (with a single frequency domain) is commonly used to optimize power consumption of digital ICs [62, 65]. Consequently, the execution units can be operated at the same frequency at lower voltage. This ensures the overall throughput of instructions does not decrease even when they are operated at a reduced voltage.

6.6.2 *EXPLOITING CDF AND HFMA FOR REDUCED RAW LATENCIES*:

Although additional pipeline stages can ensure that instruction throughput does not decrease at the reduced operating voltage, they still double the latencies of instructions. However, with CDF and HFMA approaches I can greatly reduce the impact of increased RAW latencies. Specifically, for the IALU, I subdivide each FB bank into two banks (HI and LO). The HI bank stores the 16 MSBs of the 32-bit ALU result while the LO bank stores the 16 LSBs.

With voltage scaling, the HI and LO banks are written and read separately to allow early forwarding of partial addition/subtraction results to dependent additions. Based on our synthesis results all IALU instruction except addition and subtraction have a much lower logic delay that can be accommodated within a single cycle even when the voltage is reduced. Although integer

multiplications will also experience increased latency at the reduced operating voltage, IMUL utilizes the FMA unit multiplier and is treated as an FP instruction. If an addition/subtraction instruction can utilize the CDF approach, it can execute assuming a single cycle latency for ALU instructions as partially computed ALU results can be forwarded every cycle to dependent instructions. For other ALU instructions if the CDF approach can be exploited, the complete execution result can be computed and stored in the FB to allow early scheduling of dependent instructions. Otherwise, the dependent instructions will experience an increased RAW latency due to the reduced voltage.

For FP instructions, unlike the conventional FMA units, our proposed HFMA unit does not increase the effective latency of FMA/ADD/SUB operations, although FMA units operate at a reduced operating voltage with a deeper pipeline. This consequently allows dependent FMA/ADD/SUB operations to be issued every cycle even at the reduced operating voltage. Because the result from the previous operation is forwarded to the dependent operation without rounding and normalization, there is only a single-cycle dependency between the adder output and the adder input. Moreover, the adder can be arbitrarily pipelined without increasing the length of the dependency loop. Thus, even when logic delay is doubled due to voltage scaling, the additional pipeline stages allow the addition to be carried out in two cycles without increasing the length of the feedback loop or the effective latency of the FMA/ADD/SUB operations. The ARF is also sub-divided into HI and LO banks similar to the CDF to allow the LSBs and MSBs of addition results to be forwarded in successive cycles.

The reduction in peak power of the execution units can be utilized to improve performance by turning on more SMs. The peak power reduction of the GPU through voltage scaling of execu-

tion units can also be used to increase the number of SMs in the GPU by 14% under the same peak power constraint. In my experiments, I increase the number of SMs in the GPU to 17 (from 15). This can increase the mean speedup for compute-bound kernels to 33% (DP benchmarks).

6.7 EVALUATION

To evaluate the performance benefit of my proposed approaches, I use GPGPU-Sim for performance simulations [66]. The baseline is configured to simulate a GPU similar to NVIDIA's GTX 480 [55]; see Table 6.2 for the detailed simulation parameters. The benchmarks used for performance simulations and their acronyms are listed in Table 6.3.

To evaluate the peak power consumption of GPU components, I integrated McPAT [67] with GPGPU-Sim [66]. The power estimates of the different GPU components are extensively validated using several micro-benchmarks that exercise the corresponding micro-architectural blocks

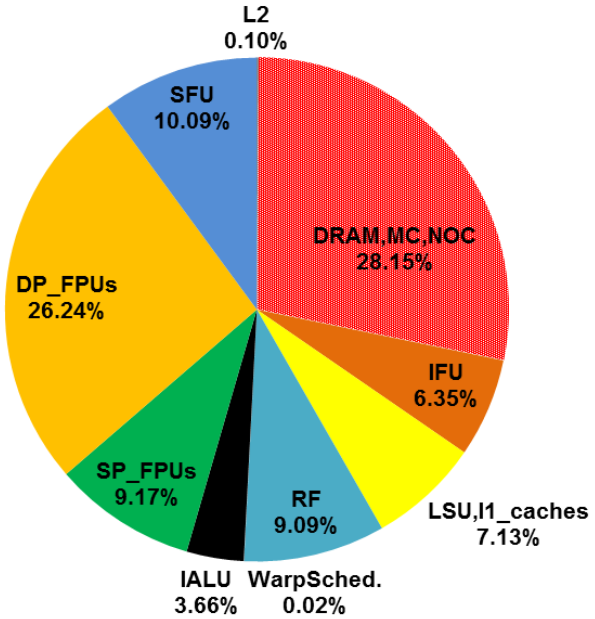


Figure 6.10: Estimated peak power breakdown for an NVIDIA GTX 480 GPU

Table 6.4: GPU peak power impact

Power model parameters		
Wire capacitance		300 fF/mm
Wire power @ 1.1V (32 bits)		8.1 mW/mm
SP FMA unit area		12702 μm^2
SP FMA estimated height		112.7 µm
Baseline GPU peak power		250 W
Power overheads evaluation		
DFN	Pipelines	2 (FPU & IALU)
	Stages b/w RF and execute	14
	Stages after execute	3
	GPU peak power with DFN	290 W
CDF	FB bank power (dynamic/leakage)	4.0 mW/32.5 µW
	GPU peak power with CDF	256 W
CDF + HFMA	ARF bank power (dynamic/leakage)	3.6 mW/204 µW
	SP FMA power increase (to support DP)	20%
	DP FMA power decrease (half as many DP units)	50%
	GPU peak power with CDF + HFMA	234W
CDF + HFMA + DVS	Execution units power reduction (including additional pipeline stages overhead)	35%
	GPU peak power with CDF + HFMA + DVS	215W

excessively and comparing estimated power consumption of our integrated McPAT-GPGPUSim simulator and measuring the actual power consumption of a GTX 480 card. Overall, our power estimates have an error of less than 15% compared to the measured GTX480 card power running real benchmarks [58]. A breakdown of the power consumption of major GPU components is

shown in Figure 6.10. The SP and DP FPUs can consume nearly 35% of the total GPU peak power while the IALUs consume about 4% of peak GPU power.

Although the average power consumption of the GPU components can be very different because these percentages depend on the activity factors, the power constraint is typically set by the peak power consumption of the components [67].

6.7.1 POWER OVERHEAD ESTIMATION:

I synthesized Verilog HDL descriptions of the FMA units with different pipeline depths to estimate the power impact of increasing the pipeline depth by 2×. The timing and power estimation of synthesized Verilog was performed using Synopsys DesignVision® and PowerCompiler®. Synthesis was performed using TSMC's 40-nm standard-cell library. The enhanced SP HFMA units that can also support DP operations are also synthesized to estimate their power overhead.

I utilized McPAT to estimate the power overhead of FBs and ARFs that are employed in CDF and HFMA [68]. I also incorporate wire power for accessing the FB and ARF using the same wire capacitance per unit length as Gebhart *et al.* [37]. The length of the wires is assumed to be one half the height of an SP FMA unit. This is because the FB and ARF can be laid out next to the FMA units. The height of FMA is estimated assuming a square layout for an SP FMA unit [69]. The power overhead of DFN and each of our proposed approaches is summarized in Table 6.4.

CDF provides a significant performance improvement at a considerably low power overhead of 6W (2%). With HFMA, the ARF increases GPU power consumption by less than 1%. However, since I also reduce the DP FMA units when HFMA is employed, I achieve a significant reduction in the peak power of the GPU (more than 6%). Finally, with voltage scaling, I further

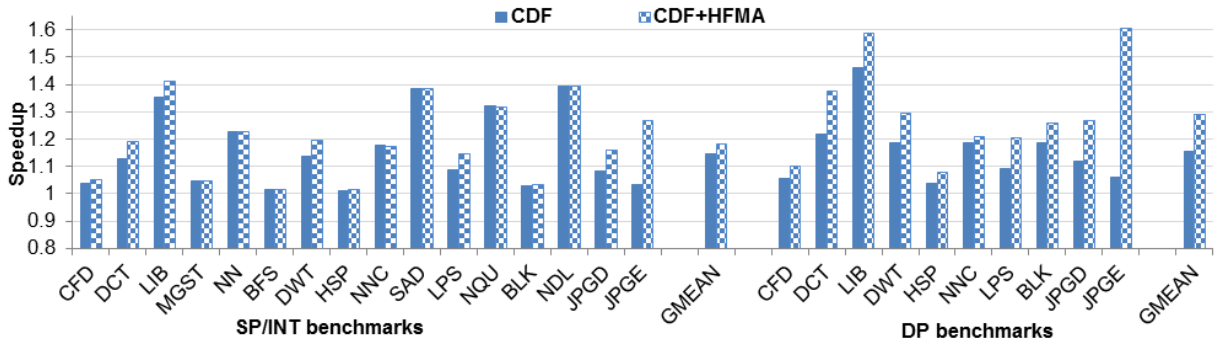


Figure 6.12: Speedup for SP/INT and DP benchmarks with CDF and HFMA

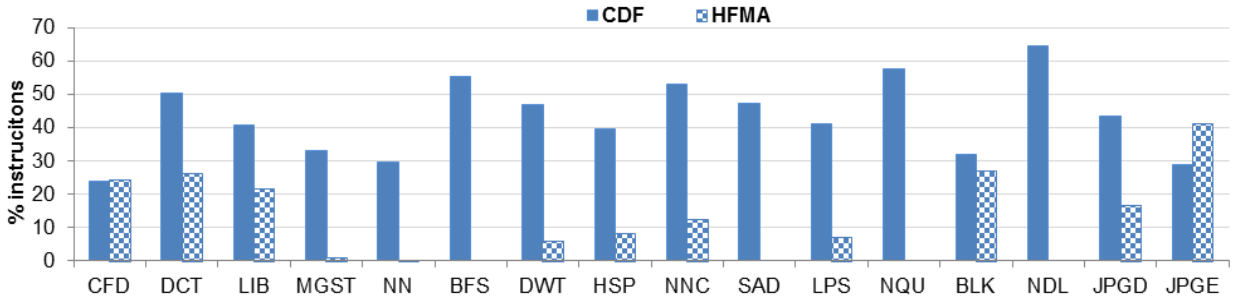


Figure 6.11: Percentage of total instructions that benefit from CDF and HFMA

reduce GPU power by 14%.

6.7.2 PERFORMANCE IMPACT

Figure 6.12 shows the speedup achieved with CDF and HFMA. The percentages of total integer and FP instructions that can execute with reduced effective latency are shown in Figure 6.11. In general a higher percentage of CDF and HFMA instructions leads to higher speedup. However, for some cases a high percentage of CDF instructions does not translate into comparable performance improvement. For example, although DCT and BFS (SP/INT) have more CDF instructions than LIB, they have comparatively less speedup. This is because both DCT and BFS have relatively low percentage of RAW stalls in the baseline GPU architecture (Figure 6.1).

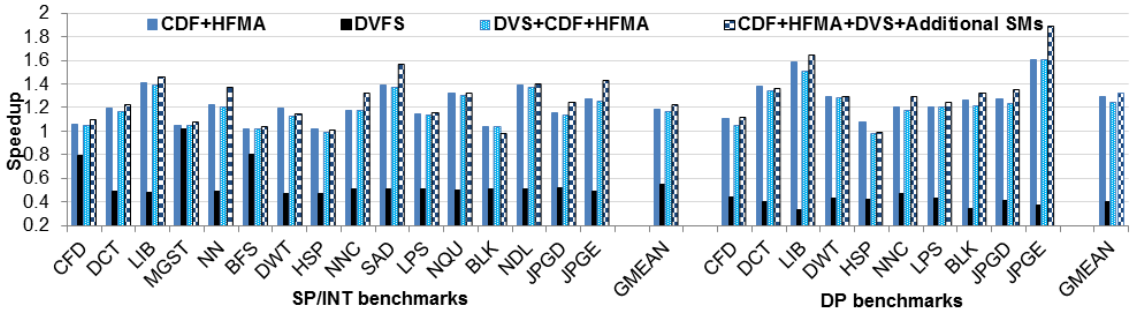


Figure 6.13: Performance impact with DVFS, DVS (with CDF and HFMA) and with additional SMs within the same power constraint

BLK and HSP experience a considerable percentage of RAW stalls. However, these RAW stalls are caused by long latency operations that execute on the SFU (e.g. log, FP div, sin etc.). Since, our approaches do not impact SFU instructions; these benchmarks do not exhibit a high speedup. For SP and INT benchmarks, CDF provides a speedup of 15% (geometric mean) while the speedup with both CDF and HFMA is 18% (geometric mean).

Since the DP RAW latencies are even longer in NVIDIA architectures (48 cycles) [70], our approaches provide more performance improvement for DP benchmarks than SP benchmarks. This is because for DP operations both CDF and HFMA provide a greater latency reduction. While HFMA allows single-cycle latencies for MACC and AACC instructions, CDF reduces the RAW latency to 18 cycles (with DVS) [71]. The geometric mean speedup for DP benchmarks is

Table 6.5: Summary of power and performance impact

	Peak GPU power	SP/INT Speedup (GMEAN)	DP Speedup (GMEAN)
CDF	256W	1.15	1.15
+HFMA	234W	1.18	1.29
+DVS	215W	1.16	1.25
+Additional SMs	250W	1.23	1.33

15% and 29% for CDF and CDF+HFMA, respectively.

Figure 6.13 shows performance impact of DVFS, dynamic voltage scaling (DVS) and activating additional SMs within the same power constraint as the baseline GPU. CDF+HFMA is also shown in the figure for performance comparisons. DVS utilizes additional pipeline stages to allow the same throughput as the baseline GPU. Both CDF and HFMA are implemented with DVS and when additional SMs are activated. Due to throughput reduction and latency increase, DVFS results in considerable performance reduction. Since DP operations already have a considerably higher latency in the baseline GPU than SP operations, with DVFS the further increase in latency affects DP benchmarks even more.

With DVS, I address both the throughput and latency issues of voltage scaling. Specifically, the throughput is maintained by increasing the number of pipeline stages, while CDF and HFMA greatly reduce the impact of long execution latencies. Consequently, even when the voltage is reduced for power reduction, the performance of DVS (compared to CDF+HFMA) does not decrease considerably for most benchmarks. Finally, for the same power constraint as the baseline GPU, the GPU can activate two additional SMs after applying DVS. The additional SMs further improve the performance of benchmarks (21% and 33% overall geometric mean improvement for SP/INT and DP benchmarks, respectively). A summary of the performance and power impact of our proposed techniques is shown in Table 6.5.

6.8 CONCLUSION

The additional pipeline stages in GPUs to allow efficient hardware resource sharing increase RAW latencies in GPU architectures. Moreover, the high power cost associated with DFNs pre-

cludes the implementation of DFNs in GPUs. Consequently, dependent instructions in GPUs have long RAW latencies that impede the performance of many GPGPU applications. I propose low-overhead alternatives to data forwarding (CDF and HFMA) that improve the performance of SP/INT and DP GPGPU applications by 18% and 29%, respectively. My proposed approaches also allow considerable reductions in the peak power consumption of the GPU by re-using SP HFMA units for DP operations (a peak power reduction of 6%).

Finally, I demonstrate that my proposed approaches can be incorporated with voltage scaling to significantly reduce peak power consumption of the GPU without any significant performance degradation. The reduced peak power can be used activate more SMs in the GPU within a power constrained GPU to improve overall performance by 33%.

7. POWER-EFFICIENT GPU ARCHITECTURE

This chapter proposes techniques for improving the power efficiency of GPU architectures. These techniques include; (i) formation of composite instructions that can accelerate integer execution, (ii) exploiting computational redundancy to improve instruction throughout, and (iii) a bit-sliced GPU architecture that improves performance of instructions with low bit-width operands. All of the proposed techniques exploit common instruction and data patterns experienced in GPGPU applications. While the techniques are proposed and evaluated for a particular GPU architecture, they are architecture-independent and can be incorporated in most single-instruction multiple-threads (SIMT) and single-instructions multiple-data (SIMD) architectures.

Composite instructions: GPUs are optimized for floating-point (FP) intensive 3D graphics. Consequently, the execution pipelines in GPUs are typically composed of FP fused multiply-add (FMA) units [72, 73, 74, 75]. However, the massive parallelism and high memory bandwidths of GPUs are also utilized to accelerate integer intensive applications. Some integer intensive applications include data compression [76], data encryption [77], image processing and medical imaging, and processor temperature analysis (hot spot) [78]. To support integer instructions, the FMA unit is enhanced with logic to perform integer arithmetic, bitwise and logical operations to allow integer instructions to utilize the same pipeline. I exploit the resources of FMA units to combine pairs of integer instructions into composite instructions that can be efficiently mapped to the FMA pipeline. These composite instructions are formed by the compiler and thus they reduce the total number instructions in a kernel. Composite instructions directly improve the fetch-decode-schedule (FDS), execute (EX) and main register file (MRF) power consumption of the streaming multiprocessor (SM) and also provide a significant performance improvement.

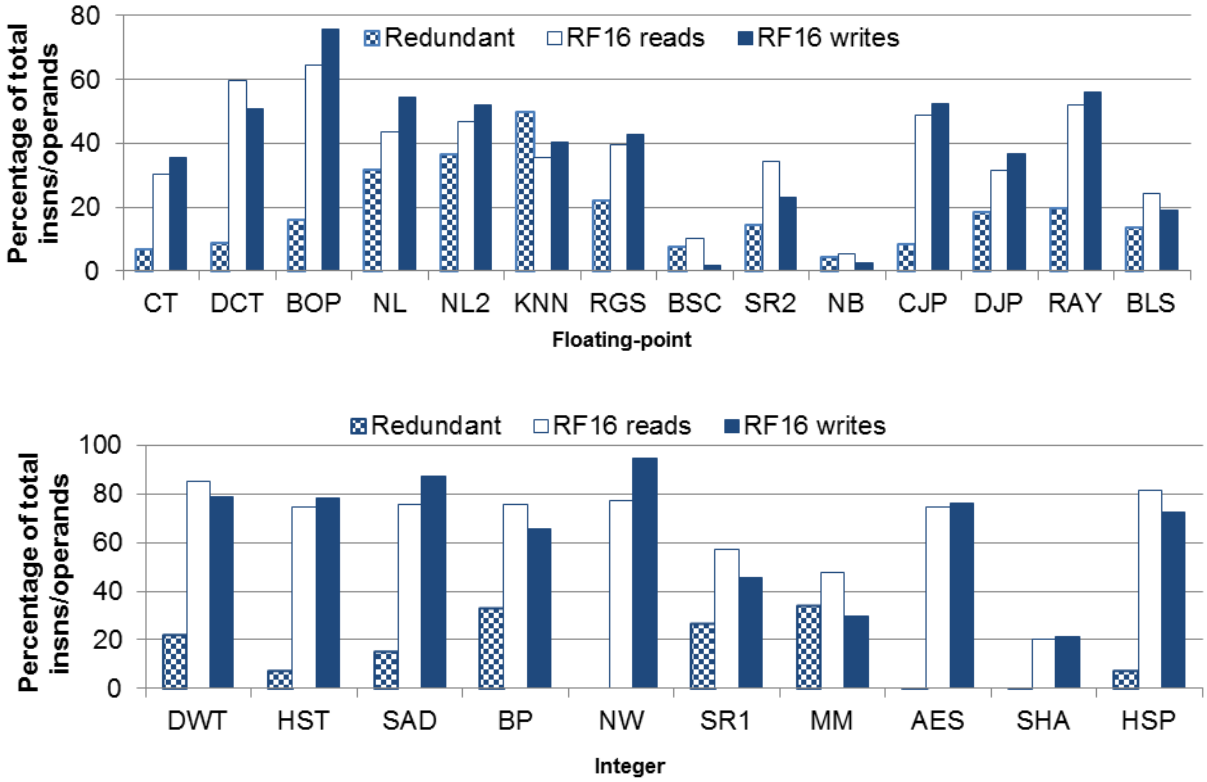


Figure 7.1: Percentage of total instructions that performed redundant computations and the percentage of instructions that utilize 16 bits or less for RF reads and write

Computational redundancy: Many GPGPU applications exhibit considerable computational redundancy. This redundancy arises when an instruction produces the same results across all the threads in a SIMT group. The *Redundant* bar in Figure 7.1 shows the number of SIMT instructions that produced the same results across all the co-issued threads (warp). For many GPGPU applications, redundant computations constitute from 10% to 50% of the overall dynamic instructions. The sources of this computational redundancy are the duplicated control instructions in different threads, operations with constants, memory address calculations and the inherent redundancy in the pixel data in image and video processing applications.

This work exploits the computational redundancy within a warp, by dynamically detecting in-

structions that produce the same results across all the threads in the warp. These instructions are then issued to a separate scalar pipeline and their source and destination registers are kept in a separate scalar RF. The scalar pipeline can provide significant performance improvement and also reduces the energy consumption by eliminating redundant computations and reducing main register file (MRF) accesses. The baseline GPU assumed in this study (NVIDIA's® Quadro™ FX5800) can issue up to two instructions (one each to the FP and special functional unit (SFU) pipeline) per thread every cycle [70]. I utilize the same dual-issue capability for my approach except I use it to issue computationally redundant instructions to a scalar unit for power efficiency improvements.

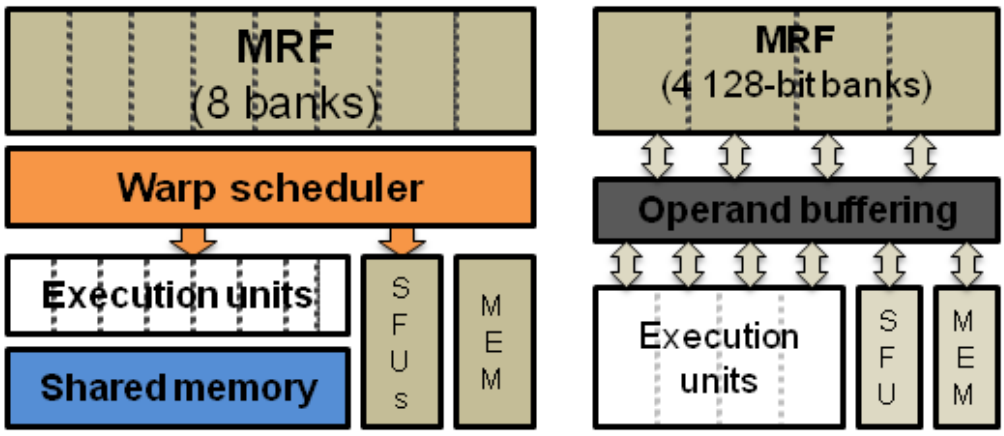
Sliced architecture: GPUs are optimized to perform operations on 32-bit data values. However, many operands do not completely utilize the 32-bit resources of the SM pipeline and MRF. The *RF16 reads* and *RF16 writes* bars in Figure 7.1 show the percentage of MRF accesses that required 16 or fewer bits for complete representation. The MSBs for these operands only contain sign-extended bits. A large percentage of accesses have more than 16 sign-extended bits and do not require the complete 32 bits of registers. This data trend can be utilized to reduce the MRF access energy and improve instruction throughput by slicing the MRF banks and the execution data-path into two 16-bit wide slices. This allows the GPU to issue two instructions with 16-bit wide operands instead of a single instruction that reads 32-bit operands, effectively improving the instruction throughput and the energy consumption of the MRF and execution units. While some GPUs provide support for 16-bit data types, these data types are used only for storage (i.e. to reduce memory bandwidth pressure) [79]. Moreover, the use of these data types requires the programmer to explicitly specify the proper data-type. My approach dynamically detects short

bit-width operands and schedules them on half bit-width slices.

The novel contributions described in this chapter include improving the performance and power efficiency of GPUs by:

- Fusing dependent *integer* instructions to efficiently utilize the register file bandwidth and the FMA unit resources. (Section 7.2).
- Exploiting computational redundancy with a low-power scalar pipeline (Section 7.3).
- Utilizing a sliced architecture for high computational throughput by exploiting low bit-width operands (Section 7.4).

Sections 7.5 through 7.5.4 present a detailed performance and power evaluation of the proposed approaches. Section 7.8 discusses how the proposed techniques can be implemented in different GPU architectures. Conclusions are given in Section 7.7.



(a) (b)

Figure 7.2: Baseline GPU architecture: (a) Streaming multiprocessor (SM), (b) SIMT cluster

7.1 BASELINE GPU ARCHITECTURE

In this work, I assume a baseline GPU architecture similar to the Nvidia® Quadro FX5800 GPU[70]. The baseline GPU consists of 30 SMs (Figure 7.2(a)), each containing an MRF providing 16384 32-bit registers, a warp scheduler, an 8-wide array of execution units, 2 SFUs, constant and texture memories, and shared memory. The resources of an SM are organized into two SIMT clusters (Figure 7.2(b)). Each SIMT cluster has a set of four execution units and four MRF banks. Each MRF bank is 128 bits wide with 512 entries and is dual-ported (1-read port, 1write port).

The multi-banked MRF architecture allows each SM to sustain a read bandwidth of 24 32-bit reads and 8 32-bit writes per cycle per SM without employing a multi-port register file, which can consume considerably more area and power than a single-port register file. This bandwidth ensures that in the absence of any bank conflicts, eight single precision FMA instructions, each of which requires three 32-bit input operands and writes one 32-bit result, can be issued every cycle in an SM. All registers of a thread reside in the same bank and each thread performs multiple MRF accesses to read all of its operands [70].

Each SM schedules threads in groups of 32 threads called warps. With 8-wide execution units, each warp is issued over 4 cycles. The execution units are typically FP FMA units that have been enhanced to also execute integer instructions [72, 73, 74, 75]; the enhanced FMA units execute both the integer and FP instructions excluding transcendental functions, which are executed on the SFUs. I assume that the pipeline latency of FMA units is 8 cycles [74]. Consequently, most instructions (excluding double precision and transcendental instructions) have an 8 cycle read-after-write (RAW) latency [72, 73, 74]. Finally, the warp scheduler can issue up to two instruc-

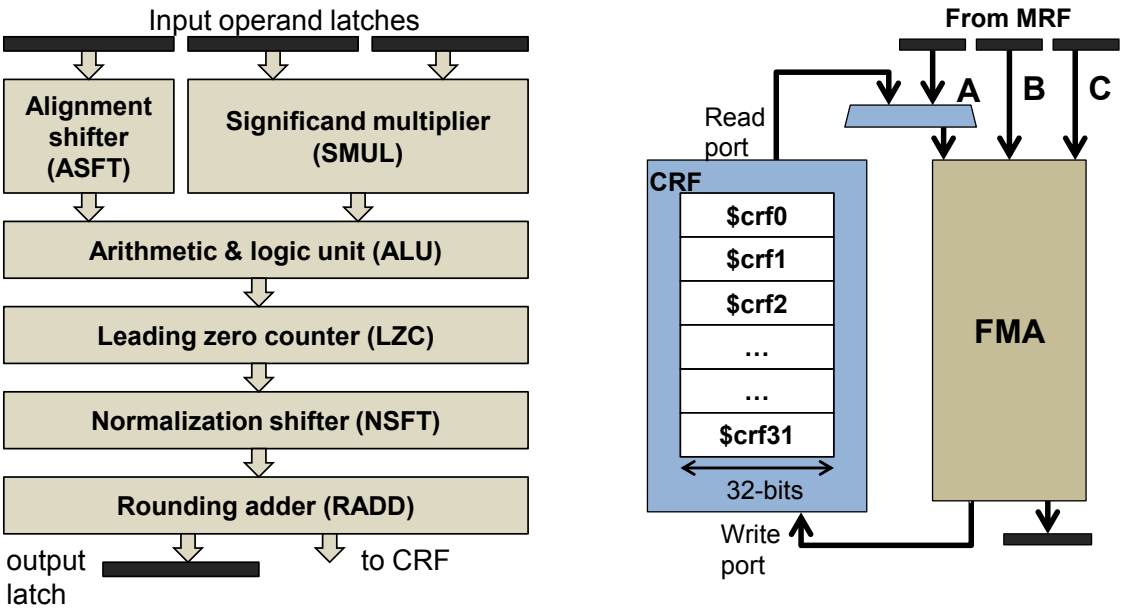


Figure 7.3: SM execution unit: a) A baseline FMA unit b) A composite register file (CRF) integrated with an FMA unit

Composite instruction	Resource utilized
Multiply-shift	SMUL-ALU
Add/logic-shift	ALU-NSFT
Add/logic-add	ALU-RADD
Shift-add/logic	ASFT-ALU
Muliply-cvt	SMUL-LZC-NSFT-RADD
Add/logic-cvt	ALU-LZC-NSFT-RADD
Shift-shift	ASFT-NSFT

Table 7.1: Composite instructions and the corresponding FMA resources utilized

tions (one to the execution units and one SFU instruction) per warp in a cycle [70].

7.2 FUSING INSTRUCTIONS WITH FMA UNIT

7.2.1 FMA UNIT

An FMA unit with integer execution enhancements is shown in Figure 7.3(a). For FP FMA

operations, the FMA unit multiplies the significands of the FP operands using the signifand multiplier (SMUL) and aligns the product to the addend using the alignment shifter (ASFT). The alignment shifter ensures that both the product and the addend have the same exponent. The shifted product is added to the addend using the ALU and the result is left-shifted using the normalization shifter (NSFT) until all the leading zeros are shifted out. The leading zero counter (LZC) is used to determine the amount of the normalization shift. The normalized result is rounded using the rounding adder (RADD) and is re-normalized if required (in case of overflow during rounding).

The FMA unit in the baseline GPU is also used to perform integer arithmetic operations by including support for integer logical and bit-wise operations along with the adder (ALU) in the FMA data-path. The ASFT and NSFT are used to perform right and left shifts on integer operands, respectively. The LZC, NSFT and RADD blocks are also used for conversion between FP and integer data types and vice-versa. Finally, 24-bit integer multiplication is performed using SMUL. These integer execution enhancements are present in the baseline GPU that utilizes the same FMA unit for both integer and FP operations [72, 73, 74, 75].

7.2.2 MOTIVATION AND APPROACH

I propose to re-use the resources provided by the FMA unit to issue composite integer instructions formed by combining dependent integer instructions. The dependent instructions are chosen such that they can easily be mapped to the resources within the FMA unit. These composite instructions are added to the instruction set architecture (ISA) of the GPU. Adding instructions to a GPU ISA is much simpler than adding them to a CPU ISA, since GPUs typically utilize an intermediate language (e.g., NVIDIA's PTX or AMD's HSAIL) that is converted to the final GPU

ISA by a platform-specific compiler that is included in the GPU driver. The identification of dependent instruction pairs and formation of composite instructions is performed statically at compile time. The motivation is to increase the amount of work done per instruction without a significant power overhead by performing composite operations within the same FMA pipeline. This reduces both the number of instructions fetched and scheduled and the thread cycles required for execution.

Since the MRF bandwidth is optimized to read *three* 32-bit operands per cycle per FMA unit and most integer instructions require only two or fewer operands per instruction, the GPU can combine two dependent integer operations such that they completely utilize the full RF read bandwidth. A composite register file (CRF) is added with each FMA unit to hold the intermediate result (i.e., result of the parent instruction in the combined pair) of the composite instruction. Dependent instructions (other than the combined instruction) read the source operand value from the CRF. Figure 7.3(b) shows the 32-entry, 32-bit CRF, integrated with the FMA pipeline. The CRF is dual-ported (1-read, 1-write) and has a dedicated slot for storing one intermediate result per thread. It is indexed by the *warp id*. The 32-entry CRF ensures that one intermediate result can be stored for each active warp (the maximum number of active warps is 32). The use of the CRF also reduces the number of accesses to the MRF, which has a much larger energy cost; the MRF has 16384 entries in each SM.

7.2.3 IMPLEMENTATION DETAILS

The combinations of different dependent integer instructions that can be mapped easily using the FMA unit are shown in Table 7.1, along with the FMA resources that are used for each composite instruction. Composite instructions can be formed by the compiler or the GPU finalizer.

PC	Instruction
0x218	xor $r0 <= $r2,$r0
0x220	shr $r0 <= $r0,0x1f
0x228	sub $r5 <= $r5,$r4
0x230	xor $r4 <= $r4,$r5
0x238	neg $r6 <= $r4
0x240	add $r0 <= $r0,$r4

(a)

PC	Instruction
0x218	xor_shr $r0<=$r2,$r0,0x1f
0x220	sub $r5 <= $r5,$r4
0x228	xor_add $r0<= $r4,$r5,$r0
0x230	neg $r6 <= $crf[warp_id]

(b)

Figure 7.4: Formation of composite instruction: a) Original instruction sequence b) New sequence with composite instructions

The GPU finalizer transforms compiled code into GPU-specific instructions. Once code generation is complete, the compiler/GPU-finalizer analyzes the instructions ad their dependencies and replaces dependent integer instructions with a single composite instruction. I enhance the FMA unit to implement bidirectional shifters for both ASFT and NSFT. This increases the number of composite instructions by allowing both the producer and consumer instructions to be either left or right shifts.

In order to form composite instructions, the compiler performs an additional pass over the *PTX intermediate language* generated by the NVIDIA compiler. The compiler pass identifies the integer operation chains shown in Table 7.1. Since the instructions are combined at the compiler level, the intermediate results, which are required by other instructions, are read from the dedicated CRF.

During the formation of composite instructions, the compiler ensures that at most only one of the instructions in the composite instruction performs memory accesses to avoid pipeline stalls due to shared memory bank conflicts. Moreover, branch targets and branches are never crossed during the search for instructions that can be combined. Finally, the compiler pass also ensures

that the CRF entry is no longer needed by any dependent instruction before allowing another composite instruction to overwrite it.

Figure 7.4(a) shows a code snippet from the DCT benchmark. The compiler is able to detect two composite instructions (highlighted in the figure) from the given code sequence. Each pair of instructions that can be combined is changed into a single composite instruction (Figure 7.4(b)). The two composite instructions utilize the FMA resources ALU-NSFT and ALU-RADD, respectively. For the second composite instruction (PC: 0x228 in Figure 7.4(b)), the intermediate result is stored in the CRF entry corresponding to the warp identification number. The instruction at PC 0x230 thus reads its operand from the CRF.

7.3 EXPLOITING COMPUTATIONAL REDUNDANCY

7.3.1 MOTIVATION AND APPROACH

Many applications execute a significant number of instructions in which all the threads in the warp produce the same result. There are several reasons for this redundancy. First, each thread typically consists of two types of computations, i.e., control and data. While the data computations can be different across different threads, the control computations (e.g. loop increments, memory address calculations) are often the same across different threads. Secondly, even for data processing, some applications exhibit data redundancy. This is true in particular for image and video processing applications where different pixel values may not change in nearby image regions that are processed by a warp.

Moreover, computations involving constants also increase the probability of redundant computations, since one of the operands is constant across all the threads. Consequently, several image processing benchmarks that utilize many constants have a high percentage of redundant compu-

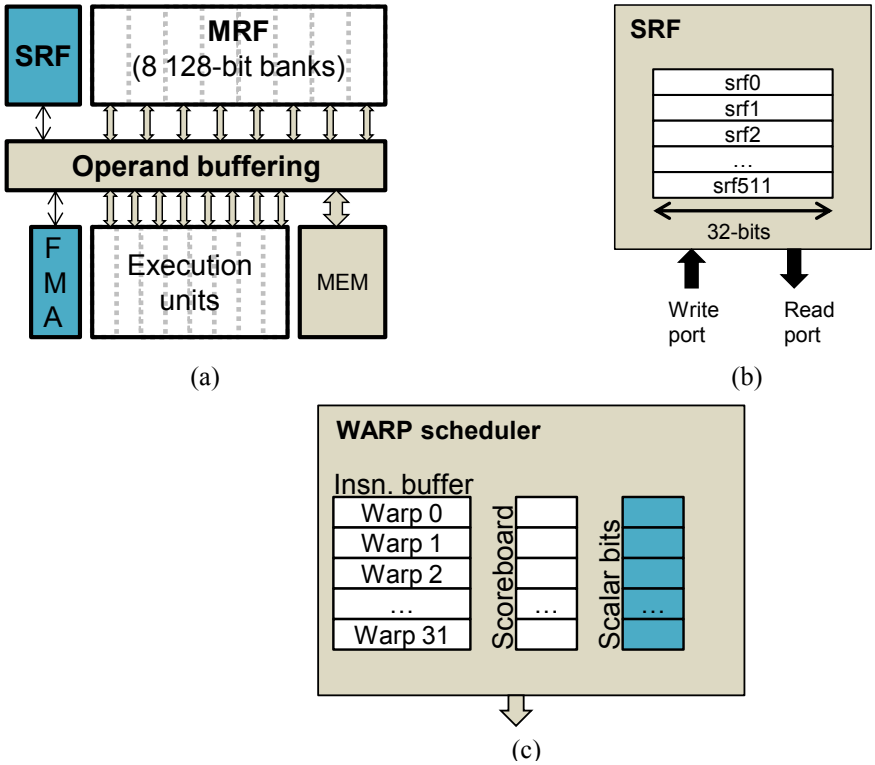


Figure 7.5: SM modifications for scalar instructions: (a) Modified SM (b) Scalar register file (SRF) (c) Scheduler modifications

tations (e.g. *knn* in Figure 7.2).

I exploit the computational redundancy to increase instruction throughput by adding a separate scalar unit with the SIMT unit. The scalar unit, which contains a scalar register file (SRF) and a single FMA unit is highlighted in Figure 7.5(a). If each input operand of an instruction is the same for all threads in the warp, the instruction is issued to the scalar unit; otherwise it utilizes the normal SIMT units. Although the baseline GPU does not have a scalar unit, Intel's Larrabee and AMD's recently announced GPUs also include a scalar unit[80, 75]. The overhead of my approach will thus be lower for such GPUs. The scalar unit is an independent unit that can be power gated for applications that do not exhibit significant computational redundancy.

7.3.2 Implementation

In order to detect redundant operations, I keep track of warp registers (32 32-bit physical regis-

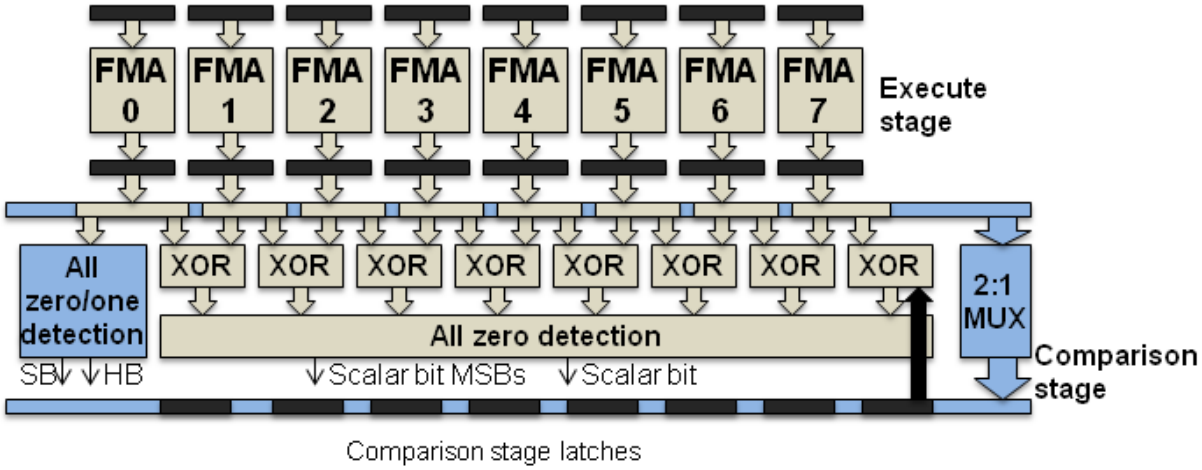


Figure 7.6: Comparison stage added to the execution pipeline

ters) that have the same value for all 32 physical registers. This is achieved by adding a comparison stage before the write-back stage in the SIMT pipeline. The comparison stage checks to see if the register write-back values across all the threads in the warp are the same. If all values are the same, the scalar bit is set for the corresponding register in the warp scheduler (Figure 7.5(c)) and a single 32-bit result is written to the SRF.

The SRF has a single bank with 512 32-bit entries (Figure 7.5(b)). Operands for a scalar instruction are read over multiple cycles, similar to the SIMT pipeline. This keeps the two pipelines in sync and also allows a 1-read, 1-write port SRF to provide sufficient bandwidth for scalar instructions. Threads in a kernel have common logical register values but each thread's registers are mapped to different physical registers in the MRF banks. The SRF registers are addressed using the logical register value of a thread.

Specifically, the SRF is indexed by the value *warp_id*kernel_register_count + logical register_id*. Since the *kernel_register_count* is known at compile time, the first indexing term

(*warp_id*kernel_register_count*) can be preloaded in an SM during the kernel launch phase. During warp scheduling, the scalar bits for all operands are checked in parallel with the score-boarding logic to determine the unit (SIMT or scalar) to which the instruction should be issued. If all the source operands of an instruction reside in the SRF, it is issued to the scalar unit; otherwise it is issued to the SIMT pipeline.

The comparison stage added to the SIMT unit is shown in Figure 7.6. A single stage of XOR gates is used to compare write-back values from adjacent SIMT lanes. Since a 32-thread warp is scheduled in *SIMT groups* of eight threads, one XOR block is used to compare the current write-back value with one of the write-back values of the previous cycle. The write-back value of the previous cycle is read from the pipeline latches. The results from the XOR gates are applied to the all-zero detection circuit, which typically exhibit delay equivalent to one or two gates using a dynamic circuit [81]. The all-zero detection circuit can also be implemented with static gates with a few logic gate stages. The all-zero detector sets the scalar bit to 0 if there is any binary 1 present in the inputs bits, indicating that the write-back values are different. Otherwise, the scalar bit is set to 1. The registers for SIMT threads are changed to a single scalar register if the scalar bit is set for all 4 SIMT groups of a warp.

The additional scalar unit provides three advantages. First, a considerable portion of the high-energy MRF accesses are replaced with low-energy SRF accesses. The low access energy for the SRF is due to its considerably smaller size (32 bits,1 bank, 512 deep vs. 128 bits, 8 banks, 512 deep of the MRF). Second, the computation is carried out only once instead of for each of the 32 threads in the warp, reducing execution energy. Third, significant performance improvement can be obtained through the co-issue of instructions to the scalar and SIMT pipelines. The baseline

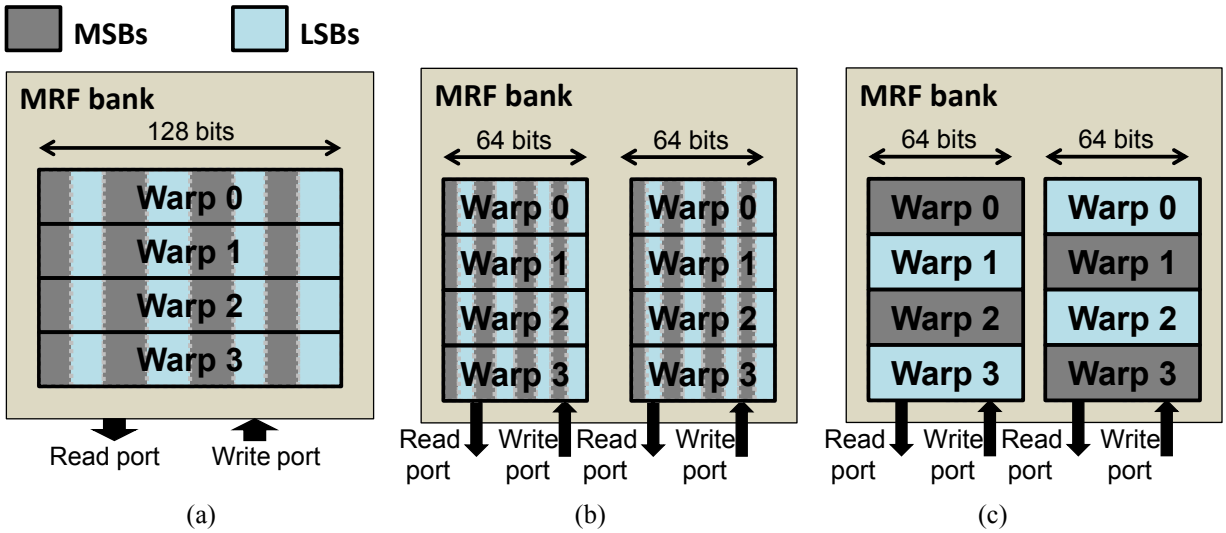


Figure 7.7: MRF bank sub-division to improve instruction throughput: (a) Baseline MRF bank (b) Sliced MRF bank (c) Sliced MRF bank with modified data layout

SM can already issue two instructions per warp (one to the FMA units and one to SFU) [70]. With my approach, I utilize the same dual issue logic to issue one instruction to the scalar unit and one to the SIMT units.

7.4 BIT-WIDTH SLICING

7.4.1 MOTIVATION AND APPROACH

The MRF can consume up to 15% of the SM dynamic power[37]. To reduce the per-access energy consumption of the MRF, I propose to slice each MRF bank into two banks, such that each 32-bit register within the 128-bit physical register of an MRF bank is split into two separate banks that are half as wide as the MRF bank. The motivation of this partitioning is that many operations only need 16 bits or less per operand. In particular, for some integer applications as many as 80-90% of the register file read and write accesses utilize less than 16 bits of the register (Figure 7.1). The 16 MSBs of each register in such cases only contain the sign-extended bits and

do not impact the results of computations.

7.4.2 IMPLEMENTATION

The Sliced MRF architecture is shown in Figure 7.7(b). A Sliced MRF bank is half as wide as the MRF bank (from 128 bits in Figure 7.7(a) to 64 bits in Figure 7.7(b)). Consequently the read and write access energy of operands that require at most 16 bits is cut by nearly one half. However, in order to determine if an operand only requires a half bit-width access, two additional bits are required for every warp register (32 32-bit physical registers). A half bit-width (HB) bit indicates that only the 16 LSBs need to be read for a given operand register and a sign-extension (SB) bit indicates the sign-extension bits of the half bit-width register. These bits are stored in the warp scheduler, similar to the scalar bits (Section 7.3). When an instruction is scheduled, the HB and SB bits are checked to determine whether or not both MSB and LSB banks need to be accessed.

During execution, the HB write-back values are detected using an *all zero/one detection* (ZOD) module in the comparison stage (Figure 7.6). The ZOD checks if there are only ones or zeros in the 16 MSBs of the result of one thread. If that is the case, and all the MSBs are the same across all the threads (scalar bit MSBs is set), the HB bit is set along with the SB bit according to the sign-extended bits value. The SB bit is based upon whether a one or zero was detected. The *scalar bit MSBs* is computed during the generation of the *scalar bit* and is set if the 16 MSBs are the same across all threads.

The Sliced MRF can also be utilized to increase the throughput of instructions with HB operands (sliced instructions). Specifically, the baseline architecture allows a 96-bit read and 32-bit write bandwidth per FMA[37]. Since the HB operands can only utilize up to 48-bit read band-

Table 7.2. Simulator configuration

# of SMs	30
SM Freq	1.30GHz
On-chip Interconnect Freq	0.65GHz
Warp Size	32
SIMD Width	8
# of Threads per SM	1024
# of CTAs per SM	8
# of Registers per SM	16384
L1$ Memory per SM	16 KB
# of Memory Channels	8
Warp Scheduling	Round Robin

Table 7.3: Benchmarks and their acronyms

Benchmark	Abbreviation	Benchmark	Abbreviation
Texture-based convolution [91]	CT	Ray tracing [86]	RAY
Discrete cosine transform [91]	DCT	Simple CUDA BLAS [91]	BLS
Binomial options pricing [91]	BOP	Discrete wavelet transform [91]	DWT
Image denoising [86]	NL	Image histogram [91]	HST
Image denoising [86]	NL2	Sum-of-absolute-differences	SAD
Image denoising [86]	KNN	Back-propagation [78]	BP
Recursive Gaussian filter [91]	RGS	Needleman-Wunsch [78]	NW
Black-Scholes options pricing	BLK	SRAD computation 2 [78]	SR1
SRAD computation 1 [78]	SR2	Matrix multiplication [91]	MM
n-body simulations [91]	NB	AES encryption [86]	AES
JPEG encoding [86]	CJP	SHA encryption [86]	SHA
JPEG decoding [86]	DJP	Hot-spot [78]	HSP

width and 16-bit write bandwidth, the GPU can dual-issue sliced instructions from two different warps using sliced FMA units that allow two parallel data-paths with half bit-widths. The parallel data-paths can be combined to form a single data-path with the complete 32-bit bit-width [82, 83].

Although the sliced instructions only utilize half of the read and write bandwidths, dual-issue of sliced instructions from two warps is difficult since all the LSBs reside in the same Sliced MRF bank (Figure 7.7(b)). This prevents two warps from dual-issuing sliced instructions because of potential MRF bank conflicts. In order to reduce these bank conflicts, I swap the LSB and MSBs of threads belonging to odd numbered warps. This is achieved by adding multiplexers

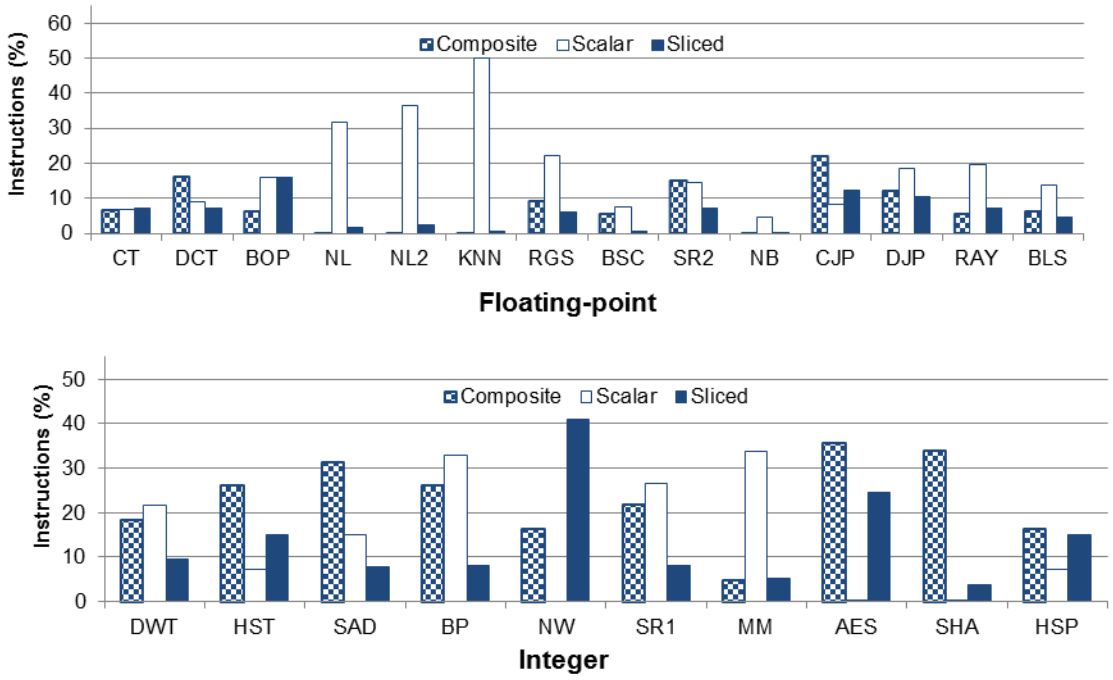


Figure 7.8: Percentage of total instructions that benefit from each of the proposed techniques

in the comparison stage (Figure 7.6). The modified data layout (Figure 7.7(c)) allows sliced instructions to issue from odd and even warps in the same cycle. Instead of a unified warp scheduler, I divide the warp scheduler into two, half as complex schedulers such that one schedules instructions from the odd warps and the other from even warps. If both schedulers have sliced instructions that are ready to be issued in a given cycle, they are issued simultaneously.

When the result of a sliced instruction requires the complete 32-bit write bandwidth, the pipeline is stalled for a cycle. However, in practice such stalls are rare (0.1% (geometric mean) of execution unit accesses caused a stall across all benchmarks). Moreover, these stalls do not reduce the instruction throughput below the baseline since each SM still executes 8 instructions per cycle even if all sliced instructions create stalls. In the absence of stalls, the throughput of sliced instructions can be increased to 16 instructions per cycle per SM.

7.5 EVALUATION

I used GPGPU-Sim [66] to evaluate the performance impact of my proposed techniques. The simulator was configured to model a GPU similar to the Nvidia® Quadro FX5800. The simulator configuration is summarized in Table 7.2. GPGPU-Sim was enhanced to maintain access rates for different architectural components. These access rates were used in the GPU power model proposed by Hong *et al.* to estimate the impact of my techniques on power consumption [22]. The benchmarks used in this study and their acronyms are shown in Table 7.3.

7.5.1 PERFORMANCE IMPACT

Figure 7.8 shows the percentage of total instructions that benefit from each of my proposed techniques. For FP intensive benchmarks, the percentage of *composite* instructions is less than that for the integer benchmarks. This is because of the lower percentage of integer operations in FP benchmarks. For encryption benchmarks, up to 35% of instructions can be combined to form composite instructions. These benchmarks perform numerous shift and add operations that can be easily mapped as composite instructions.

The *scalar* instructions shown in Figure 7.8 represent the percentage of total instructions that were issued to the scalar unit. Image processing, machine learning and control-processing intensive kernels such as NL, NL2, KNN, SR2, DJP, BP, DWT and MM perform a large number of scalar operations. The reasons for their high computational redundancy are described in Section 7.3. Encryption algorithms (SHA and AES) exhibit the least amount of computational redundancy. This is because these algorithms perform many bit-level manipulation instructions that are different for each thread.

The number of dual-issued *sliced* instructions is also shown in Figure 7.8. The percentage of

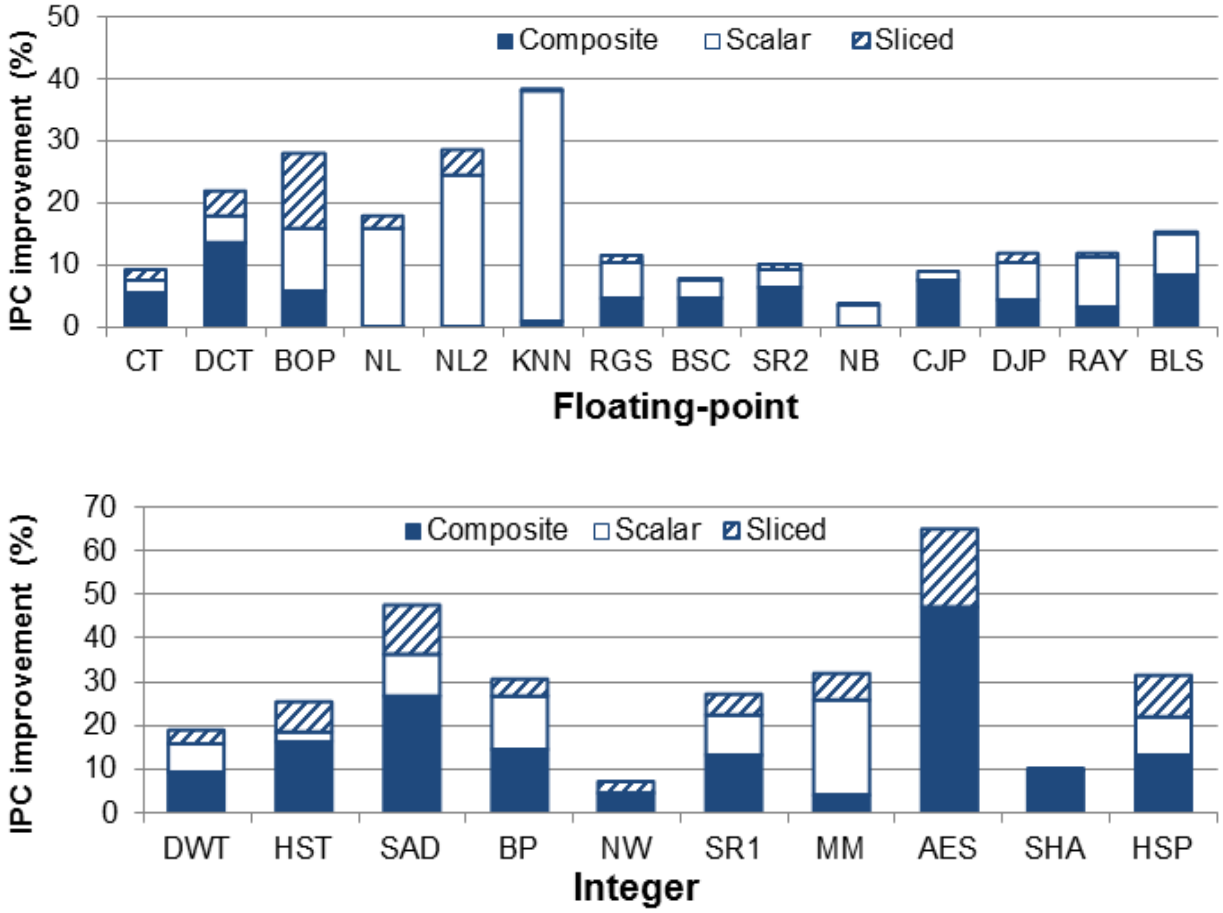


Figure 7.9: Performance impact of proposed approaches

half bit-width operands read and written from the MRF (Figure 7.1) is considerably higher, than the percentage of sliced instructions. This is because instructions can be dual-issued to the execution slices only if two HB instructions from odd and even warps are ready simultaneously in the same scheduling cycle.

Figure 7.9 shows the performance impact of my approaches for FP and integer benchmarks. Overall, the integer benchmarks exhibit higher speedups than the FP benchmarks. The speedup obtained generally correlates with the percentage of instructions that benefited from my techniques.

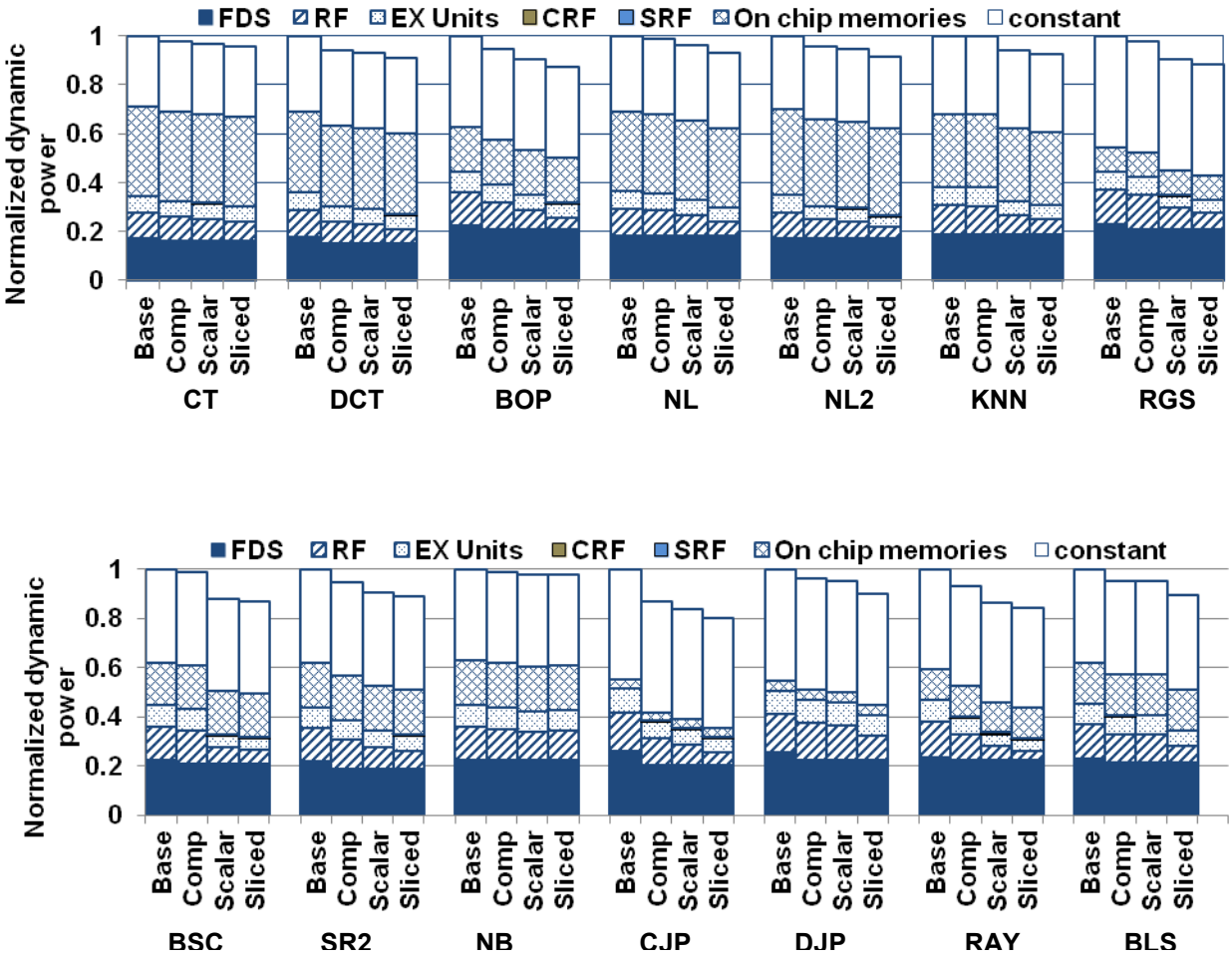


Figure 7.10: Dynamic power consumption per SM -- Floating-point benchmarks

However, in some cases (e.g. SHA) the memory-intensive or branch-divergent nature of the application reduces the overall performance impact. Although NW has a high percentage of sliced instructions, it also has a highly branch divergent behavior. Its performance improvement is thus limited by branch divergence serialization.

Moreover, as my techniques are applied successively, the performance improvement of some benchmarks starts to saturate (memory latencies and arithmetic latencies start to become more visible). Consequently, for many benchmarks (e.g. RGS, CJPG, DJPG, SRAD2, RAY, BLAS

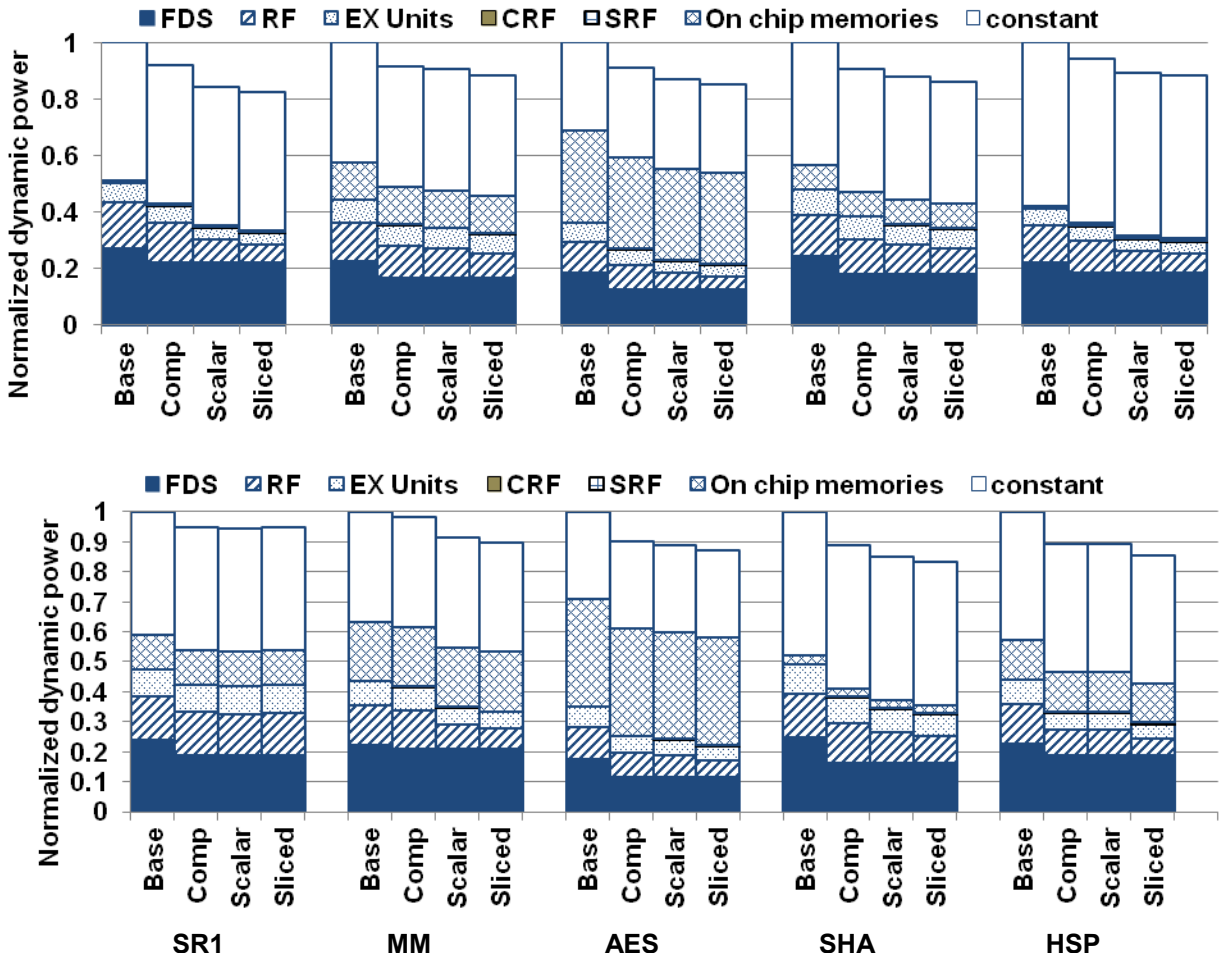


Figure 7.11: Dynamic power consumption per SM -- Integer benchmarks

and SRAD1) the relative performance improvement due to scalar and sliced techniques may be less than the relative number of instructions that benefited from them.

Overall, my approaches result in a geometric mean speedup of 25% and 12% for integer and FP benchmarks, respectively. Across all benchmarks, my approaches result in a geometric mean speedup of more than 15%.

7.5.2 SM DYNAMIC POWER

The breakdown of dynamic power consumption per SM is shown in Figure 7.10 and Figure

Table 7.4: Access energy and leakage power of register files

	MRF	CRF	SRF	Sliced MRF
Organization	8-banks, 128-bit words, 512 words	32-banks, 32-bit words, 32 words	1-bank, 32-bit words, 512 words	16-banks, 64-bit words, 512 words
Read energy (nJ)	$2.8x10^{-2}$	$7.1x10^{-4}$	$1.4x10^{-3}$	$1.4x10^{-2}$
Write energy (nJ)	$2.8x10^{-2}$	$7.4x10^{-4}$	$1.4x10^{-3}$	$1.4x10^{-2}$
Leakage power (per bank)	7.06 mW	0.28 mW	1.68 mW	3.54 mW

7.11 for FP and integer benchmarks respectively. All power results are shown for the baseline (base), composite (comp), scalar and sliced architectures. The *comp* architecture refers to an SM that supports composite instructions. The *scalar* architecture refers to an SM architecture that has support for both composite instructions and the proposed scalar unit. The *sliced* architecture represents an SM that incorporates all three of my techniques. The constant power shown in the power break-down models several components of an SM (e.g. frame buffers) that consume a relatively constant amount of power as long as the SM remains active [22]. My techniques do not impact constant power.

All of my proposed techniques can reduce the dynamic power consumption of SMs. Overall, there techniques achieve a 12% (geometric mean) reduction in SM dynamic power. The power reduction is higher for integer benchmarks (14% geometric mean) than FP benchmarks (11% geometric mean) since the integer benchmarks benefit more from my proposed approaches.

7.5.2.1 FETCH-DECODE-SCHEDULE POWER (FDS)

Composite instructions reduce the FDS power due to the reduction in the number of fetched/executed instructions. The number of instructions reduced is equal to the number of composite instruction executed by each SM (Figure 7.8). Overall, composite instructions reduce the FDS power by 15% (geometric mean). Since the integer benchmarks have a higher percent-

age of composite instructions, the FDS power reduction is higher for integer benchmarks than FP benchmarks.

7.5.2.2 MAIN REGISTER FILE (MRF) POWER

The dynamic energy and leakage power cost of MRF, CRF, SRF and Sliced MRF was obtained using CACTI[68]. A similar approach to estimating MRF power was taken by Gebhart *et al.* [37]. Although the leakage power and dynamic energy per access for CRF and SRF were included in the power model to estimate the overheads of my approaches, these structures do not contribute significantly to the overall SM and GPU power.

Table 7.4 shows the power and energy costs of the MRF, CRF and SRF as obtained using CACTI. The read and write energy cost per 32-bit register of CRF is nearly ten times less than the MRF due to its smaller size. Similarly, the SRF reduces the energy per access by nearly 20 times. Finally, for operands with bit-widths of 16 or less, the access energy is almost cut in half by the Sliced MRF architecture.

All of my proposed techniques reduce MRF power. Composite instructions reduce MRF accesses by accessing the CRF for intermediate result reads and writes. The scalar instructions utilize the SRF for redundant computations instead of the MRF. The Sliced GPU architecture reduces MRF energy consumption by reducing the bank widths and accessing half bit-width banks for a large percentage of operands. Overall my approaches reduce the MRF power consumption by 48% (geometric mean).

7.5.2.3 EXECUTION UNIT (EU) POWER

I enhanced the FMA execution units to include input multiplexers for reading from the CRF and swapping LSBs and MSBs of operands read from the Sliced MRF for odd warps. I also implemented bidirectional shifters for the ASFT and NSFT shifters. The estimated energy overhead

of these enhancements over the baseline FMA units is 2%. This estimate is obtained after synthesizing a Verilog descriptions of the baseline FMA unit and a modified FMA unit using Synopsys Design Compiler and the TSMC 45nm standard cell library.

For the scalar and sliced techniques, I also included a comparison stage before the write back stage in the SIMT pipeline. The overhead of this additional logic including the additional pipeline flip flops was estimated by synthesizing its Verilog descriptions using Synopsys Designe Compiler and the 45nm TSMC standard cell library. The energy overhead of additional logic is summarized in Table 7.5 along with the energy consumption of the execution units [69]. The comparison stage has a dynamic energy overhead of 4.3% over the execution stage (8 FMA units).

All three techniques proposed in the paper reduce EU power consumption. Composite instructions reduce the number of instructions issued to the EUs. Scalar unit reduces the dynamic power consumption of SIMT EUs nearly a factor of 32 for redundant computations. The sliced architecture also reduces the dynamic power of instructions with HB operands by nearly one half. Overall, my techniques reduce the EU power by 29% (geometric mean).

Table 7.5: Energy overhead of additional logic (the comparison stage).

	Dynamic energy (pJ)	**Leakage energy (pJ)**
Execution stage	28.48	9.76
Additional logic	0.72	.01
Overhead (%)	4.3	0.3

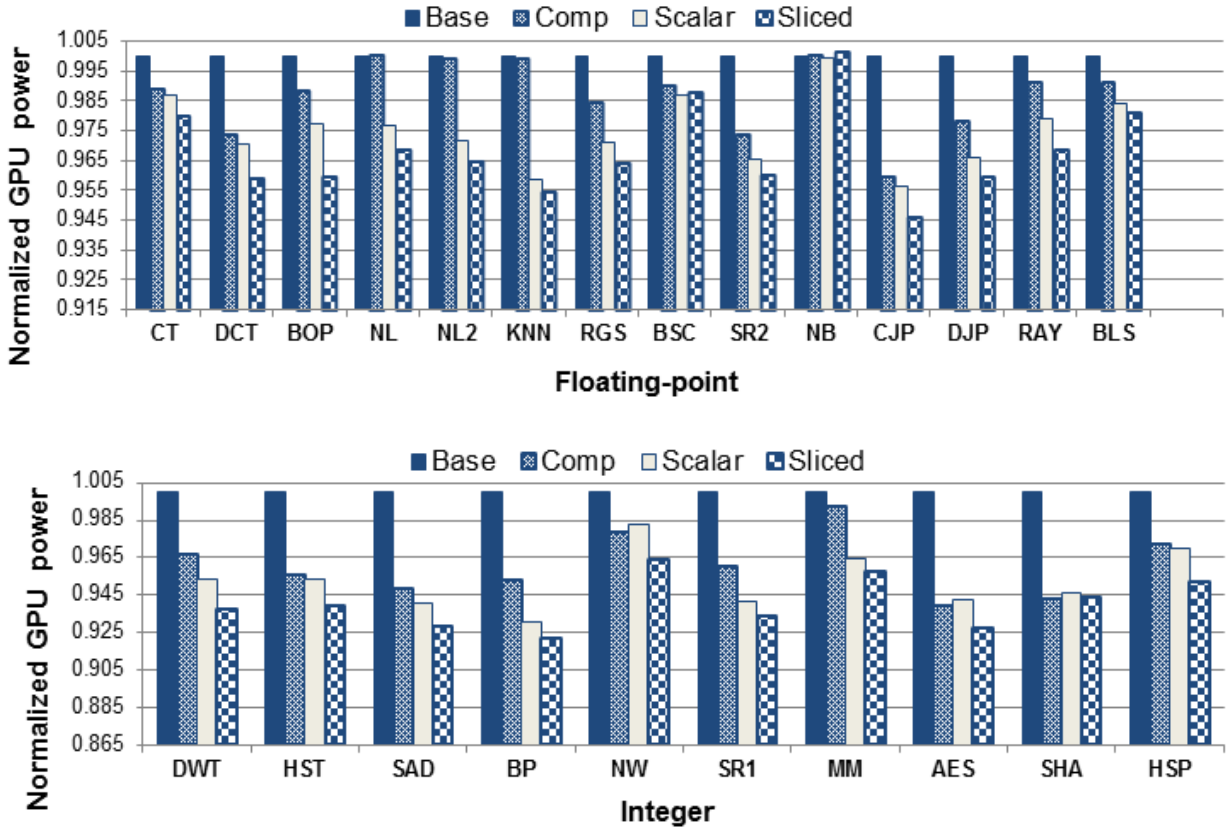


Figure 7.12: GPU power consumption

7.5.3 GPU POWER

Figure 7.12 shows the impact of my techniques on the total GPU power. The GPU power includes dynamic power consumption of SMs, on-chip memories, the interconnection network, global memory power, and the GPU leakage power. The leakage overhead of additional RFs and logic is also included in GPU power. At the GPU level, my techniques reduce the power consumption by nearly 5% (geometric mean). Due to the greater optimization opportunities in integer benchmarks, they exhibit a greater power reduction (almost 6%) than FP benchmarks (more than 3%). Although a 5% reduction in power consumption may seem insignificant, for power constrained GPUs, reductions in power consumptions can directly be translated into performance

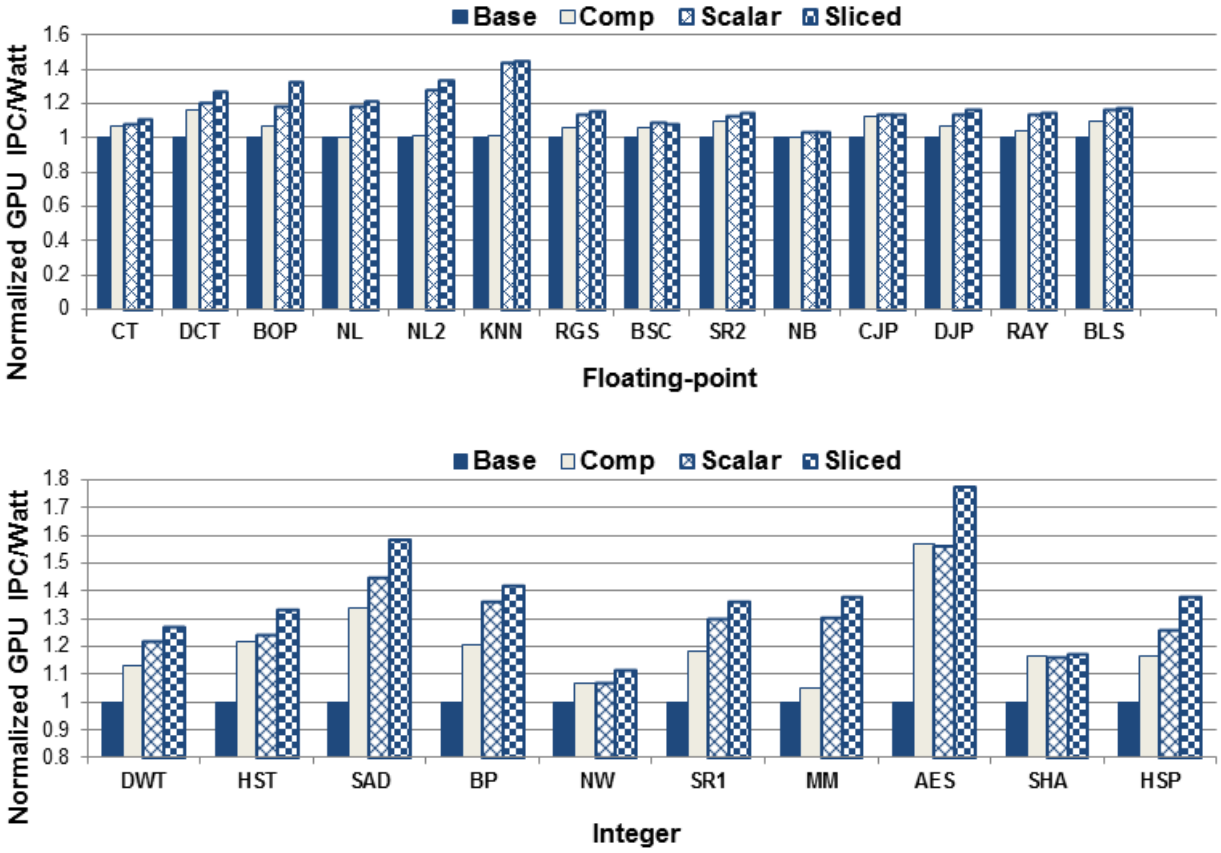


Figure 7.13: GPU IPC per Watt

improvements. Moreover, this power reduction comes with a significant performance improvement, resulting in much higher power efficiency than the baseline GPU. The next section describes the impact of my approaches on power efficiency.

7.5.4 GPU POWER EFFICIENCY

The primary goal of this portion of my dissertation was to improve the power efficiency of GPU architectures for GPGPU applications. The impact of my proposed approaches on power efficiency is summarized in Figure 7.13. Power efficiency is represented by the instructions per cycle (IPC) per Watt for the entire GPU. Due to the limited power overhead of my approaches, the achieved performance improvements can be translated into significant improvement in power

efficiency of the GPU. For integer and FP benchmarks, my approaches can provide more than 35% and 19% (geometric mean) improvement in power efficiency, respectively. Across both integer and FP benchmarks, my approaches achieve more than 25% (geometric mean) power efficiency improvement.

7.6 DISCUSSION

Although I evaluated all of the proposed approaches with a GPU configuration similar to the NVIDIA FX5800 (GT200 architecture), my proposed techniques can also be employed with more recent GPU architectures (e.g., Fermi and Kepler). Due to a lack of publically-available validated performance simulators and power models for these newer architectures, I limited my evaluation to an architecture similar to the GT200.

All of my techniques utilize the dual-issue capability of the baseline GPU [70] to issue up to two instructions per cycle. Moreover, all of my proposed approaches exploit common instruction and/or data patterns. These patterns can be exploited in any GPU architecture with some implementation modifications. In this section, I describe the primary benefits of each my proposed approaches and show how these benefits can still be achieved in more recent GPU architectures (e.g., Fermi and Kepler [84, 85]).

7.6.1 COMPOSITE INSTRUCTIONS

The primary motivation for utilizing composite instructions is to exploit common instruction patterns to reduce the number of instructions that need to be executed for the same amount of work. I re-use the FMA pipeline to combine dependent instructions that can easily be mapped to the existing FMA resources. This allows the GPU and its compiler to greatly reduce the total

number of instructions in a kernel at a negligible power overhead. It is important to note that I do not implement a new hardware datapath that is suitable for only a few kernels. Using the FMA datapath, my approach is able to able to cover a large percentage of general instruction patterns that exist in most kernels.

The main benefit of composite instructions comes from the fact that every time a composite instruction is issued, the GPU effectively executes two instructions. While in the baseline GPU two issue cycles would be needed to do the same work, a composite instruction only takes one issue cycle. This allows a different instruction (which can be another composite instruction) to be executed in the next issue cycle. In other words, a composite instruction increases the *effective* IPC of the kernel by doing the same work in fewer cycles.

Since FMA units are present in all GPUs, composite instructions can be easily incorporated in any GPU architecture. The inclusion of composite instructions in the GPUarchitecture requires a simple compiler pass that identifies composable instructions and replaces them with composite instructions. The replacement can either be done in the compiled code or by the GPU driver that converts the immediate representation in the compiled code to the GPU's architecture-specific ISA.

Both Fermi and Kepler include separate integer and FP execution pipelines. For both of these architectures, the performance improvement with composite instructions may be even higher than for the GT200 architecture, because normal integer instructions and composite instructions can be issued to the integer and FP pipelines in parallel.

7.6.2 EXPLOITING COMPUTATIONAL REDUNDANCY

I employed a scalar pipeline to exploit the computational redundancy in kernels to improve

their performance. Computational redundancy is an architecture-independent characteristic and will exist in any GPU architecture. The extent of computational redundancy depends upon the application domain, the particular kernel, and the particular input data set. In order to take advantage of computational redundancy, I maintained a scalar bit within each SIMT register. If the scalar bits of all the operands of an instruction are set, the instruction is executed in the scalar pipeline; otherwise it executes in the SIMT pipeline.

The performance improvement with the scalar pipeline is achieved by utilizing the dual-issue capability of the GPU to issue instructions to the SIMT and scalar pipelines simultaneously. The baseline GPU architecture schedules a warp over four cycles in an 8-wide SIMT. Although a computationally redundant instruction that executes in the scalar pipeline only requires a single-cycle (instead of four), I do not exploit this to issue more scalar instructions in the remaining three cycles. This is done for two reasons. First, for the GT200, this would increase the complexity of FDS logic, since instead of scheduling warps every four cycles it should also be able to schedule new warps every cycle (if they execute in the scalar pipeline). Second, I employ a single write-port SRF to store the results of computationally redundant instructions. For cases where both the SIMT and scalar pipelines need to write the result to the SRF, a single write port can be shared such that the scalar pipeline can use it in the first cycle and the SIMT pipeline can use it in the fourth cycle (after the comparison stage has detected the SIMT execution result to be the same across all SIMT lanes).

The Kepler architecture schedules a complete warp (32 threads) in a single cycle. The comparison stage (shown in Figure 7.6) can easily be extended to compare results across all 32 SIMT lanes. Since the SIMT unit only performs comparison between adjacent lanes (using XOR gates),

an increase in the SIMT width does not increase any critical paths for that stage (the all zero detection logic is made using dynamic gates and its delay does not increase with the increased SIMT width). However, since the Kepler architecture issues the complete warp in a single cycle, for cases in which both the SIMT and scalar pipelines need to write the result to the SRF, an additional write port will be required in the SRF. However, the SRF has a very lower power impact and will not drastically change the power efficiency of the approach in Kepler.

Lastly, the performance advantage with the scalar pipeline is likely to be higher with Kepler. This is because the Kepler scheduler can schedule warps every cycle. This will allow scalar pipeline to be utilized more efficiently, as new scalar instructions can be issued every cycle (compared to the baseline GPU, which can only schedule scalar instructions every four cycles).

7.6.3 BIT-WIDTH SLICING

Bit-width slicing was motivated by the fact that many operands have sign-extended bits that waste the register file bandwidth and execution resources of the GPU. I partitioned the register file banks and the execution datapaths into two paths. The dual-issue capability of the GPU is exploited to issue up to two instructions to the sliced data paths. This approach requires the SIMT unit to maintain to a scalar bit for the 16 MSBs per SIMT register. This *scalar bit MSBs* is determined similar to the scalar bit used for exploiting computational redundancy. The increased SIMT width of Kepler thus does not impact the implementation of this approach.

7.7 CONCLUSION

GPUs have traditionally improved their peak compute performance by increasing the number of compute resources and/or their frequency. However, modern high performance GPUs are

power-constrained and it is becoming more difficult to increase their compute resources or frequencies within their power budgets. Consequently, power efficient approaches need to be developed to improve performance in future processors. In this study, I proposed three techniques that improved both performance and power efficiency of GPU architectures: (i) Using FP FMA units to accelerate integer instructions; (ii) Employing a scalar pipeline to execute computationally redundant SIMT instructions; and (iii) Employing a bit-sliced GPU architecture to efficiently utilize RF and execution bandwidth.

First, I exploited the resources of the execution pipeline to form composite instructions that reduced the power consumption of critical resources (FDS, MRF, EU) and also improved performance significantly. Second, I exploit computational redundancy with a scalar unit to provide additional performance and power improvements. Finally, I take advantage of short bit-width operands, which commonly occur in GPU applications, using a sliced GPU architecture. Individually, my proposed techniques provide geometric mean performance improvements of 9% (15% for integer and 5% for FP), 7% (7% for integer and 9% for FP) and 4% (7% for integer and 2% for FP) for composite, scalar and sliced techniques, respectively. Together, my techniques improved performance by 15% (geometric mean) and power efficiency by 25%.

8. CONCLUSIONS

This dissertation proposed and evaluated several techniques to improve the compute efficiency of embedded processors. Modern embedded architectures employ a SoC design approach that integrates different processors (e.g. CPUs, DSPs and GPUs) on the same chip. These heterogeneous architectures can provide significant performance improvements by allowing software developers to utilize the processor most suited to a given task. However, the limited power budgets of embedded architectures and increasing compute requirements of modern applications requires future SoC architectures to significantly improve their compute efficiency. The research proposed in this dissertation improves performance and energy efficiency of embedded processors by employing cross-layer optimizations. These optimizations take advantage of application and data characteristics, a closer interaction between the compiler and the micro-architecture, micro-architectural modifications and circuit-level approaches to estimate and reduce processor power consumption.

The key contributions of this dissertation are enumerated below:

1. **An analysis of performance, power and energy trade-offs of FP, FxP and software emulated FP codes**: FP arithmetic is typically avoided in embedded applications due to its high performance and power overhead. However, this dissertation quantitatively shows that for many applications FP codes are more energy efficient and have better performance than equivalent FxP and software emulated FP codes.
2. **A novel high-throughput FMA unit for low-power embedded processors that reduces power and energy consumption while improving the performance of typical FP applications**: The proposed design utilizes the insight that typical FP applications

involve long dependence chains of FMA/ADD operations. The longer latency of FP FMA/ADD operations than integer operations results in significant performance degradation for FP applications. The proposed design ameliorates the performance loss due to these long dependence chains by avoiding FP rounding and normalization for a large percentage of dependent FP FMA/ADD instructions.

3. **Employing Virtual-FPUs to reduce the area and power overhead of FPUs**: Virtual-FPUs utilize the FxP execution resources of the processor to mimic an FP pipeline. The proposed approach improves performance and energy efficiency of many FP kernels compared to dedicated FPUs while incurring an extremely low area and power overhead.
4. **Performance-power improvements for power-constrained GPUs through reduced effective pipeline latencies**: For power constrained GPUs, this dissertation proposes compiler-directed data-forwarding that allows the execution pipelines to be operated at a lower voltage without increasing the RAW latencies of a large percentage of instructions. The overhead of supporting DP FP operations in GPUs is also reduced by re-using SP FMA units. The combination of voltage scaling and re-using SP FMA units allows GPUs to increase the number of SMs (and hence performance) without violating the power constraint of the GPU.
5. **Accelerating integer instruction execution using the FP FMA units in GPUs**: GPUs have a large number of FP FMA units that can be used improve performance and energy consumption of many GPGPU applications. This dissertation proposes to map dependent integer instructions to the arithmetic blocks within an FMA unit to form a sin-

gle composite instruction. Composite instructions reduce the total number of instructions in the kernel and consequently improve kernel performance and energy consumption. The key insight in this approach is that the energy consumed per instruction in a GPU is dominated by several non-execution stages of the pipeline (e.g. instruction cache access, fetch and decode logic, fetch buffer, instruction and warp schedulers, register file accesses). A composite instruction increases the execution energy consumption (since integer instructions are executed in the FP pipeline) but reduces the energy consumption of all the other micro-architectural components of the GPU.

6. **Scalar execution of computationally redundant SIMT instructions**: This dissertation shows that many GPPGU applications have a large percentage of computationally redundant instructions. These computationally redundant instructions have similar operand values for all SIMT threads. Executing computationally redundant instructions results in unnecessary register file and execution energy consumption. This dissertation proposes to dynamically detect computationally redundant instructions and issue them to a scalar pipeline to improve both the instruction throughput and power efficiency of the GPU.
7. **Employing a bit-sliced GPU architecture to exploit low bit-width operands**: This dissertation shows that a large percentage of instruction operands in GPPGU applications have 16 or more sign-extended MSBs. Reading such operands from the register file and executing them in 32-bit execution units wastes RF and execution bandwidth. This dissertation proposes to efficiently exploit low bit-width operands in GPGPU applications by employing a bit-sliced GPU architecture. The bit-sliced GPU architecture

divides the warp scheduler, the register file and the execution units into two slices. Instructions whose operands only require 16 or fewer bits for accurate representation can be co-issued to the execution slices allowing a significant improvement in performance and power consumption of the GPU.

9. FUTURE RESEARCH

This dissertation proposes several techniques to improve the compute efficiency of embedded processors. However, there are still many avenues for future research that can be explored further. One potential area for future research is to enhance the proposed FMA unit to make it IEEE-754 compliant. This can be achieved by a employing a rounding prediction scheme that predicts whether or not rounding is required in the FP result. If rounding is predicted to be required, a rounding one can be added to the dependent FP operation along with the intermediate result forwarded from the parent operation.

Many embedded processors provide hardware support for FP arithmetic in the form of co-processors (e.g. the NEON SIMD units in ARM processors). Virtual-FPUs in such co-processors can lead to different energy and power trade-offs that were not studied in this dissertation. The SIMD FPUs have a significant leakage and dynamic power overhead. However, for applications that do not have many dependent FMA/ADD operations (e.g. FFT algorithms), they can provide a higher performance than the Virtual-FPUs. These tradeoffs can be investigated as a potential future research direction.

During my dissertation research I also contributed to the development of a configurable and validated power model for GPUs. The power model integrates McPAT and GPGPU-Sim to provide estimates of temporal and spatial power variations in the GPU. The power model will allow researchers to analyze GPU power consumption in detail along with the impact of architectural and micro-architectural changes on power and energy consumption of the GPU.

While this dissertation focussed on improving power efficiency of GPGPU applications, fur-

ther research into the characteristics of graphics workloads can be done to evaluate the potential of the proposed approaches for graphics workloads. However, I was unable to do this evaluation due to the lack of publically available simulation infrastructure. Investigating power efficiency techniques for graphics also has significant potential for future research.

BIBLIOGRAPHY

[1] NVIDIA, "NVIDIA® Tegra multiprocessor architecture," 2010.

[2] Texas Instruments, "System-level software performance: How to get the most performance out of the OMAP4 platform," 2009.

[3] Oracle®, [Online]. Available: http://openjdk.java.net/projects/sumatra/.

[4] HSA foundation, [Online]. Available: http://hsafoundation.com/.

[5] W. Wolf, High-Performance Embedded Computing: Architectures, Applications and Methodologies, Morgan Kaufmann, 2006.

[6] Texas Instruments, "OMAP™ 5 mobile applications platform," 2011.

[7] Texas Instruments, *TMS320C6454 Fixed-point digital signal processor,* 2006.

[8] M. Schulte, J. Glossner, S. Jinturkar, M. Moudgill, S. Mamidi and S. Vassiliadis, "A low-power multithreaded processor for software defined radio," *The Journal of VLSI Signal Processing,* vol. 43, pp. 143-159, 2006.

[9] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti and K. Flautner, "SODA: A high-performance DSP architecture for software-defined radio," vol. 27, pp. 114-123, 2007.

[10] K. Castille, *TMS320C62x/C67x power consumption summary,* Texas Instruments.

[11] D. Menard, D. Chillet, F. Charot and O. Sentieys, "Automatic floating-point to fixed-point conversion for DSP code generation," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2002.

[12] S. Z. Gilani, N. S. Kim and M. Schulte, "Energy-efficient floating-point arithmetic for software-defined radio architectures," in *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 2011.

[13] S. Gilani, N. S. Kim and M. Schulte, "Energy-efficient floating-point arithmetic for digital signal processors," in *Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, 2011.

[14] V. Reddi, S. Gilani, E. Gunadi, N. S. Kim, M. Schulte and M. Lipasti, *CoDeC: Compressed dependence chains for resource constrained processors,* 2013.

[15] A. Pedram, S. Gilani, N. S. Kim, R. van de Geijn, M. Schulte and A. Gerstlauer, "A linear algebra core design for efficient level-3 BLAS," in *IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2012.

[16] S. Gilani, N. S. Kim and M. Schulte, "Virtual floating-point units for low-power embedded processors," in *IEEE International Conference on Application-specific Systems, Architectures and Processors*, Delft, 2012.

[17] S. Gilani, N. S. Kim and M. Schulte, *Performance-power improvements for power-constrained GPUs through reduced effective pipeline latencies,* 2013.

[18] S. Gilani, N. S. Kim and M. Schulte, *Power-efficient computing for compute-intensive GPGPU applications,* 2013.

[19] "GeForce® 8800 & NVIDIA® CUDA™: A new architecture for computing on the GPU," [Online].

Available: www.gpgpu.org.

[20] J. Lee, V. Sathisha, M. Schulte, K. Compton and N. S. Kim, "Improving throughput of power-constrained GPUs using dynamic voltage/frequency and core scaling," in *International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[21] ITRS, 2011. [Online]. Available: http://www.itrs.net/Links/2011ITRS/2011Chapters/2011PIDS.pdf.

[22] S. Hong and H. Kim, "An integrated GPU power and performance model," in *International Symposium on Computer Architecture*, 2010.

[23] E. M. Schwarz, M. Schmookler and S. Trong, "Hardware implementations of denormalized numbers," in *IEEE Symposium on Computer Arithmetic*, Spain, 2003.

[24] A. Mitra, M. Chakraborty and H. Sakai, "A block floating-point treatment to the LMS algorithm: efficient realization and a roundoff error analysis," *IEEE Transactions on Signal Processing,* vol. 53, pp. 4536- 4544, 2005.

[25] A. Oppenheim, "Realization of digital filters using block-floating-point arithmetic," *IEEE Transactions on Audio and Electroacoustics,* vol. 18, no. 2, pp. 130-136, 1970.

[26] "A 2048 complex point FFT processor using a novel data scaling approach," in *International Symposium on Circuits and Systems*, 2003.

[27] K. Kalliojarvi and J. Astola, "Roundoff errors in block-floating-point systems," vol. 44, pp. 783-790, 1996.

[28] IEEE, "IEEE standard for floating-point arithmetic," 2008.

[29] A. Gaffar, O. Mencer, W. Luk and P. Cheung, "Unifying bit-width optimisation for fixed-point and floating-point designs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 79-88.

[30] F. Fang, T. Chen and R. Rutenbar, "Lightweight floating-point arithmetic: Case study of inverse discrete cosine transform," in *EURASIP Journal on Signal Processing, Special Issue on Applied Implementation of DSP and Communication Systems*, 2002.

[31] N. Hockert and K. Compton, "FFPU: Fractured floating point unit for FPGA soft processors," in *International Conference on Field-Programmable Technology*, 2009.

[32] H. Keding, M. Willems, M. Coors and H. Meyr, "FRIDGE: A fixed-point design and simulation environment," in *Design, Automation and Test in Europe*, 1998.

[33] A. Sergyienko and O. Maslennikov, "Implementation of givens QR-decomposition in FPGA," in *Parallel Processing and Applied Mathematics*, 2006.

[34] J. Eilert, D. Wu and D. Liu, "Efficient complex matrix inversion for MIMO software defined radio," in *IEEE International Symposium on Circuits and Systems*, 2007.

[35] P. Luethi, C. Studer, S. Duetsch, E. Zgraggen, H. Kaeslin, N. Felber and W. Fichtner, "Gram-schmidt-based QR decomposition for MIMO detection: VLSI implementation and comparison," in *IEEE Asia Pacific Conference on Circuits and Systems*, 2008.

[36] A. Ahmedsaid, A. Amira and A. Bouridane, "Improved SVD systolic array and implementation on FPGA," in *IEEE International Conference on Field-Programmable Technology*, 2003.

[37] M. Gebhart, D. Johnson, D. Tarjan, S. Keckler, W. Dally, E. Lindholm and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *International symposium on Computer architecture*, 2011.

[38] M. Gebhart, S. Keckler and W. Dally, "A compile-time managed multi-level register file hierarchy," in *IEEE/ACM International Symposium on Microarchitecture*, 2011.

[39] I. Kim and M. Lipasti, "Macro-op scheduling: relaxing scheduling loop constraints," in *International Symposium on Microarchitecture*, 2003.

[40] D. Brooks and M. Martonosi, "Value-based clock gating and operation packing: dynamic strategies for improving processor power and performance," *ACM Transactions on Computer Systems,* vol. 18, no. 2, pp. 89-126, 2000.

[41] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. Lee, S. Richardson, C. Kozyrakis and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *annual international symposium on Computer architecture*, 2010.

[42] O. Ergin, D. Balkan, K. Ghose and D. Ponomarev, "Register packing: Exploiting narrow-width operands for reducing register file pressure," in *International Symposium on Microarchitecture*, 2004.

[43] G. Andrea and H. Corporaal, "Floating-point to fixed-point conversion of C code," in *International Conference on Compiler Construction*, 1999.

[44] CodeMaestro, [Online]. Available: http://www.codemaestro.com/reviews/9.

[45] J. Hauser. [Online]. Available: http://www.jhauser.us/arithmetic/SoftFloat.html.

[46] ARM®, "ARM® compiler toolchain," 2011.

[47] M. Pharr and R. Fernando, GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (GPU Gems), Addison-Wesley Professional, 2005.

[48] J. Preiss, M. Boersma and S. Mueller, "Advanced clockgating schemes for fused-multiply-add-type floating-point units," in *IEEE Symposium on Computer Arithmetic*, 2009.

[49] N. Banerjee, A. Raychowdhury, K. Roy, S. Bhunia and H. Mahmoodi, "Novel low-overhead operand isolation techniques for low-power datapath synthesis," *IEEE Transactions on Very Large Scale Integration Systems,* vol. 14, pp. 1034-1039, 2006.

[50] Texas Instruments, "TMS320C6745,TMS320C6747 Fixed/floating-point digital signal processor," 2010.

[51] S. Liu, E. Lindholm, M. Y. Siu, B. W. Coon and S. F. Oberman, "Operand collector architecture". US Patent 7,834,881 B2, 16 November 2010.

[52] NVIDIA®, "NVIDIA® CUDA™ C programming guide v4.0," 2011.

[53] B. S. Nordquist and S. D. Lew, "Apparatus, system, and method for coalescing parallel memory requests". US Patent 7492368, 17 February 2009.

[54] NVIDIA®, "CUDA C best practices guide," 2011.

[55] NVIDIA®™, "NVIDIA's® Next generation CUDA™ compute architecture: Fermi™," 2009.

[56] NVIDIA®, "GeForce 8800 & NVIDIA CUDA: A new architecture".

[57] J. Hwa, S. Mueller, C. Jacobi, K. Tran, S. Cottier, B. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida and S. Dhong, "A fully-pipelined single-precision floating point unit in the synergistic processor element of a CELL processor," in *Symposium on VLSI circuits*, 2005.

[58] J. Leng, S. Gilani, T. Heatherington, A. ElShafiey, V. Reddi, N. S. Kim and T. Aamodt, *A flexible and adaptive framework for exploring energy-efficiency in GPUs,* 2013.

[59] Microway®, "GPGPU architecture comparison of NVIDIA® and ATI™ GPUs," June 2010. [Online]. Available: http://www.microway.com/pdfs/GPGPU_Architecture_and_Performance_Comparison.pdf.

[60] NVIDIA®, "NVIDIA® GeForce™ GTX 680 whitepaper".

[61] M. Ercegovac and T. Lang, Digital arithmetic, Morgan Kaufmann, 2004.

[62] R. Puri, L. Stok, J. Cohn, D. Kung, D. Pan, D. Sylvester, A. Srivastava and S. Kulkarni, "Pushing ASIC performance in a power envelope," in *Proceedings of the 40th annual Design Automation Conference*, 2003.

[63] D. Lackey, P. Zuchowski, T. Bednar, D. Stout, S. Gould and J. Cohn, "Managing power and performance for System-on-Chip designs using Voltage Islands," in *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, 2002.

[64] L. Chang, R. Montoye, B. Ji, A. Weger, K. Stawiasz and R. Dennard, "A fully-integrated switched-capacitor 2:1 voltage converter with regulation capability and 90% efficiency at 2.3A/mm2," in *IEEE Symposium on VLSI Circuits (VLSIC)*, 2010.

[65] M. Khellah and M. Elmasry, "Power minimization of high-performance submicron CMOS circuits using a dual-V/sub dd/ dual-V/sub th/ (DVDV) approach," in *International Symposium on Low Power Electronics and Design*, 1999.

[66] A. Bakhoda, G. Yuan, W. Fung, H. Wong and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[67] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *IEEE/ACM International Symposium on Microarchitecture*, 2009.

[68] HP, [Online]. Available: http://quid.hpl.hp.com:9081/cacti.

[69] S. Galal and M. Horowitz, "Energy-Efficient Floating-Point Unit Design," *IEEE Transactions on Computers,* vol. 60, no. 7, pp. 913-922, 2011.

[70] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *IEEE International Symposium on Performance Analysis of Systems & Software*, 2010.

[71] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands and K. Yelick, "The potential of the cell processor for scientific computing," in *Proceedings of the Conference on Computing frontiers*, 2006.

[72] E. Lindholm, J. Nickolls, S. Oberman and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro,* vol. 28, no. 2, pp. 39-55, 2008.

[73] Advanced Micro Devices, [Online]. Available: http://developer.amd.com/gpu_assets/Heterogeneous_Computing_OpenCL_and_the_ATI_Radeon_HD_5870_Architecture_201003.pdf.

[74] Advanced Micro Devices, 2008. [Online]. Available: http://developer.amd.com/gpu_assets/r600isa.pdf.

[75] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Transactions on Graphics,* vol. 27, no. 3, pp. 18:1--18:15, 2008.

[76] A. Ozsoy and M. Swany, "CULZSS: LZSS lossless data compression on CUDA," in *IEEE International Conference on Cluster Computing*, 2011.

[77] H. Nguyen, GPU Gems 3, Addison-Wesley Professional, 2007.

[78] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee and K. Skadron, " Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization*, 2009.

[79] NVIDIA®, "NVIDIA® CUDA™ SDK v2.3," [Online]. Available: http://developer.nvidia.com/cuda-toolkit-23-downloads.

[80] Advanced Micro Devices, [Online]. Available: http://developer.amd.com/afds/assets/presentations/2620_final.pdf.

[81] N. Weste and D. Harris, CMOS VLSI Design: A circuits and systems perspective, Addison Wesley, 2010.

[82] A. Akkaş and M. Schulte, "Dual-mode floating-point multiplier architectures with parallel operations," *Journal of Systems Architecture,* vol. 52, no. 10, pp. 549-562, 2006.

[83] L. Huang, L. Shen, K. Dai and Z. Wang, "A new architecture for multiple-precision floating-point multiply-add fused unit design," in *IEEE Symposium on Computer Arithmetic*, 2007.

[84] NVIDIA®, *NVIDIA's® Next Generation CUDA™ compute architecture: Fermi™,* 2009.

[85] NVIDIA®, *Whitepaper: NVIDIA® GeForce™ GTX 680,* 2012.

[86] D. Chang, C. Jenkins, P. Garcia, S. Gilani, P. Aguilera, A. Nagarajan, M. Anderson, M. Kenny, S. Bauer, M. Schulte and K. Compton, "ERCBench: An open-source benchmark suite for embedded and reconfigurable computing," in *International Conference on Field Programmable Logic and Applications*, 2010.

[87] J. Preiss, M. Boersma and S. Mueller, "Advanced Clockgating Schemes for Fused-Multiply-Add-Type Floating-Point Units," in *IEEE Symposium on Computer Arithmetic*, 2009.

[88] J. Hauser. [Online]. Available: http://www.jhauser.us/arithmetic/SoftFloat.html.

[89] M. Woh, Y. Lin, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, R. Bruce, D. Kershaw, A. Reid, M. Wilder and K. Flautner, "From SODA to scotch: The evolution of a wireless baseband processor," in *IEEE/ACM International Symposium on Microarchitecture*, 2008.

[90] ARM®, [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a9.php.

[91] NVIDIA®, "NVIDIA® CUDA™ SDK v4.0".

[92] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *IEEE Transactions on Computers,* vol. 60, no. 7, pp. 913-922, 2011.

[93] K. Turkowski, "Fixed point square root," 1994.

[94] S. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie and M. Taylor, "SD-VBS: The San Diego vision benchmark suite," in *IEEE International Symposium on Workload Characterization*, 2009.

[95] D. Young, Iterative solution of large linear systems, Courier Dover Publications, 2003.

LIST OF ABBREVIATIONS

Abbreviations	Description
ARF	Accumulator register file
ASFT	Alignment shift in FMA operations
BFP	Block floating-point
BFP-FP	Hybrid block-floating-point-floating-point approach
CDF	Compiler-directed forwarding
Comp	Composite instructions
CRF	Composite register file
DFN	Data-forwarding network
DP	Double precision floating-point
DSP	Digital signal processor
DVFS	Dynamic voltage and frequency scaling
DVS	Dynamic voltage scaling
EC	Exponent correction
EES	Exponent, exception and sign handling
EU	Exponent update
EX	Execution
FDS	Fetch-decode-schedule
FMA	Floating-point fused multiply add instruction
FP	Floating-point
FPU	Floating-point unit
FxP	Fixed-point
GPGPU	General purpose GPU applications
GPU	Graphics processing units
HFMA	High throughput fused multiply add unit
IWL	Integer word length
LSB	Least significant bit
LZA	Leading zero anticipator
LZC	Leading zero counter
MAC	Integer multiply-accumulate
MRF	Main register file
MSB	Most significant bit
NSFT	Normalization shift in FMA operations
OOO	Out-of-order
RAW	Read-after-write
RND	Rounder
SE	Software emulated FP
SFU	Special functional unit
SIMD	Single instruction multiple data
SIMT	Single instruction multiple threads
SNR	Signal-to-noise ratio
SoC	System-on-chip
SP	Single precision floating-point
SRF	Scalar register file
Virtual BFP-FP	Virtual FPUs exploiting the BFP-FP approach
Virtual-FPU	Virtual floating-point units
VLIW	Very long instruction word
WL	Word length