ENERGY EFFICIENT COMPUTING EXPLOITING DATA SIMILARITY AND COMPUTATION REDUNDANCY

BY

ZHENHONG LIU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

ABSTRACT

Applications in various fields, such as machine learning, scientific computing and signal/image processing, need to deal with real-world input datasets. Such input datasets are usually discrete samples of slow-changing, continuous data of physical phenomena, like temperature maps and images. Due to the continuous nature of the physical phenomena, these datasets often contain data points with similar or even identical values. Eventually, it results in repeated operations performed on the same or similar data points, i.e. redundant computation. Redundant computation can be exploited to improve the energy/power efficiency and performance of processors, especially when the benefits from process technology scaling and power scaling keep diminishing.

This dissertation first proposes a cost-effective generalized scalar execution architecture for GPUs, called G-Scalar. It exploits the redundant computation performed on identical data. G-Scalar offers two key advantages over prior architectures supporting scalar execution for only non-divergent arithmetic/logic instructions. First, G-Scalar is more power efficient as it can also support scalar execution of divergent and special-function instructions, the fraction of which in contemporary GPU applications has notably increased. Second, G-Scalar is less expensive as it can share most of its hardware resources with register value compression, of which adoption has been strongly promoted to reduce high power consumption of accessing the large register file. Compared with the baseline and previous scalar architectures, G-Scalar improves power efficiency by 24% and 15%, respectively, at a negligible cost.

Lock and Load (LnL) architecture is then proposed to extend the coverage to redundant computation on similar data, by enabling approximate computing. In the LnL architecture, approximate computing is triggered by similarity of values returned by load instructions, and then approximation is applied to the annotated code region following the load instructions. Such a design reduces the overhead of checking eligibility of approximation for every instruction and allows us to deploy more sophisticated techniques for checking the eligibility of approximation and approximating the output values for all the skipped threads at the end. LnL is further enhanced to fuse the same approximated instructions from multiple warps into a single instruction, exploiting the fact that only a subset of threads in approximated warps are executed and many execution lanes are left unused by the skipped threads. This not only improves the performance but also reduces energy consumption, as it reduces the number of fetched, decoded, scheduled and executed instructions. Our experiment shows that LnL can improve energy efficiency and performance by 62% and 23%, respectively.

Finally, we propose AxMemo to exploit the computation redundancy on CPUs. Inspired by LnL, AxMemo focuses on memoizing relatively large blocks of code with a variable number of inputs. In contrast, existing memoization techniques mostly replace costly floating-point operations that have a limited number of inputs with memory lookup. Since AxMemo aims to replace long sequences of instructions with a few lookup operations, it alleviates the von Neumann and execution overheads of passing instructions through the processor pipeline altogether. To address the challenge of handling various numbers of inputs, we develop a novel use of Cyclic Redundancy Checking (CRC) to hash the inputs and use the hash as lookup tags. It enables AxMemo to efficiently memoize relatively large code regions with variable input sizes and types using the same underlying hardware. Our experiment shows that AxMemo offers 2.82× speedup and 63% energy reduction, with mere 0.2% of quality loss averaged across ten benchmarks. These benefits come with an area overhead of just 2.08%.

To my parents, for their patience and unwavering support.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Nam Sung Kim. This dissertation is only made possible with his continued support and guidance. The inspiring discussions with him and his constructive feedback guided me through all the difficulties in my research. I would also like to thank Professor Wen-Mei Hwu, Professor Naresh R. Shanbhag and Professor Josep Torrellas for serving as my committee members, as well as for their valuable comments and suggestions.

I am grateful to all my collaborators: Dr. Syed Gilani, Professor Murali Annavaram, Professor Daniel Wong, Dr. Amir Yazdanbakhsh, Professor Hadi Esmaeilzadeh and Dong Kai Wang. It is their hard work and contributions that made the projects comprising this dissertation achieve their current state. My special thanks go to Dr. Hao Wang, who greatly helped me with my transition to a Ph. D. student.

My sincere appreciation goes to all my colleagues, for all of the free IT service they offered, the insightful peer review they provided and the delightful conversations we shared. I am also heavily indebted to all my friends. They not only made the life as a graduate student much more enjoyable, but also gave me all the rides that made it possible for me to survive in the Midwest without having a car.

Finally, I am extremely thankful to my family for being incredibly supportive and patient. Without their endless support and encouragement, I would never be able to make it to the end of this long journey.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The improvement of the computation capability of processors has been driven by the combination of process technology scaling and architecture innovations. Semiconductor process technology scaling allows us to integrate more hardware resources into the integrated circuit (IC) chips and make the new architectural features possible. From another aspect, it is the architecture innovations that allow the extra hardware resources to be fully utilized. However, the effectiveness of such an approach is getting more and more limited, because semiconductor technology scaling is approaching its fundamental physics limit [1]. Though the transistor feature size is still shrinking, the failing power scaling limits further integration of hardware resources. Power and thermal constraints have become a major bottleneck, causing problems such as dark silicon [2].

As it is becoming increasingly more difficult to further improve processor performance, we propose an alternative approach in this dissertation. Instead of trying to complete computation faster by adding more and more hardware resources, we try to reduce the amount of needed computation. To achieve this goal, we exploit the data similarity and computation redundancy that exist in many applications.

Literally, data similarity simply means that a set of data has similar values (according to certain application-specific metric, such as the absolute difference of the values compared to a threshold). Many applications have input datasets that show data similarity, such as input images for machine learning applications. Data similarity can be exploited to improve energy efficiency. However, not all such cases can be exploited to improve energy efficiency. For instance, consecutive memory addresses appear to be similar, but the data returned by memory accesses to those memory locations can be drastically different, and none of the operations can be eliminated without introducing significant error. Therefore, we only consider an input dataset having data similarity if the input data points have similar value and the corresponding outputs are also similar. In such cases, some of the computation can be eliminated and the outputs can be estimated from other data points.

A major source of data similarity is that applications need to process real-world data. These data are sampled from certain physical phenomena, such as heat maps and images. Due to the nature of the phenomena, the sampled data are often continuous and slow-changing. These data usually further show “spatial correlation”, where a data point is similar compared with its “neighbors”. A simple example of such a case is a heat map of a surface with small temperature variation. In the heat map example, the “neighbors” are naturally the neighboring data points on the heat map grid. The correlation can provide even more opportunity for optimization.

When multiple instances of the exact same sequence of instructions take identical inputs and generate identical outputs, we consider that the computation performed is redundant. Eliminating computation redundancy is an effective way to improve the energy efficiency of processors. There are many different ways to exploit computation redundancy at different granularities. However, the core idea is the same, which is reusing the computation results produced earlier during the execution. A key challenge for this problem is how to detect when redundant computation happens, and how to “store” and “reuse” previous results efficiently. To address the problem, different processor architectures with different execution models require different schemes to exploit the computation redundancy. Also, we need to focus on different parts of the processor for different architectures. For example, execution units consume a major portion of the total dynamic energy for graphics processing units (GPUs) [3], while for CPUs, energy spent on execution units can be trivial compared to the front-end and scheduling logic [4]. Therefore, it is important to choose the right targets for the schemes to exploit computation redundancy. In this dissertation, we implement different techniques for GPU and CPU to exploit the computation redundancy, respectively.

1.1 Related Work

Prior work has proposed various techniques to improve power and energy efficiency by reducing computation redundancy. Scalar execution on GPU avoids the redundant computation in the single-instruction, multiple-data (SIMD) pipelines [5, 6, 7]. Scalar execution converts a SIMD instruction into a single-instruction, single-data, or"scalar", instruction when the operands have the same values across the SIMD lanes. Yilmazer et al. further proposed to group more than one scalar instructions together and execute them in an execution pipeline in a batch [8]. However, all these scalar execution architectures only exploit scalar opportunity among non-divergent instructions, i.e. SIMD instructions whose lanes are all active. In contrast, we extend the concept of scalar instructions to divergent instructions.

Lee et al. proposed to analyze instructions eligible for scalar executions at compile-time [9]. However, this approach is purely for code analysis and limited by compile-time information and cannot exploit value similarity originated from executing load instructions. Collange et al. proposed methods to dynamically detect uniform and affine vectors in GPUs [10], which is tag-based and limited to a subset of the registers. Kim et al. proposed affine execution unit to explore the affine instructions [11]. However, it is only applicable to limited types of instructions.

Many approximation techniques have also been proposed [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]. These proposals use a variety of techniques to apply approximation, including data similarity exploitation [12, 13], neural network [17, 18, 25], loop perforation [21], memoization [23], operation substitution [20, 24, 26], and various approximate circuit techniques [14, 15, 16, 19, 22]. Among all the previous work, SAGE [12] and Warp Approximation [13] are the most relevant ones. Compared with scalar execution, these approximation techniques relax the constraints on the operand values and further exploit the opportunity to reduce redundant computation when the applications are fault-tolerant. SAGE proposed to fuse multiple threads into one thread. However, it relies on compilers to generate multiple versions of approximated kernels and requires tuning at runtime to choose the proper kernel for the target output error. Warp Approximation also checks operand values at runtime. Because it checks the operands of every instruction, it has to use a simple comparison scheme and fails to exploit the cases when the values are arithmetically similar but architecturally different.

Memoization is another useful technique for reducing redundant computation. However, it is traditionally used in very limited scope, such as avoiding unnecessary expensive arithmetic operations [27]. More recently, there have been proposals using both software-based and hardware-based techniques for more general-purpose memoization. Razlighi et al. [28] proposed a memoization-based neural network that does not need multipliers. It implemented LUTs with content addressable memories (CAMs) to replace the multipliers in neural networks. However, this implementation is only for neural networks and is not suitable for other applications. Brumar et al. [29] proposed a software implementation for task-level memoization. This approach is limited to task-based programs and the pure software approach only benefits a few benchmarks due to the large performance overhead. Tuck et al. [30] proposed a memoization method using their hardware-based memory access disambiguation approach. However the memoization itself is still software-based and brings little speedup. Connors et al. [31] proposed a compiler-directed instruction-level computation reuse, and Tsumura et al. [32] proposed an auto-memoization processor. These two proposals need either significant modification to the processor pipeline or complex hardware to track the input trees, which do not justify the relatively small performance improvement. Imani et al. [33] proposed memoization scheme on GPU using resistive content addressable memory (CAM). The CAM-based design is expensive and requires significant modification to the baseline GPU architecture. Sinha and Zhang [34] proposed a memoization-based approximate computing design on FPGA. However, this proposal only works for reconfigurable logic, because it cannot handle different number of inputs without using reconfigurable logic. Zhang and Sanchez [35] proposed to leverage caches for memoization. This proposal uses the concatenated inputs as LUT tag and it can only support up to 128 bits of total input, and the authors mostly showed the effectiveness of the scheme with one-input or two-input functions.

1.2 Dissertation Contributions and Organization

In this dissertation, we make the following observations:

1. We show that a significant fraction of instructions can be divergent and eligible for scalar execution in many contemporary GPU applications.

Therefore, scalar execution of such divergent instructions is critical for pushing the power efficiency envelope.

2. We observe that many GPU applications show not only data similarity, but also spatial correlation in the input dataset. Combined with the fault-tolerant nature of the applications, we show that there are abundant opportunities for exploiting these application characteristics.

3. We demonstrate that CPU applications also show a significant level of computation redundancy, which can be exploited to improve energy-efficiency and performance. We also explain that in order to improve power efficiency on CPU, it is crucial to reduce the front-end overhead, instead of focusing on the execution units only like previous work on GPUs.

Based on these observation, we propose G-Scalar architecture [36] and Lock and Load architecture [37] to exploit computation redundancy on GPUs, and AxMemo architecture to exploit computation redundancy on CPUs. The contributions are as follows:

1. We propose G-Scalar on GPU that can support scalar execution of not only non-divergent arithmetic/logic instructions, but also divergent and special-function instructions without requiring a dedicated scalar execution pipeline. G-Scalar can improve the power efficiency of a GPU similar to NVIDIA GTX 480 by 24% and 15%, compared with the baseline and previous scalar execution GPU architectures, respectively.

2. We propose Lock and Load architecture on GPU to perform coarse-grained, load-triggered approximation to reduce power consumption of the GPU. The coarse-grained approximation scheme allows us to implement our proposed Warp Fusion scheme to further improve GPU performance. Lock and Load architecture along with the Warp Fusion can improve energy efficiency and performance by 62% and 23% respectively, using a GPU configuration similar to NVIDIA GTX 480.

3. We propose AxMemo on CPU to efficiently perform general-purpose, approximate memoization. AxMemo replaces long sequences of instructions with a few lookup operations. It avoids the execution of those

instructions completely, not just the computation in execution units. AxMemo provides 2.64× average speedup and 2.58× average energy reduction using a high-performance in-order (HPI) ARM processor as baseline. These benefits come at the cost of 0.2% of average quality loss and 2.08% area overhead.

To achieve these, we need to address various of challenges. For example, tracking the value check results for register written by divergent instruction in G-Scalar, efficiently checking and tracking load instructions in Lock and Load and handling the variable number of inputs of the substituted computation blocks in AxMemo. All these challenges are discussed and addressed in this dissertation. The rest of the dissertation is organized as follows:

In Chapter 2, we first introduce the concept of scalar execution, which avoids redundant computation on identical operand values. Then we propose the G-Scalar architecture, a generalized scalar execution architecture that supports scalar execution on divergent instructions. We describe the details on how we implement efficient scalar detection, execution and register file access optimization. Finally we show the evaluation set up and results for G-Scalar.

In Chapter 3, we extend the idea of scalar execution from identical data to similar data. We then propose Lock and Load architecture to exploit data similarity. We study the application characteristics to justify our design decision for Lock and Load. Then we elaborate the design and implementation of Lock and Load, including load value checking logic, instruction set extension and so on. Finally we show the evaluation of Lock and Load.

In Chapter 4, we adapt the core idea of Lock and Load to CPU architecture and propose AxMemo, a general-purpose memoization scheme. We elaborate the architecture and hardware design, instruction set extension, lookup operations and lookup table implementation, and compiler support for AxMemo. Then we evaluate the effectiveness of AxMemo.

The last chapter, Chapter 5, concludes the dissertation.

CHAPTER 2

SCALAR EXECUTION ON GPU: G-SCALAR ARCHITECTURE

2.1 Background

2.1.1 GPU Architecture and SIMT Execution Model

GPU Architecture: A graphics processing unit (GPU) typically consists of a large number of small cores. These cores are simple, pipelined cores with pipeline stages like Fetch, Decode, Scheduling and Execution. The small cores are grouped into stream multiprocessors (SMs) with shared Fetch, Decode, and Scheduling (FDS) logic. Each SM can be view as a larger core with all execution units designed for single instruction, multiple data (SIMD) operations. Although designed for graphics processing, GPUs are also widely used to accelerate data-parallel general-purpose applications, therefore given the name general-purpose GPUs (GPGPUs). GPUs are known to be highly energy efficient for data-parallel applications compared with CPUs. There are mainly two reasons for the high efficiency of GPUs. First, to hide the operation latencies, GPUs do not have to use Out-of-Order (OoO) execution to exploit instruction-level parallelism. Instead, GPUs exploit the abundant data parallelism to issue independent instructions for hiding the latencies. Without the power-hungry OoO logic, GPUs show significant improvement in energy efficiency. Second, since GPUs are essentially SIMD processors, the shared front-end amortizes the overhead associated with instruction fetching, decoding and scheduling over many operations (typically 32 or 64). Together, these two factors result in the high efficiency of GPUs.

In this work, we consider a GPU architecture that is similar to NVIDIA GTX 480 as our baseline [38], shown in Figure 2.1. The baseline GPU architecture has 15 SMs. Each SM in the baseline GPU has a large 32768×4-byte register file (i.e., 1024 vector registers, each of which consists of thirty-two

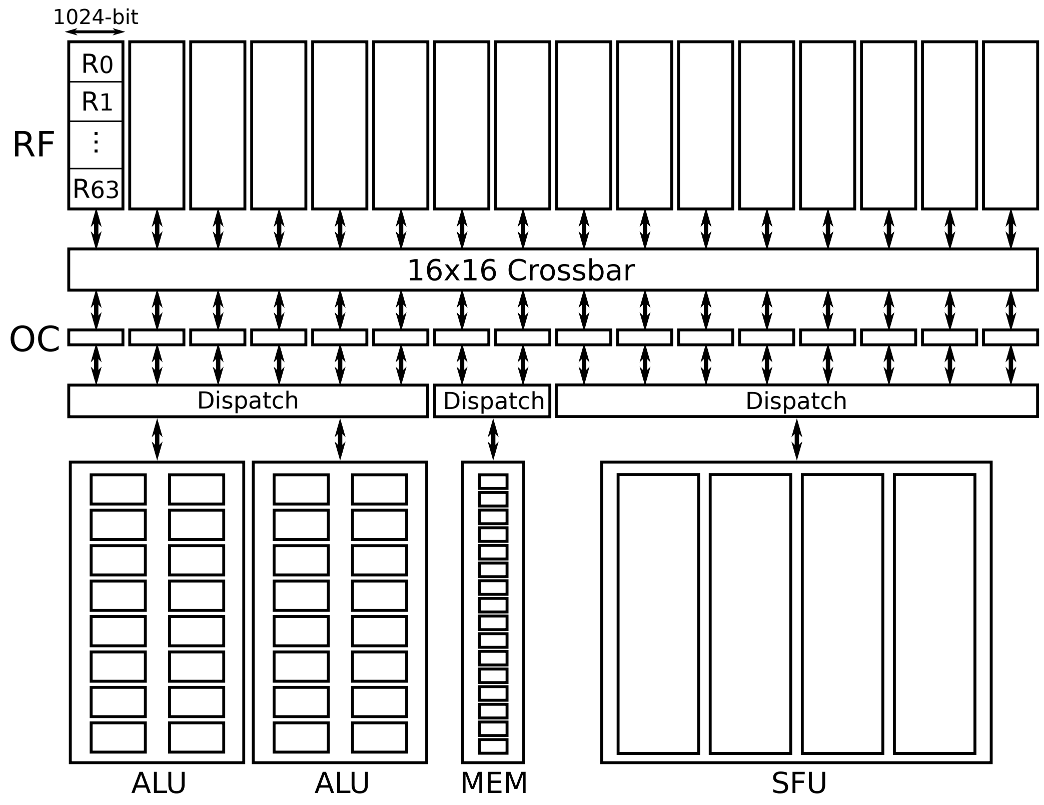


Figure 2.1: SM architecture of the baseline GPU similar to NVIDIA GTX 480 [38]. The major components include register file (RF), operand collector (OC), arithmetic logic unit (ALU), load/store unit (MEM) and special function unit (SFU).

4-byte registers). As a typical instruction operates on two or three vector registers, the register file is partitioned into 16 banks. This allows an instruction to access multiple vector registers in a single cycle with single-port SRAM arrays constituting each bank, but necessitates a 16×16 crossbar between banks and 16 operand collectors supplying operands to SIMD execution pipelines [39]. Each bank provides sixty-four 32×4-byte vector registers and consists of eight 64×128-bit single-port SRAM arrays. When the GPU accesses a bank, all eight SRAM arrays are activated. As all the operands are ready for an instruction, a scheduler dispatch the instruction to an appropriate SIMD execution pipeline [39].

Each SM has three types of SIMD execution pipelines: (1) two 16-lane arithmetic/logic, (2) one 16-lane memory, and (3) one 4-lane special-function pipelines. Depending on the width of each execution pipeline, a warp is dispatched to the arithmetic/logic, memory and special-function pipelines over two, two, and eight batches, respectively. Since each SM has two arithmetic/logic pipelines, up to two arithmetic/logic instructions can be dispatched in a single cycle.

SIMT Execution Model: To accommodate the underlying hardware architecture, GPUs use an execution model called single instruction, multiple thread (SIMT). The parts of the program that run on the GPU are called kernels. A kernel consists of a large number of threads, conceptually organized as a thread grid. All the threads have the same sequence of (static) instructions and each of them is only responsible for processing a small portion of the input data. Ideally, the threads should run with no or little dependencies with other threads, i.e. data parallelism. The thread grid is further divided into thread blocks, which is the unit of dispatching to SMs. Finally, the threads in thread blocks are grouped into 32-thread warps and executed in lock-step.

Since all threads have the same static code and threads in a warp are executed in lock-step (i.e. SIMD), GPUs need a mechanism to enable fine-grained control for each thread in case of the so-called control flow divergence. For example, in a kernel with an *if else* statement, half of the threads in a warp take the *if* path and the rest take the *else* path. In the CPU program, the thread can simply branch to the corresponding target address. However, this is not possible for the GPU kernels, since the threads in a warp share the front-end. Therefore, GPUs use a mask, called an active mask, to control each thread in a warp. Each bit of the active mask marks if the corresponding thread should be active or not for the current instruction.

As we mentioned earlier, GPUs do not have OoO logic. Therefore, GPUs utilize the large number of active warps to hide the instruction latencies. The mechanism is essentially the same as simultaneous multithreading (SMT). By interleaving the instructions from different active warps, the latencies of most of the instructions can be hidden. In other words, the GPUs keeps switching context among the active warps. To support fast context switch, each SM in the GPU has a large register file (RF) such that all the context of the active warps can be saved in the RF at the same time. Such design allows GPUs to switch the context among the active warps with no overhead.

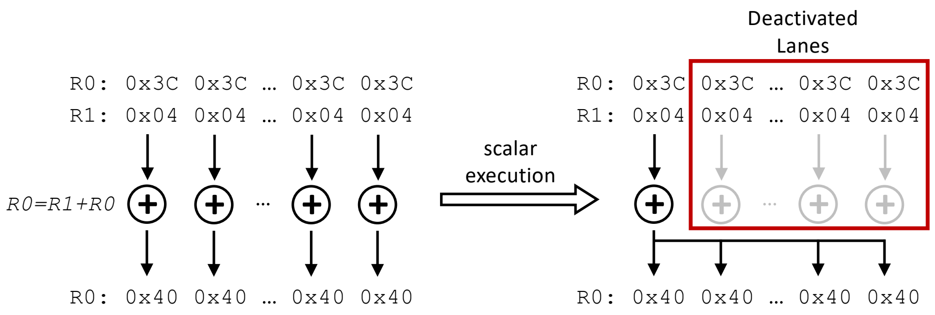

Deactivated Lanes
R0: 0x3C 0x3C … 0x3C 0x3C
R1: 0x04 0x04 … 0x04 0x04
R0=R1+R0
R0: 0x40 0x40 … 0x40 0x40
scalar execution
R0: 0x3C 0x3C … 0x3C 0x3C
R1: 0x04 0x04 … 0x04 0x04
R0: 0x40 0x40 … 0x40 0x40

Figure 2.2: A conceptual illustration of scalar execution. The SIMD instruction has two operands `R0` and `R1` that have same values and produce the same results across all SIMD lanes.

2.1.2 GPU Power Reduction Techniques

A detailed analysis shows that the execution units and register file are the two most power-consuming components in the GPU and consume about 24% and 16% of total GPU chip power, respectively [3]. In compute-intensive applications, the percentage is even higher. Consequently, a large body of work has been proposed to improve power efficiency of these two power-consuming components, exploiting various characteristics of applications (e.g., [5, 40, 6, 7]).

A GPU register file is typically comprised of 32 $\times$ 4-byte vector registers, each of which can supply source operands for 32 threads in a single warp [38]. It has been observed that the 32 (scalar) registers in a vector register often store the same (scalar) value [5, 6, 7] or similar values [40, 13] at runtime. Such value characteristics were exploited in two distinct ways to improve power efficiency of GPUs: (1) scalar execution of instructions [5, 6, 7, 13] and (2) register value compression [40].

Scalar Execution. For a SIMD instruction, we call an operand "scalar operand" when all the SIMD lanes have the same value for that operand, like the operand `R0` and `R1` in Figure 2.2. If all its operands are scalar operands, the instruction can use scalar execution to reduce the redundant computation. Conceptually, we can execute the instruction in only one of the SIMD lanes and deactivate the other lanes, then duplicate the results for the deactivated lanes.

To reduce redundant computations and thus power consumption of execution pipelines, prior work proposed various scalar execution architectures to execute such instructions using a dedicated scalar execution pipeline [5, 6, 7].

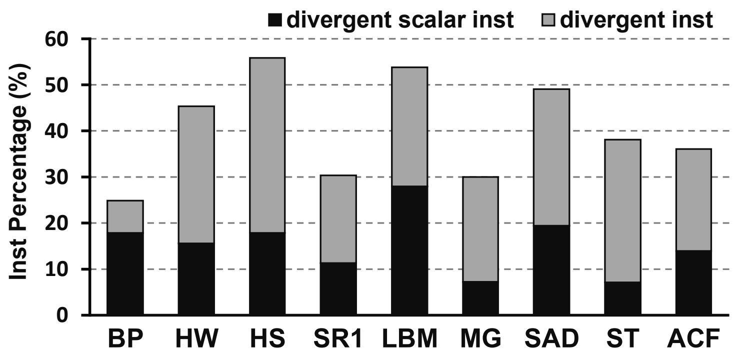


Figure 2.3: Percentage of divergent instructions and divergent scalar instructions in total instructions.

The scope of scalar execution was expanded to instructions operating on vector registers storing similar values [13] although the benefit is primarily from operating on scalar values.

As GPUs began to support more general-purpose applications, the fraction of divergent instructions in contemporary GPU applications has significantly increased, and various optimization techniques have been proposed to efficiently handle them (e.g., [41, 42, 43, 44, 45, 46]). Specifically, we observe that many divergent instructions are also eligible for scalar execution (denoted by divergent scalar instructions) if we consider only 4-byte register values in active lanes in a divergent path. Figure 2.3 shows that 28% of total instructions are divergent instructions and 45% of total divergent instructions are divergent scalar instructions in these benchmarks.

Despite such a high percentage of divergent scalar instructions in contemporary GPU applications, prior architectures do not support scalar execution of divergent instructions. Furthermore, they do not consider scalar execution of special-function instructions such as sin, cos and exp, as implementing a separate Special-Function Unit (SFU) for a scalar execution pipeline incurs a very high chip cost. For example, NVIDIA GTX 480 provides only one 4-lane SFU per SM [38] partly due to its chip cost, while special-function instructions consume 3$\sim$24$\times$ more energy than typical arithmetic/logic instructions [3]. Hence, ignoring divergent code regions or special-function instructions while exploiting register value locality will undeniably reduce the effectiveness of prior techniques.

Data Compression. Data compression also exploits the data similarity to reduce power consumption. For on-chip memory value compression, the base-delta-immediate (BDI) compression scheme [47] has been used [40, 48, 49, 50]. It exploits the observation that values stored in a cache line [47] or a vector register [40] are similar. For example, if eight 4-byte values from a given SIMT pipeline are `0xC04039C0`, `0xC04039C8`, ..., and `0xC04039F8`, the first value becomes the base value, and `0x00`, `0x08`, ..., and texttt0x38 become the delta values for the eight 4-byte values. Consequently, a 256-bit (=8×4-byte) value can be compressed to a 96-bit value (i.e., 32-bit base and 8×8-bit delta values, respectively) and stored in a vector register in a compressed format. Such an implementation of the compression hardware for BDI typically requires N 32-bit adders/subtractors [47] where N is the number of 4-byte values in a cache line or a vector register.

In this work, we propose G-Scalar, a GPU architecture to cost-effectively support generalized scalar execution of instructions. Prior scalar execution architectures require a dedicated scalar execution pipeline [5, 6, 7], and/or only support scalar execution of non-divergent arithmetic/logic instructions [5, 6, 7, 13]. Consequently, supporting the scalar execution of only non-divergent arithmetic/logic instructions may significantly limit the benefit for many GPU applications. In contrast, G-Scalar can support scalar execution of not only non-divergent arithmetic/logic instructions but also divergent and special-function instructions without any dedicated scalar execution pipeline, as it can share most hardware resources with our register value compression scheme. In particular, when a GPU adopts our low-cost register value compression technique, G-Scalar can support generalized scalar execution practically at no cost.

2.2 Register Value Compression

Our key contribution is G-Scalar, generalized scalar execution architecture that supports scalar execution of not only non-divergent arithmetic/logic but also divergent and special-function instructions. Before explaining G-Scalar in depth, however, we need to describe our enhanced GPU microarchitecture to cost-effectively support a register value compression technique, as G-Scalar is built upon this microarchitecture enhancement.

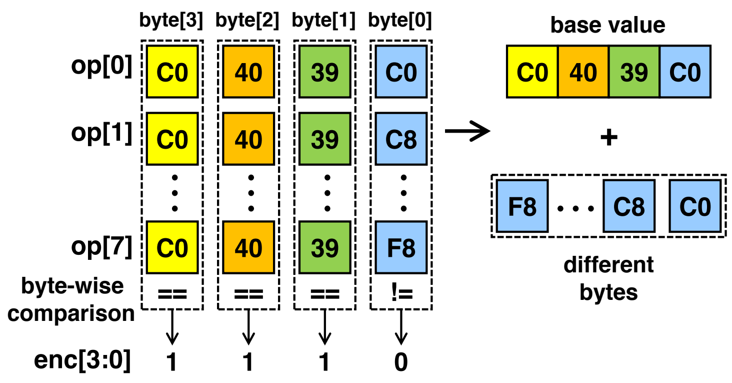


Figure 2.4: Proposed compression technique. Byte-wise comparison results are stored in compression bits `enc[3:0]`. `1 (0)` means all bytes are (not) the same. `op` denotes an operand stored in a 4-byte register.

2.2.1 Compression Scheme

Exploiting value similarity, we first propose a register value compression technique with a lower cost than BDI [47]. Using the same example in Section 2.1, we illustrate our compression technique in Figure 2.4. Instead of subtracting the base value from each operand, our compression technique directly compares all 4-byte values in a vector register byte by byte to check whether or not `byte[i]` for every `op` has the same value. In this example, the byte-wise comparison determines that `byte[3]` (=`0xC0`), `byte[2]` (=0x40), and `byte[1]` (=0x39) have the same values across `op[0]`, `op[1]`, ..., `op[7]`. That is, `0xC04039` becomes the base value, and `byte[0]` from each `op` (= `0xF8`, ..., `0xC8`, `0xC0`) becomes the delta value for each `op`. Consequently, our compression technique stores the 3-byte base value and 8 different bytes with the corresponding encoding bits (1110_2). In our compression technique, the base value can be up to 4 bytes (for a vector register storing a scalar value), and we always use bytes from `op[0]` for the base value for simplicity. Note that our technique is more efficient than BDI in hardware implementation although it does not provide the same compression ratio as BDI in some special cases, especially when the hexadecimal representation of two similar values differs widely.

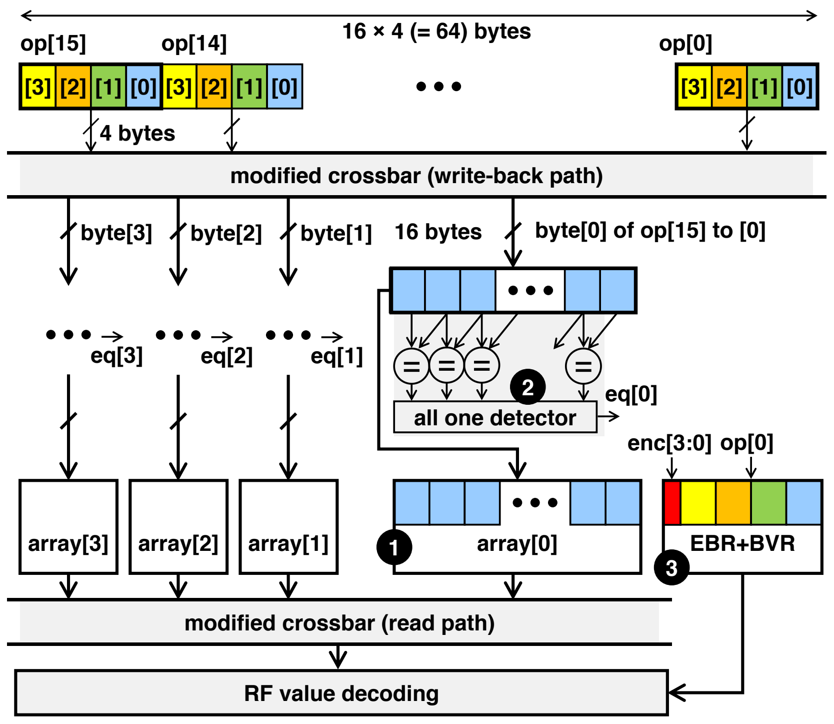


Figure 2.5: Microarchitecture to support register value compression for 16-lane SIMT. `BVR` and `EBR` denote base value register and compression bit register, respectively. `op`, `enc`, `BVR` and `EBR` denote operand, compression bit, base value register and compression bit register.

2.2.2 Microarchitecture Support

Figure 2.5 depicts necessary microarchitecture changes to support our register value compression technique for a hypothetical 16-lane SIMT pipeline. In the baseline register file, a bank is comprised of four SRAM arrays (eight for a 32-lane SIMT pipeline), each of which stores four 4-byte values. Suppose that we desire to retrieve only `byte[0]` of each 4-byte value to reconstruct sixteen 4-byte values. In a traditional register file, `byte[0]` of sixteen 4-byte values are distributed across all four arrays. Thus, we still need to activate all four arrays.

To support power-efficient accesses of the register file for our register value compression technique, we propose to reorder bytes such that an array stores only `byte[i]` of all sixteen 4-byte values in a vector register, as illustrated in Figure 2.5 (❶). This allows us to activate only one array to retrieve/store `byte[i]` of all sixteen 4-byte values. For example, we only activate `array[0]` to retrieve/store `byte[0]` of all sixteen 4-byte values (`op[0]`, `op[1]`, ...,

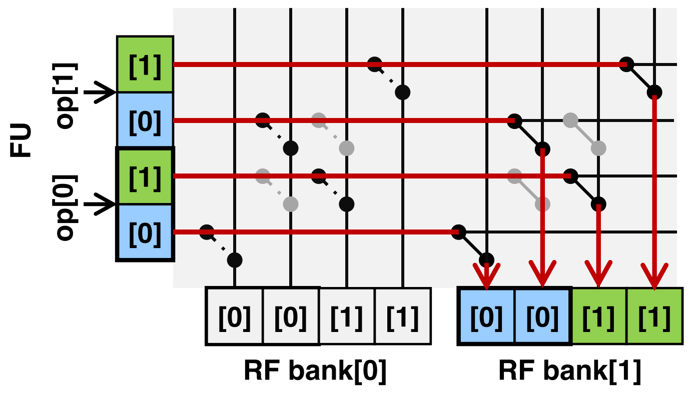


Figure 2.6: Adapted crossbar for reordering bytes. `FU` denotes a hypothetical 2-lane SIMT pipeline.

`op[15]`) from/to a bank. However, we also need to reorder the bytes back to the standard order before sending them to the SIMT pipeline.

After analyzing the baseline GPU architecture, we propose a slight adaptation of the existing crossbar between banks and operand collectors to cost-effectively reorder bytes. Figure 2.6 illustrates a part of an adapted crossbar (write-back path) designed for a hypothetical 2-lane SIMT pipeline and two banks of 2×2-byte vector registers. The gray switches represent switches in the traditional crossbar and the red arrows depict the flow of values. This adaptation simply rearranges the connection points of the crossbar switches, practically incurring no penalty for performance, power or space. Furthermore, bytes of each 4-byte value corresponding to the base value of a vector register are not stored to the register file after compressing the vector register value, and thus they are not sent over the crossbar when retrieved. Consequently, our compression scheme reduces not only the power consumption of accessing vector registers but also that of sending values through a large crossbar. Lastly, we need minor adaptations in the arbiter and control signals of the switches. Nonetheless, we see that such adaptations are insignificant. Note that we always store bytes to a bank in a reordered way whether or not a vector register is compressed in this crossbar architecture. Hence, values stored in registers are oblivious to the compression technique, significantly simplifying the complexity to control the circuit.

In contrast, the register value compression based on BDI [40] requires a relatively complex interconnect network to pack (unpack) compressed (decompressed) values, as the delta part can have diverse sizes for different registers, such as 1-byte for register 1 and 3-byte for register 2. As the number of bytes for a single compression operation (e.g., a vector register or a cache line) increases, the complexity of the interconnect network increases even more significantly. Note that BDI was originally proposed for cache value compression where a cache line is comprised of 32 bytes [47], whereas a vector register consists of 128 and 256 bytes for NVIDIA and AMD GPUs, respectively [38, 51].

The comparison logic depicted in Figure 2.5 (❷) generates encoding bits (`enc[3:0]` in Figure 2.5) that are used not only to determine which array(s) should be activated but also to decompress a compressed vector register value. The comparison logic is almost the same as the logic depicted in prior work [5], but it compares values byte by byte instead of 4-byte word by word. Our circuit-level analysis shows that one cycle is sufficient for the comparison logic to generate its `eq` signal, assuming a typical implementation of all one detector. Then `eq[3:0]` signals are encoded such that `enc[3:0]` bits store 0000_2, (no byte is the same across all 16 4-byte values), 10002 (`byte[3]` is the same), 11002 (`byte[3:2]` is the same), 11102 (`byte[3:1]` is the same), and 11112 (`byte[3:0]` is the same or a vector register storing a scalar value). Lastly, we store `enc[3:0]` and the base value of a vector register in an encoding bit register (`EBR` in Figure 2.5) and a base value register (`BVR` in Figure 2.5), respectively, as illustrated in Figure 2.5 (❸); we simply store the first operand value (`op[0]` in Figure 2.5) of a vector register to a base value register.

To decode sixteen 4-byte operands in the example illustrated in Section 2.2.1 (i.e., `byte[3:1]` is the same across sixteen 4-byte values), we activate only `array[0]` and a small array storing a 4-byte base value and 4-bit encoding bits (`BVR` and `EBR`), instead of all four arrays (`array[3:0]`). For a vector register storing a scalar value, we activate only the small array storing the base value and encoding bits. The retrieved bytes from the activated array(s) are reordered back to the standard order after they go through the adapted crossbar. Then the bytes, which are not stored in the register file, are replaced with the corresponding bytes from the base value register by the decompression logic illustrated in Figure 2.7. In the example depicted in Section 2.2.1, the decompression logic selects `byte[3]`, `byte[2]`, and `byte[1]` from the base value register, while only `byte[0]` for each operand is from `array[0]` based on the encoding bits ($=1110_2$). For the decompression logic,

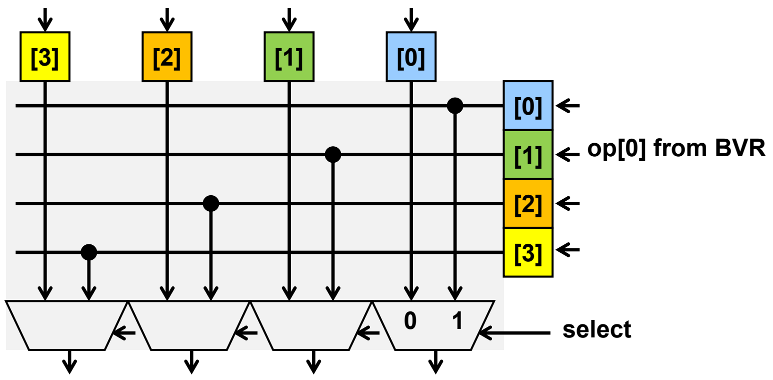


Figure 2.7: Decompression logic. `select` is the selection signal generated using the four compression bits in `EBR`.

our conservative circuit-level analysis shows that one cycle is sufficient for the decompression logic. Note that even the baseline GPU deploys some circuits to broadcast a value from the memory pipeline to all SIMT lanes, e.g., when a warp issues a load instruction to the global memory and all threads in the warp attempt to load a value from the same address.

In the example above, a bank for 16-lane SIMT pipelines is comprised of four 16-byte wide arrays. In our experiment, however, a memory compiler synthesizes a bank consisting of eight arrays with 16-byte (= 128-bit) I/O width, when it attempts to satisfy the timing constraint for given capacity (= 8 KB), number of ports (= 1), and I/O width (= 1024 bits). That is, we need to use two arrays to store `byte[i]` of all thirty-two 4-byte values of a vector register. Since these two arrays can be activated independently, we can optionally apply our register value compression technique to each half of a vector register separately, denoted by half-register value compression. This increases chances to partially compress more vector registers and support more fine-grain scalar execution for some 32-lane SIMT architectures such as Fermi [38] comprised of 16-lane pipelines at the cost of providing one more set of base value and encoding bit registers per vector register. We will describe how we can exploit this for more fine-grain scalar execution in Section 2.3.3.

2.2.3 Handling Branch Divergence

The register value compression technique described in Section 2.2.1 seamlessly works for non-divergent instructions. For a divergent instruction, only some threads in a warp will be active and they will perform partial writes to a vector register. Consequently, the baseline GPU supports per-word writes to efficiently update vector register values for divergent instructions. For compressed vector registers, however, we cannot perform partial updates unless we decompress it. That is, we always need to retrieve all the values of a vector register before we compress it again, demanding read-modify-write (RMW) operations.

Considering a high energy and performance penalty of such RMW operations, we simply do not encode vector register values for divergent instructions. Although encoding bits are still generated, they are ignored by another bit (denoted by D as in "divergent" and affixed to `enc[3:0]`). These encoding bits are used to support scalar execution of divergent instructions described in Section 2.3.2. Lastly, vector registers, which are already encoded by previous non-divergent instructions, still can be decoded. If a divergent instruction attempts to update values of a compressed vector register, we can employ either hardware- or compiler-assisted techniques.

As a hardware-assisted technique, we can check whether or not the destination vector register of a currently executed instruction is compressed at the scoreboard stage. The scoreboard needs to check the active mask of the instruction and the encoding bits of the destination vector register. If the instruction is divergent and the destination vector register is encoded, the GPU inserts a special register-to-register move instruction retrieving/decompressing the compressed destination vector register value and store it back to the vector register without compressing it. This special move instruction is designed to temporarily ignore the active mask. Prior work reports that the hardware-based technique increases the number of dynamic instructions by only 2% on average [40]. In addition to such a hardware-only approach, a compiler-assisted technique can analyze the lifetime of registers at compile time and identify which registers will store dead values [52]. This information can avoid unnecessary special move instructions. Leveraging such compile-time information, we may further reduce the overhead to less than 2%.

As discussed earlier in this section, a divergent instruction needs to perform partial updates to a vector register. The baseline GPU achieves this by activating write-enable signals associated with active lanes; each array has four write-enable signals for four 4-byte values. Since all 16 bytes of `byte[i]` are stored in one array in our compression technique, we need write-enable signals for each byte in an array. Note that this does not change the number of logical write-enable signals, but each 128-bit wide array needs 16 physical write-enable signals. Analyzing circuit implementations of the write datapath of arrays, we discover that providing 16 write-enable signals does not require any change in the SRAM core. That is, we only need to connect more write-enable signals to write-drivers of the SRAM I/O peripheral block in a finer-grained way. Our circuit-level analysis shows that more write-enable wires increase the area of a large memory array by less than 1%. This is because the I/O circuit has a tight width pitch but a loose height pitch where the write-enable signals run horizontally. Lastly, our compression technique activates all four arrays for a divergent instruction partially updating its destination register, as such a partial update is applied to a decoded vector register value, and each byte of a 4-byte value is distributed across four arrays. In contrast, the baseline architecture may activate fewer arrays for a partial update depending on a given active mask value (`M`). This effect will be accounted for our evaluation.

2.3 Scalar Execution Architecture

Prior work demonstrated that scalar execution of eligible non-divergent arithmetic/logic instructions could significantly improve the performance and power efficiency of GPUs [5, 6]. In this section, leveraging the enhanced microarchitecture for our register value compression, we demonstrate that G-Scalar can support scalar execution of eligible divergent and special-function instructions. This greatly increases the percentage of scalar execution of instructions practically at no cost.

2.3.1 Efficient Scalar Execution

Our register value compression technique not only reduces the energy consumption of register file and its crossbar but also easily determines how similar the values of a vector register are. A vector register storing a scalar value has its encoding bits (`enc[3:0]`) equal to 1111_2. This allows us to easily support a scalar execution approach proposed by prior work [5] at practically no further hardware cost, because a base value register becomes effectively a scalar register. When a non-divergent instruction operates on two vector registers storing scalar values, only two 4-byte values are sent from the corresponding base value registers to an appropriate SIMT pipeline and only one execution lane in that SIMT pipeline will be active. Similar to prior work [5], scalar execution of an instruction only needs to write-back its computed value to a base value register and sets the encoding bits of the destination vector register to 1111_2. Lastly, when at least one of vector registers for an instruction is not a vector register storing a scalar value, the instruction is not eligible for scalar execution, and the decompression logic automatically decompresses vector registers storing scalar values.

Although the key high-level concept of identifying and executing scalar instructions is the same as previous scalar execution architectures [5, 6, 7], G-Scalar has notable advantages over previous scalar execution architectures. First, G-Scalar effectively provides 16 banks for scalar values because each bank has its own small array for base value registers. In contrast, we observe that a single bank for scalar values in prior scalar execution architectures can be a performance bottleneck for applications with many instructions eligible for scalar execution. An instruction can access all of its vector registers in parallel if there is no bank conflict. However, it always takes multiple cycles for an instruction eligible for scalar execution to retrieve its register values because there is only one bank for scalar values. Furthermore, when one warp issues a scalar instruction, we observe that other warps are also likely to issue scalar instructions. This is because the warps are executed roughly at the same pace. Therefore, there can be a burst of scalar instructions, all of which wait for accessing a single bank for scalar values at the operand collectors. If all operand collectors are allocated, the warp schedulers must stop issuing instructions from ready warps. With semiconductor technology scaling, future GPUs also tend to have more hardware resources, such as a larger register file with more banks and more SIMT execution pipelines. Thus, relying on only a single bank for scalar values may not be a scalable approach.

Second, G-Scalar does not need separate scalar execution pipelines. Instead, it leverages existing SIMT execution pipelines and clock-gates all but one lane for scalar execution [3]. Note that prior work estimates that the cost of supporting per-lane clock gating is very small [3]. We identify a few

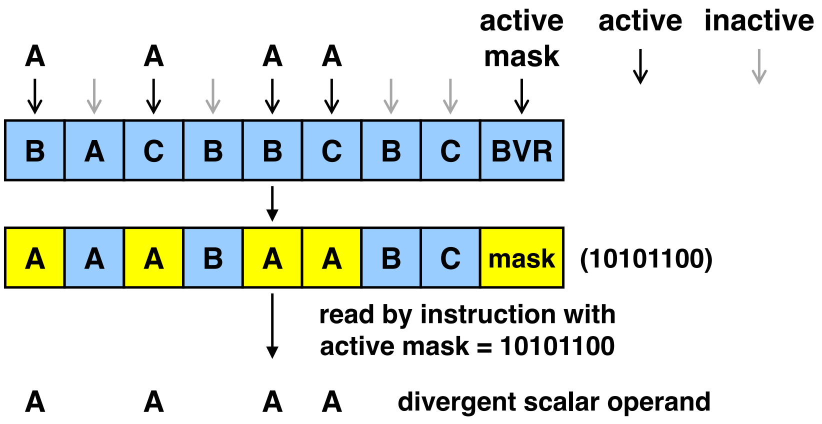


Figure 2.8: An example for divergent scalar value. `M` denotes active mask. A vector register of eight values is updated partially first, then read by another divergent instruction.

reasons to obviate from implementing separate scalar execution pipelines. Since the front-end can schedule and issue only up to two instructions per cycle, a separate scalar execution pipeline may bring only little performance improvement because the front-end will soon become a performance bottleneck. Furthermore, implementing even one more SFU incurs a considerable penalty of chip power and space. For example, a GTX 480 GPU has only four SFUs per SM because each SFU consumes a large amount of chip power and space. This is why a previous architecture implements the scalar execution pipeline only for arithmetic/logic instructions. In contrast, G-Scalar can support scalar execution for any vector instructions, because it uses SFUs that are already in the SIMT pipelines.

2.3.2 Scalar Execution of Divergent Instructions

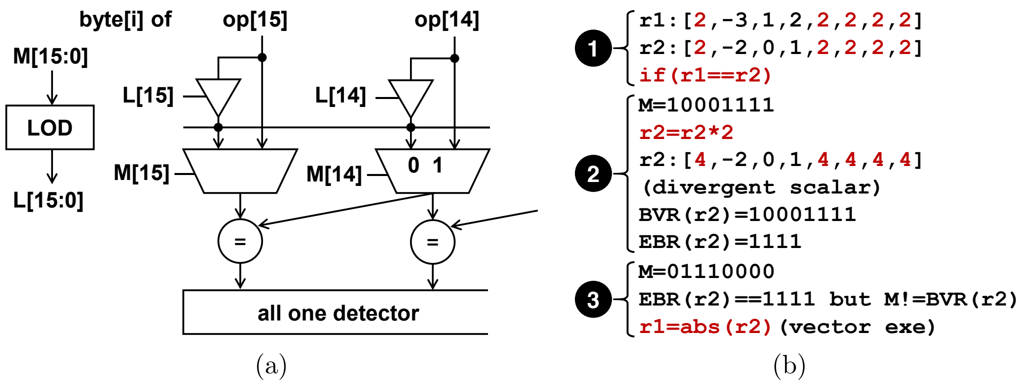
No prior scalar execution architecture can support scalar execution of divergent instructions [5, 6, 7, 13]. However, we observe that approximately 50% of executed instructions are divergent in some GPU applications such as `lbm` [53] and `heartwall` [54]. Meanwhile, we discover that 4-byte values, which are associated with active lanes in a divergent path, are often the same in a vector register.

Suppose that a vector register has eight values, AAABAABC. We cannot consider that such a vector register stores a scalar value for a non-divergent instruction. Nonetheless, we may consider that the vector register stores a scalar value for a divergent instruction with an active mask (`M`) value equal to 10101100 (i.e., A-A-AA–), as shown in Figure 2.8. To support scalar execution for such divergent instructions, we need to slightly adapt the comparison logic to correctly compare the (partial) write-back values of a vector register updated by a divergent instruction and detect whether a given divergent instruction is eligible for scalar execution.

As depicted in Figure 2.5 (❶), our comparison logic for compression compares each byte of a 4-byte write-back value to that of a 4-byte write-back value in its neighboring lane. Such an implementation makes the comparison of partial write-back values impossible for a divergent instruction, since the inactive lanes of a divergent instruction do not have any valid write-back values. This breaks the comparison chain in the comparison logic. However, we tackle this challenge based on the following observation: providing a 4-byte value from any active lane for values of inactive lanes should not change the outcome of comparison, because we only check whether or not all active lanes have the same write-back value. Suppose that a given SIMT execution pipeline has four lanes, while `lane[0]`, `lane[1]`, and `lane[3]` are active. The comparison of `op[0]`, `op[1]`, and `op[3]` is equivalent to that of `op[0]`, `op[1]`, `op[0]`, and `op[3]`. We can accomplish such a comparison by slightly adapting the comparison logic detailed in Section 2.3.3.

Figure 2.9a shows the adapted comparison logic. We have a shared byte-wide bus that can be driven by a write-back value from only one of active lanes. The logic detecting the leading one of a given 16-bit active mask value (`M`) (e.g., `L[15:0]` = 0100000000000000_2 for `M[15:0]` = 0100000011111010_2) can enable only one tristate buffer connected to the write-back value from the first active lane (i.e., `lane[14]`). This allows us to send a write-back value from an active lane to all inactive lanes just for a comparison purpose and check whether or not active lanes operate on the same 4-byte value. Since the comparison logic has enough timing slack, the adapted comparison logic can still complete a comparison in one cycle according to our circuit-level analysis.

As described in Section 2.2.3, we do not compress the destination register of a divergent instruction. Nonetheless, we still generate its encoding bits and store them to its encoding bit register. This is to indicate whether or not 4-byte values, which correspond to active lanes of subsequent divergent instructions, are the same (i.e., a vector register of a divergent scalar value).



```
r1:[2,-3,1,2,2,2,2,2]
r2:[2,-2,0,1,2,2,2,2]
if(r1==r2)
M=10001111
r2=r2*2
r2:[4,-2,0,1,4,4,4,4]
(divergent scalar)
BVR(r2)=10001111
EBR(r2)=1111
M=01110000
EBR(r2)==1111 but M!=BVR(r2)
r1=abs(r2)(vector exe)
```

Figure 2.9: (a) The adapted comparison logic for divergent instructions. (b) An example of checking active mask (denoted by `M`).

For example, `if(r1==r2)` in Figure 2.9b (❶) starts a branch divergence; Figure 2.9b (❷) and (❸) illustrate two following divergent paths. Since `r2` = `r2*2` stores a (divergent) scalar value with respect to a given active mask value in Figure 2.9b (❷), the corresponding encoding bit register (`EBR(r2)`) is set to 1111_2 indicating that `r2` stores a divergent scalar value. However, `r2` stores a scalar value only with respect to the current active mask value (`M` = 10001111). The following instruction (`r1` = `abs(r2)`) in Figure 2.9b (❸) is on the other divergent path and operates on `r2`. Although `EBR(r2)` indicates that `r2` contains a scalar value, we cannot perform scalar execution for `r1` = `abs(r2)` because the encoding bits of `r2` are invalid with respect to the current active mask (`M` = 01110000). From this example, we can see that the operand values in the active lanes (`-2`, `0`, `1`) are indeed different.

To correctly determine whether or not a current instruction operates on vector registers of divergent scalar values with respect to its active mask, we need to remember which lanes were compared to generate the corresponding encoding bits (i.e., the active mask of the previous instruction that wrote its values to the register). Since we do not encode values of a divergent destination register, we do not need to store its base value to the corresponding base value register. Exploiting this, we propose to store the associated active mask to its base value register (Figure 2.9b (❷)).

Depending on whether or not a register is updated by a non-divergent or divergent instruction (`D` = 0 or 1), the interpretation of `enc[3:0]` may change and the corresponding base value register may store a base or active mask value. When `D` is set to 1, we do not actually compress register values.

Consequently, we ignore the corresponding encoding bits and bring all values from the register file. However, when D and enc[3:0] are set to 1 and 1111_2, respectively, we compare the active mask value of a current instruction with the value of a base value register corresponding to a vector register that the instruction operates on (Figure 2.9b (❸)). If these two values are matched, the source vector register contains a (divergent) scalar value. Note that we still need to bring all 16 operand values from the register file, but we activate only one lane for scalar execution. Otherwise, we cannot perform scalar execution for the instruction although enc[3:0] is set to 1111_2.

Lastly, scalar execution of divergent instructions is the same as scalar instructions that are not divergent (i.e., only one lane is active), but retrieving/storing values from/to the register file is different. As we disable the register value compression for divergent instructions, we store a scalar value to the register file without compressing the register value. In particular, we leverage an existing mechanism depicted in Figure 2.5 to broadcast a divergent scalar value to the write paths associated with all active lanes. Although a source vector register stores the same value with respect to a given active mask, we still retrieve the value from the register file.

2.3.3 Half-Warp Scalar Execution

As described in Section 2.2, we can optionally compress each half of a vector register separately, providing one more pair of base value and encoding bit registers for each vector register. Leveraging a half-warp execution architecture in some GPUs [38] and our half-register value compression technique, we can support half-warp scalar execution. Suppose that enc[3:0] of the first half of a vector register (denoted by encL[3:0]) is 1100_2 but enc[3:0] of the second half of the vector register (encH[3:0]) is 1111_2. In previous scalar execution architectures, we cannot support scalar execution of such an instruction, as some values of the first half of the vector register are not the same. In contrast, we can still support scalar execution of such an instruction for the second-half warp.

In some occasions, the first and second halves of a vector register are scalar each, but they have two distinct scalar values. In contrast to traditional scalar execution architectures, G-Scalar can support scalar execution for each half

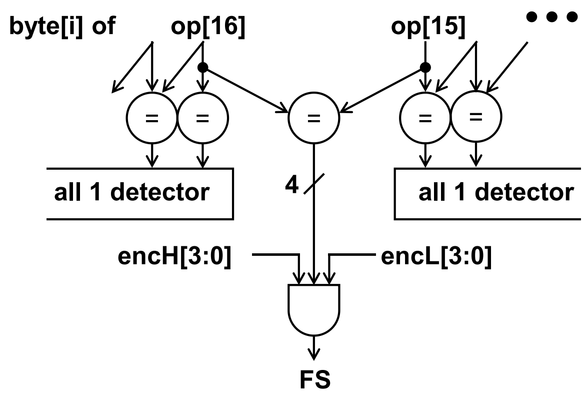


Figure 2.10: The adapted comparison logic for half-warp scalar execution. FS denotes "full scalar".

warp. To indicate whether or not the first and second halves have the same value, we need another flag bit (FS in Figure 2.10); the modified comparison logic for this architecture is depicted in Figure 2.10. This is necessary for scalar execution of a full-warp using only one lane for all 32 threads.

Note that the enhanced microarchitecture for half register value compression can seamlessly handle the write-back from scalar execution of a half warp. Moreover, it allows us to support half-warp scalar execution at practically no further hardware cost. Considering the complexity and benefit, we support half-warp scalar execution only for non-divergent instructions in this study. Lastly, this half-warp scalar execution allows even future GPUs with wider SIMT pipelines (meaning possibly fewer full-warp scalar instructions) to continuously benefit from scalar execution. However, implementing one more set of base value and encoding bit registers increases the hardware cost of the register file from 3% to 7%.

2.4 Evaluation

2.4.1 Methodology

We use GPGPUSim 3.2.2 [55] and GPUWattch [3] to evaluate the effectiveness of G-Scalar and our register value compression technique. GPGPUSim is configured to model a GPU architecture similar to NVIDIA GTX480 [38]. The key configuration parameters are tabulated in Table 2.1. GPUWattch

Table 2.1: Simulator configuration.

# of SMs	15	Registers per SM	128KB
SM Frequency	1.4GHz	Register File Bank	16
NoC Frequency	0.7GHz	OC per SM	16
Warp Size	32	Schedulers per SM	2
SIMT EXE Width	16	L1$ per SM	16KB
Threads per SM	1536	Memory Channels	6
CTAs per SM	8	L2$ Size	768KB

Table 2.2: Benchmarks.

Rodinia		**Parboil**	
Benchmark	**Abbr.**	**Benchmark**	**Abbr.**
`b+tree`	**BT**	`cutcup`	**CC**
`backprop`	**BP**	`lbm`	**LBM**
`heartwall`	**HW**	`mri-grid`	**MG**
`hotspot`	**HS**	`mri-q`	**MQ**
`leukocyte`	**LC**	`sad`	**SAD**
`pathfinder`	**PF**	`sgemm`	**MM**
`srad_1`	**SR1**	`spmv`	**MV**
`srad_2`	**SR2**	`stencil`	**ST**
		`tpacf`	**ACF**

is also configured to estimate the power consumption of G-Scalar GPU.

We use 17 benchmarks from Parboil [53] and Rodinia [54] benchmark suites that represent diverse GPU applications in Table 2.2. Although one of our key contributions is scalar execution of divergent instructions, we exclude exceptionally divergent benchmarks (e.g., `myocyte`), while including a fair number of non-divergent benchmarks (e.g., `mri-q`, `sgemm`, `spmv`). Lastly, we exclude unusually memory-intensive benchmarks (e.g., `bfs`), as the performance and power consumption are dominated by the memory subsystem, whereas scalar execution architecture aims to improve power efficiency of compute-intensive applications.

We synthesize the compressor and decompressor logic with a commercial 40nm standard cell library. The results shown in Table 2.3 include the additional 1024-bit pipeline registers for compressor and decoder each. Since we have one decompressor for each operand collector and one compressor for each SIMT execution pipeline, we need 16 decompressor and 4 compressor per SM. These compressor and decompressor increase the chip power and

Table 2.3: Estimation of encoder/decoder area, delay and power consumption including that of the pipeline registers at 1.4 GHz. The encoder includes the broadcasting logic depicted in Figure 2.9.

	Decompressor	Compressor
Area (μm^2)	7332	11624
Delay (ns)	0.35	0.67
Power (mW)	15.86	16.22

space by 0.32 W (1.6%) and 0.16 mm^2 (0.7%) respectively, compared to the baseline SM. Considering the small cost of chip power and space, we do not place-and-route the compressor and decompressor implementations, as the conclusion of this study will not notably changes. Lastly, as our compression technique is simpler than BDI, our compressor logic associated wires consume only 19% and 30% of prior work [40].

In order to estimate the access energy of register file, we use a memory compiler, which determines the size of an array constituting a 64 $\times$ 1024-bit bank for given capacity (= 8 KB), number of ports (= 1), and I/O width (= 1024 bits). This gives us a bank comprised of 8 $\times$ 128-bit arrays with 128-bit I/O width, agreeing to the size used for prior work [56]. For 32-bit base value registers, 4-bit encoding bit registers, and `D` and `FS` bits, we synthesize a 64 $\times$ 38-bit array. The energy consumption of accessing a 38-bit register in this array is 5.2% of that of accessing an entire 1024-bit vector register in a bank. This register file architecture needs practically no modification to the baseline register file architecture except for only extra write-enable signals. The memory array for base value registers, encoding bit registers, and `D` and `FS` bits increases the size of register file by $\sim$3%.

Lastly, the encoding bits should be known before reading the register file to determine which arrays should be activated. This increases the pipeline latency by one cycle, but the throughput of register file is not affected as we pipeline such an operation. In total, we increase the GPU pipeline latency by three cycles (i.e., one cycle each for (1) compressing a register value; (2) decoding a register value; and (3) accessing a base value register, encoding bit register and flag bits).

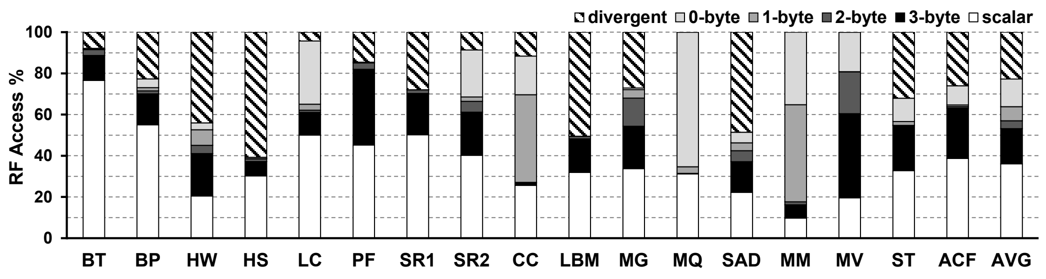


Figure 2.11: RF access distribution for operand values. Here “scalar” means all the operands are identical and “n-byte” means the first n MSBs of the operand values are the same, and “divergent” means the operands are accessed by a divergent instruction.

2.4.2 Register File Compression

Figure 2.11 plots value similarity of vector registers in these benchmarks. We collect values of vector registers by executing the benchmarks. We compare all thirty-two 4-byte values of each vector register byte by byte. “n-byte” denotes that the first n most significant bytes of all 4-byte values in a vector register are the same. As seen in Figure 2.11, the average percentage of (non-divergent) scalar, 3-, 2-, and 1-byte categories are 36%, 17%, 4%, and 7%, respectively.

We also show the dynamic power consumption of register file in Figure 2.12. More specifically, we compare the dynamic power consumption of our register value compression technique with that of scalar-only register file [5] and recent register value compression techniques [40]. The scalar register file consumes 37% less dynamic power than the baseline register file, whereas our register file consumes 54% less dynamic power on average. That is, our register file consumes about 17% less dynamic power than the scalar register file (46% vs. 63%). In some benchmarks such as `MG` and `MV`, where there are relatively fewer scalar values and many 3-byte and 2-byte accesses, our register value compression technique can reduce the dynamic power consumption of register file by more than 40% compared with the scalar register file technique. Our register compression scheme also reduces more dynamic power consumption than the prior register value compression technique [40] at a lower penalty of chip power and space. The chips space required by our compression technique is only 52% of the chip space demanded by the prior compression scheme [40] according to our logic synthesis of both compression techniques. Lastly, as our compression technique uses a separate small array

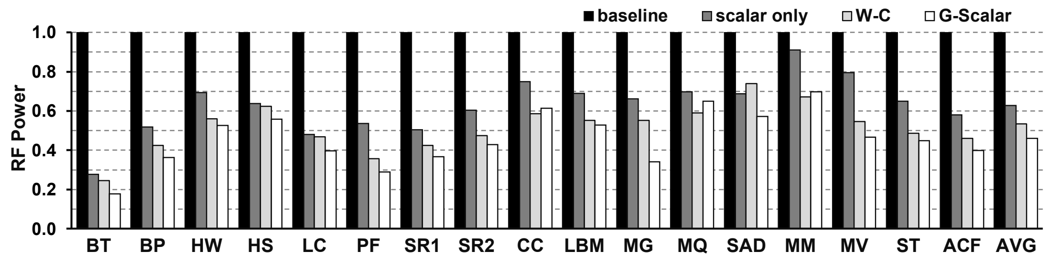


Figure 2.12: Normalized RF dynamic power consumption. Here “scalar only” is a scalar RF technique proposed in [5] and “W-C” is the warped-compression technique [40].

to store base values, it often activates one fewer array of register file than the prior compression technique for the same compression ratio, and the average compression ratio of our compression technique is 2.2, whereas that of BDI is 2.1 when the same input datasets for the benchmarks are used. Furthermore, the simplicity of our compression technique in fact facilitates G-Scalar execution at practically no further cost.

In our experiment, we use 32-bit unsigned integers for all address computations. However, recent GPUs began to support larger DRAM capacity. Consequently, GPUs need to use 64-bit integers for address computations if they support the DRAM capacity larger than 4 GB. In this case, we can obtain more power reduction with our register value compression technique. As mentioned earlier, it is very likely that only few LSBs are different for addresses in a warp. If the addresses are 64-bit, we can have more bytes with the same value and thus more power reduction. For data types that are smaller than 4 bytes (short integer and character types), our scheme can at least avoid the unnecessary access to the sign/zero extended bytes.

Lastly, as the current trend deploys a larger register file for GPUs and uses 8- and 10-transistor memory cells to tolerate every-increasing process variations [57], the energy consumption of register file is to further increase. This can justify adoption of a register value compression technique due to power and thermal constraints that do not scale well, which in turn can support G-Scalar practically at no cost.

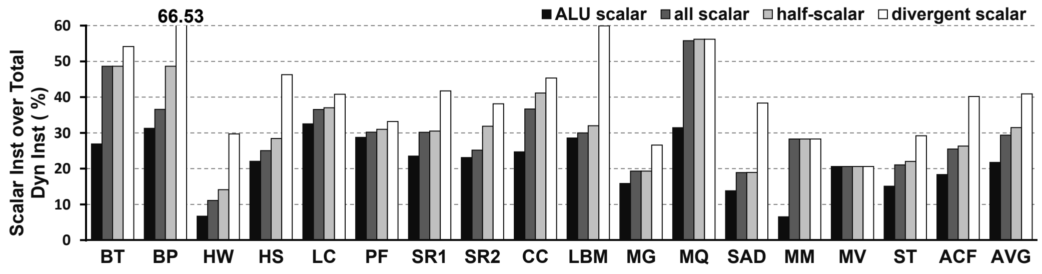


Figure 2.13: Percentage of instructions eligible for scalar execution.

2.4.3 Instructions Eligible for Scalar Execution

Figure 2.13 shows the percentage of instructions eligible for scalar execution. In Figure 2.13, "ALU scalar" is the baseline architecture, which only supports scalar execution of non-divergent arithmetic/logic instructions. The following category, "all scalar", covers special-function and memory (load/store) instructions eligible for scalar execution atop "ALU scalar". The next category, "half-scalar", includes instructions eligible for half-warp scalar execution atop "all scalar". Finally, "divergent scalar" includes divergent instructions eligible for scalar instructions atop "half-scalar". "ALU scalar" covers only 22% of total dynamic instructions on average. Atop "ALU scalar", we can cover 7%, 2%, and 9% more instructions when we cover scalar execution of special-function, memory, half-warp, and divergent instructions, respectively. Compared with "ALU scalar", G-Scalar can almost double the number of instructions eligible for scalar execution, increasing the number of scalar instructions to 40%.

Although special-function instructions contribute to only 3% of total dynamic instructions eligible for scalar execution, they consume 3$\sim$24$\times$ more energy than other floating-point instructions [3], and SFUs contribute up to 60% of the power consumption of GPUs in some benchmarks [58]. Therefore, supporting scalar execution for special-function instructions can considerably reduce GPU power consumption.

For a memory instruction eligible for scalar execution, all the threads in a warp attempt to access a value at the same address. That is, memory instructions eligible for scalar execution cannot improve the performance of the memory system, as the memory pipeline already has logic to coalesce memory requests from multiple threads in a warp. Nonetheless, scalar execution of memory instructions can still reduce power consumption of computing target

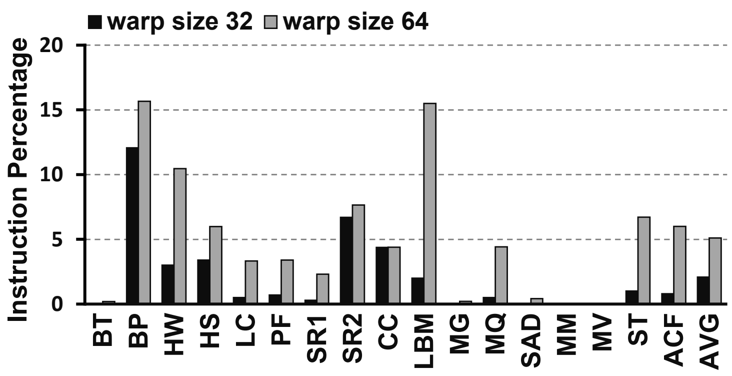


Figure 2.14: Percentage of instructions eligible for “half-scalar” execution for different warp sizes. For warp size of 64, we keep the same 16-thread checking granularity so it should actually be “quarter-scalar” instruction.

memory addresses.

The instructions eligible for half-warp scalar execution contribute to 12% of total executed instructions for BP. Although half-warp scalar execution is optional, it shows that the number of scalar instructions also depends on the granularity of checking vector registers storing scalar values. That is, if we increase the warp size to 64 and keep the 16-thread checking granularity, the average number of “half-scalar” (“quarter-scalar” in this case) will increase to 5% as shown in Figure 2.14. Note that 64-thread warps are supported by AMD GPUs, which execute 64-thread *wavefront* instructions using 16-lane SIMD pipelines in the Graphics Cores Next (GCN) architecture [21]. For some benchmarks, the number of instructions eligible for “half-scalar” execution increases significantly. One reason is that two scalar instructions with different operand values (from 32-thread warps) may be organized into one instruction from a 64-thread warp. Therefore, such instructions from those 64-thread warps are no longer scalar instructions but become “half-scalar” instructions. If the warp size further increases in the future, the half-scalar execution can be attractive to maintain the benefit of scalar execution.

The number of divergent scalar instructions depends on the total number of divergent instructions. Intuitively, benchmarks with many divergent instructions have more instructions eligible for divergent scalar execution. For example, HS, LBM and SAD have 17%, 30% and 19% divergent scalar instructions, respectively. Especially for LBM, supporting divergent scalar instructions can double the number of instructions eligible for scalar execution, compared with the previous scalar execution architectures.

2.4.4 Power Efficiency Improvement

Figure 2.15 shows the dynamic power efficiency, which is define as

$$\frac{\text{IPC}}{\text{Dynamic Power}}$$

In "ALU scalar", a scalar register file is also used to reduce dynamic power consumption of register file. When all of our proposed techniques are combined, we can improve the dynamic power efficiency by 24% on average. Compared with "ALU scalar", G-Scalar further improves the dynamic power efficiency by 15%. Among all the benchmarks, `BP` shows very high (79%) dynamic power efficiency improvement. `BP` is highly compute-intensive, and the total dynamic power of the GPU is over 100 W from GPUWattch [3]. Over 50% of the GPU dynamic power is consumed by execution units and register files. Especially, SFUs alone consume more than 25% of the total dynamic power although only 14% of total dynamic instructions are SFU instructions. With a very high percentage of special-function scalar instructions in `BP` ($\sim$60%), we can significantly reduce the dynamic power consumption (43%). The SFU dynamic power consumption is reduced to less than 10% of the baseline architecture. One of the reasons is that each thread of `BP` needs to compute 2.0 to the n^{th} in floating-point for n iterations. G-Scalar can execute such instructions as scalar instructions. However, some benchmarks show significant improvement in the number of scalar instructions but their efficiency improvement is not proportional. For example, more than 40% of total dynamic instructions are eligible for scalar execution in `LBM`, but the power efficiency improvement is less than 20%. The primary reason is that those benchmarks are rather memory intensive. A significant fraction of their dynamic power is consumed by memory subsystems, such as L2 cache, NoC and memory controller. We further show the overall GPU power efficiency in Figure 2.16 and normalized power consumption in Figure 2.17. GPUWattch models a fixed static power of 41.9 W [3] for GTX 480. With static power taken into consideration, the average power efficiency improvement dropped from 24% (dynamic power only) to 12%. Note that the target SM clock frequency (1.4 GHz) of the GTX 480 is very aggressive for the 40 nm TSMC process technology. As a comparison, the much more recent GTX 980 only runs at $\sim$1.2 GHz, implemented using 28 nm TSMC process technology;

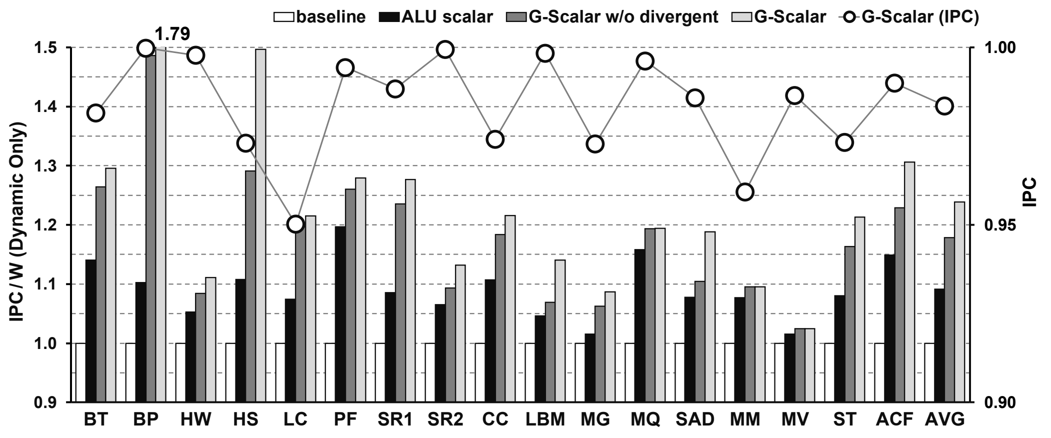


Figure 2.15: Normalized GPU dynamic power efficiency and performance. “ALU Scalar” and “G-Scalar w/o divergent” enable scalar execution on ALU pipeline and all three types of pipelines, respectively. “G-Scalar” extends scalar execution to half-scalar and divergent-scalar instructions. “G-Scalar (IPC)” shows the performance impact of adding three cycles of latency in normalized IPC.

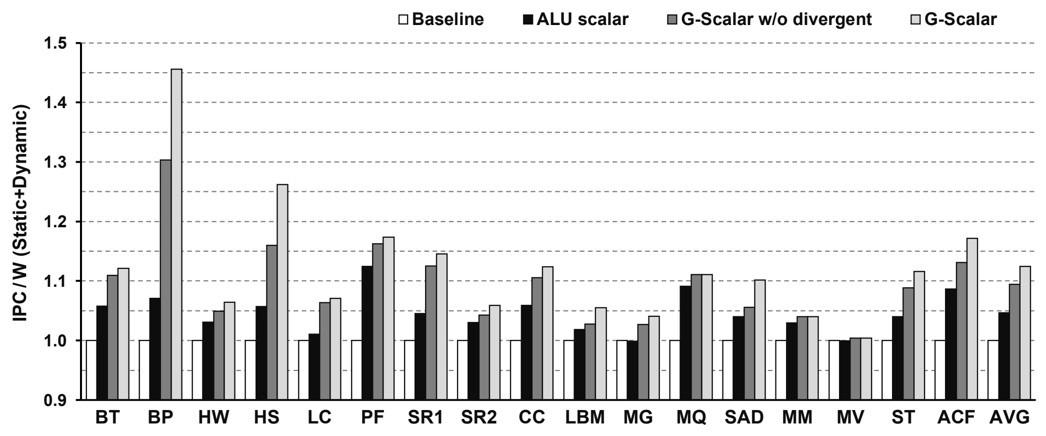


Figure 2.16: Normalized overall GPU power efficiency and performance. Both static power and dynamic power are included.

while GTX 1080 only runs at ∼1.7 GHz, implemented using 16 nm FinFet TSMC process technology [59]. Such an aggressive frequency target could be the main reason of such high ratio of static power of GTX 480.

2.4.5 Performance Impact

As we described in Sections 2.2 and 2.4.1, we increase the latency of pipelines by three cycles to account for cycles for reading encoding bits, decompressing

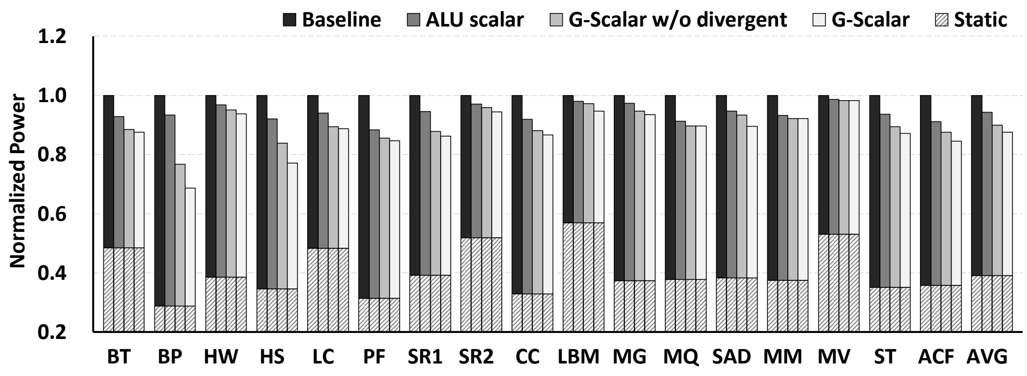


Figure 2.17: Normalized GPU power. Both static power and dynamic power are included.

and compressing a vector register. Figure 2.15 also shows the performance impact of G-Scalar after increasing the latency of baseline pipelines by three cycles. On average, the performance degradation is only 1.7%. The additional latency primarily increases the chance of pipeline stall due to data dependency. Although there is no data bypassing in GPUs, a large number of active warps can still effectively hide this latency [60]. Among all the evaluated benchmarks, `LC` shows the most significant performance degradation because it has an insufficient number of warps to effectively hide latency while utilizing many long latency instructions (e.g., integer DIV). These two factors together make `LC` more sensitive to the three-cycle latency increase, but still shows more than 20% IPC/Watt improvement.

2.5 Conclusion

As GPUs began to support more general-purpose applications, the fraction of divergent instructions in contemporary GPU applications has significantly increased. In this work, we demonstrate that (1) many divergent instructions are also eligible for scalar execution if we consider only operand values in active lanes in a divergent path and (2) special-function instructions, many of which are also eligible for scalar execution, consume a large fraction of execution power. However, prior scalar execution architectures cannot support scalar execution of these instructions. Tackling such limitations of prior scalar execution architectures, we propose G-Scalar, generalized scalar execution architecture along with a low-cost register value compression technique.

G-Scalar can support scalar execution of not only conventional non-divergent arithmetic/logic but also divergent and special-function instructions. Furthermore, if GPUs adopt our low-cost register value compression technique, G-Scalar is practically free, as it is architected to (1) share most of hardware resources with our register value compression technique and (2) reuse existing hardware resources of SIMT execution pipelines for scalar execution instead of implementing dedicated scalar execution pipelines. Our evaluation shows that G-Scalar, which consumes only 1% more chip space than the baseline GPU, can double the number of instructions eligible for scalar execution. This in turn improves power efficiency of GPUs by 24% and 15% compared with the baseline and previous scalar execution architectures, respectively. Lastly, our register value compression technique alone can reduce the power consumption of register file by 54%.

CHAPTER 3

EXPLOITING DATA SIMILARITY WITH APPROXIMATION ON GPU

3.1 Background

In Chapter 2, we showed that the G-Scalar scalar execution architecture can reduce redundant computations and thus energy consumption of SMID execution lanes, by executing only one thread and replicating its output value for all other deactivated threads at the end of execution. As GPU applications often comprise a significant fraction of code allowing approximation [12, 13], the scope of scalar execution can be expanded to instructions operating on vector registers storing similar input values [13]. This proposal simply ignores the last d bits of the operand values by masking the last d bits, and checks if the instruction with the masked values is eligible for scalar execution. However, it was noted that the primary benefit of such a technique still came from scalar execution [36].

Although the aforementioned approximate execution architecture was indeed demonstrated to be effective, we observe the following limitations. First, it requires a value similarity check for every instruction in a region of code that can be approximated, or an *approximable* region of code. This limits the complexity of the logic checking similarity of input values, resulting in limited opportunity for capturing approximable instructions more aggressively. Second, as it needs to expand the output value from one executed thread of *every* approximated instruction for all other skipped threads, it can only use a very simple technique to approximate the output value, i.e., simply replicating the output value of the executed thread for all the skipped threads. Lastly, it does not improve performance as it does not reduce the number of fetched, decoded, scheduled and executed instructions.

To further improve energy efficiency on GPU, we exploit the similar values produced and consumed by threads in warps. We propose Lock and Load

(LnL) architecture, which triggers approximate computing for an approximable region of code when load instructions return similar values for the code region that consums the values as inputs to start subsequent (dependent) computations. LnL checks the value similarity only for load instructions preceding approximable code regions, executes only a subset of threads in each warp, and approximates the output values of skipped threads only at the end of the code region executing store instructions. Therefore, the cost of checking the value similarity and approximating the output values of the skipped threads can be amortized by all the instructions in the code region. Specifically, we propose a low-cost hardware mechanisms to capture more approximable warps by approximating value similarity than the prior approximated execution architecture for GPUs. Furthermore, we propose to leverage an existing hardware mechanism in modern GPUs to more precisely approximate the output values of skipped threads than the prior approximated execution architecture for GPUs. This allows LnL to deploy more sophisticated and aggressive techniques to check the value similarity and approximate the output values, further improving energy efficiency.

Second, we enhance LnL to fuse repeated executions of an approximable instruction by multiple warps into a single execution by a warp, exploiting SIMD execution lanes unused by skipped threads of approximated instructions. In contrast to prior approximate execution architecture which simply disables the unused SIMD execution lanes, LnL can improve performance as it reduces the occurrence of repeatedly fetching, decoding and scheduling the same instruction for multiple warps.

We show that LnL can deliver 23% higher performance and 62% higher energy efficiency with negligible loss in accuracy of the final computed values. Furthermore, LnL costs less than 1% and 1% more space and power, as it uses novel arithmetic techniques and existing hardware resources for checking value similarity and interpolating the computed values. Note that prior GPU architecture enhanced to support approximated execution only reduces energy consumption, whereas LnL not only reduces energy consumption but also improves performance, potentially offering higher energy efficiency than the prior GPU architecture.

3.1.1 Value Similarity in Structured-Grid Applications

GPUs use SIMT execution model to improve energy efficiency. Threads in a warp execute the same instruction on their respective operands in a lock-step manner. If the operand values across the threads are similar, the computed values of the threads can also be similar. It has been reported that operand values for threads in a warp are often very similar when applications take input values with spatial correlation [5, 12, 13, 8, 40, 36]. This is especially true in an important class of applications, such as “structured-grid” applications.

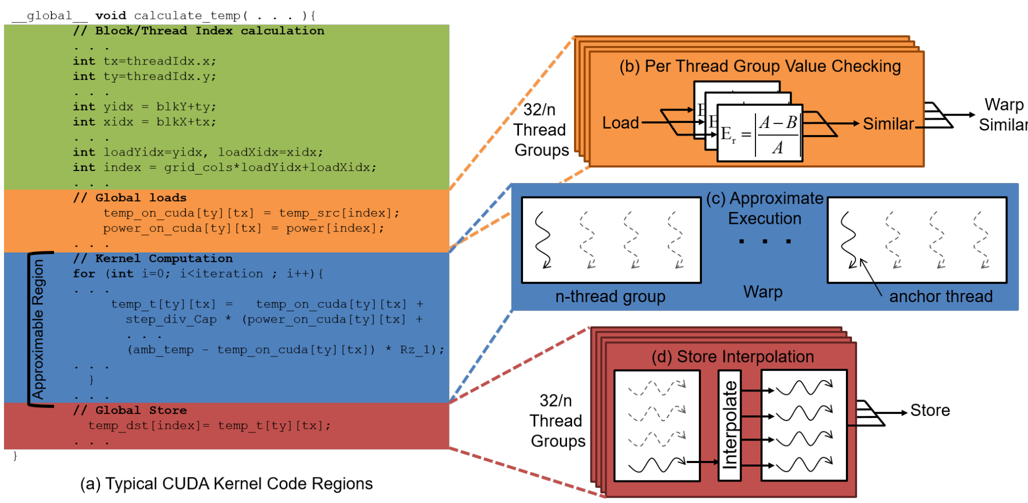
Generally, structured-grid applications compute grid cell values on a regular n-dimensional grid. The structured grid often comes from discretizing physical space with finite difference or volume methods [61]. Each output point of structured-grid applications is computed as a function of the value associated with the point and/or the values of neighboring points. As threads are mapped to each input value in a regular fashion, neighboring threads in a warp often use the values of neighboring points (e.g., temperature values of two adjacent grid points in `HotSpot` [54]). The spatial correlation and regular mapping of input values together result in that neighboring threads in a warp are very likely to have similar input values [13]. The examples of structured grid applications comprise physics simulation, which solves partial or ordinary differential equations with an iterative solver on dense multidimensional arrays or medical imaging.

3.1.2 Approximated Execution Architecture

The value similarity discussed in Section 3.1.1 gives us a chance to apply approximation techniques to GPUs. If two threads have similar values for their respective operands, we can leverage the output value of one thread to approximate that of the other thread. A naïve way to exploit such value similarity first checks the similarity of operand values across all threads for every instruction, through the addition of a comparison stage in the execution pipeline. If the input operand similarity satisfies a certain metric, a thread with representative operand values is executed and the remaining threads are not executed by clock-gating SIMT execution lanes associated with the skipped threads [13]. At the end of the execution stage, the output value of the executed thread is replicated for all the skipped threads and the replicated values are written back to the register file. As every instruction needs to be checked for similarity of input values (almost every cycle), the comparison stage can support only a simple method. For example, prior work checks whether or not operand values only differ in the d Least Significant Bits (LSBs) and are equal in the remaining (32-d) Most Significant Bits (MSBs) [13]. Such a simple method cannot capture similarity of values that are arithmetically close but micorachitecturally different as the LSBs are different. For instance, 127_{10} (= 01111111_2) and 128_{10} (= 10000000_2) are arithmetically close but microarchitecturally different as all the bits are different, and thus, is not categorized as having value similarity. Such a case is frequent for floating-point (FP) values, each of which is encoded with the exponent and mantissa fields. Furthermore, when the computed value of a single thread is replicated for all skipped threads, choosing approximable instructions should be very conservative to reduce the error. Lastly, such an approach does not improve performance as the approximated instruction in a warp still takes one issue slot. That is, it does not reduce the number of fetched, decoded, scheduled and executed instructions.

3.2 Load-Triggered Approximation

To better exploit the approximation opportunities discussed in Section 3.1.1, we propose Lock and Load (LnL), a load-triggered approximation technique, which is overviewed in Figure 3.1. LnL does not check the similarity of operand values for every instruction. Instead, LnL checks input values returned by load instructions and then decide whether or not to approximate a region of code. For a warp with loaded value similarity, LnL approximates the execution of instructions by executing a subset of threads (referred tLnLo as *anchor threads*) and skip execution of the remaining *non-anchor* threads. When storing the final computed values of the approximated warp at the end of the approximated region of code, LnL interpolates the final computed values of anchor threads for those of non-anchor threads. As such, LnL can support more aggressive techniques to check the similarity of values and expand the computed values of executed anchor threads for those of skipped non-anchor threads, compared to a naïve replication approach described in Section 3.1.2. In this section, we will first overview LnL. Subsequently, we



```
__global__ void calculate_temp( . . . ){
        // Block/Thread Index calculation
        . . .
        int tx=threadIdx.x;
        int ty=threadIdx.y;
        . . .
        int yidx = blkY+ty;
        int xidx = blkX+tx;
        . . .
        int loadYidx=yidx, loadXidx=xidx;
        int index = grid_cols*loadYidx+loadXidx;
        . . .
        // Global loads
            temp_on_cuda[ty][tx] = temp_src[index];
            power_on_cuda[ty][tx] = power[index];
        . . .
        // Kernel Computation
        for (int i=0; i<iteration ; i++){
        . . .
             temp_t[ty][tx] =   temp_on_cuda[ty][tx] +
               step_div_Cap * (power_on_cuda[ty][tx] +
               . . .
               (amb_temp - temp_on_cuda[ty][tx]) * Rz_1);
        . . .
          }
        . . .
        // Global Store
          temp_dst[index]= temp_t[ty][tx];
        . . .
}
```

Figure 3.1: Overview of LnL, a load-triggered warp approximation technique. (a) LnL identifies load-triggered approximable regions. (b) For global load operations, loaded values are checked for similarity. (c) Load operations that return similar values trigger approximate execution. (d) For store operations, results of approximated warps are expanded by interpolation.

will present architectural changes to support LnL in Section 3.3.

3.2.1 Load-Triggered Approximable Regions

Typically, a GPU kernel can be broken down into four code regions. An example is given in Figure 3.1(a), which shows a stripped-down version of the `HotSpot` kernel, one of representative structured-grid applications [54]. The first code region, colored green in Figure 3.1(a), calculates index values of memory locations based on thread block and thread IDs. The second code region, colored orange in Figure 3.1(a), loads input values from the global memory using the calculated index values. The global memory here is referring to the memory address spaces that are visible to all threads, including texture and constant memory spaces. The third code region, colored blue in Figure 3.1(a), performs computations using the loaded values from the memory. The fourth code region, colored red in Figure 3.1(a), stores the computed values back to the memory.

The code of the third region can be considered as a mathematical function f with a set of input values I and a set of output values. When f has the following “similar-in, similar-out” property: if $I_1 \approx I_2$, then $f(I_1) \approx f(I_2)$, we

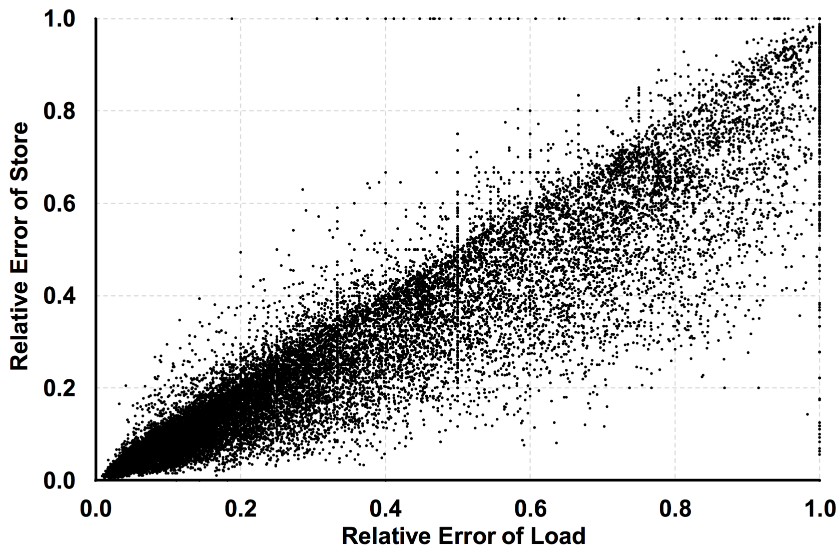


Figure 3.2: Relative errors of loaded values versus those of stored values in an approximable region of `Bilateral Filter` code.

consider that the third region is approximable and denote it an approximable code region. Figure 3.2 shows the relationship of the input value difference (or error) and the output value error of an approximable region of `Bilateral Filter` code. If the relative differences across values returned by a load instruction at the start of the approximable code region is small, then the relative differences across values stored by a store instruction is similarly small. Notably, the store error is typically smaller than the load error. We observe similar correlation behavior in structured-grid applications. As the approximable code region has the property mentioned above, we can just check the input values of the approximable code region (i.e., I from load instructions before the approximable code region) for each warp to decide whether or not we apply approximation to it.

3.2.2 Load-Triggered Value Checking

By only checking values returned by load instructions, instead of all input operands of every instruction, we drastically reduce the total number of checked instructions. Furthermore, the input values are usually copied to the memory space visible to users, such as global and constant memory spaces. Therefore, the value checking scheme does not need to be simple, such as bit wise comparison [13] which is often unable to capture similarity

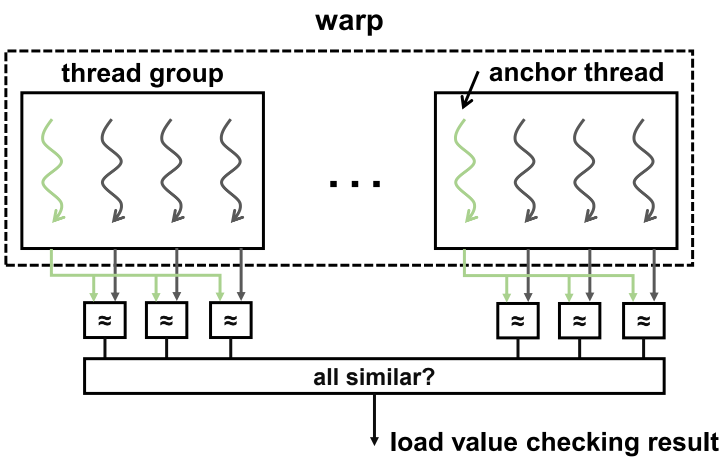


Figure 3.3: How the threads in a warp are grouped for load value checking. In this example, the size of each thread group is 4.

of values when the values are arithmetically similar but microarchitecturally very different as discussed in Section 3.1.2. Instead, we can afford a more sophisticated scheme calculating the relative difference to decide whether to approximate the warp or not.

Shown in Figure 3.3, for each approximable instruction, we first logically divide a warp into multiple *thread groups*. Each thread group has n consecutive threads (i.e., $32/n$ groups per warp) and the first active thread is the anchor thread. For each load, we check the loaded values within the thread group using certain criteria. For instance, in each thread group, we can calculate the difference (or error) of the loaded value of each thread as depicted in Figure 3.1(b), relative to that of the anchor thread, the first active thread in each n-thread group (solid arrows in Figure 3.1(c)). Supposing the loaded value of the anchor thread is `A` and that of the compared thread is `B`, we define the relative error E_r as:

$$E_r = \left|\frac{A - B}{A}\right|$$

Then we check whether or not E_r is smaller than a given threshold.

If all threads in a thread group meet such criteria, we consider that the loaded values in the thread group are similar. This checking result is only from a single load operation, which may be only one of the inputs of an approximate code region. If the approximate code region has more than one input (as is the case in Figure 3.1(a), the kernel has two inputs: `temp_on_cuda`

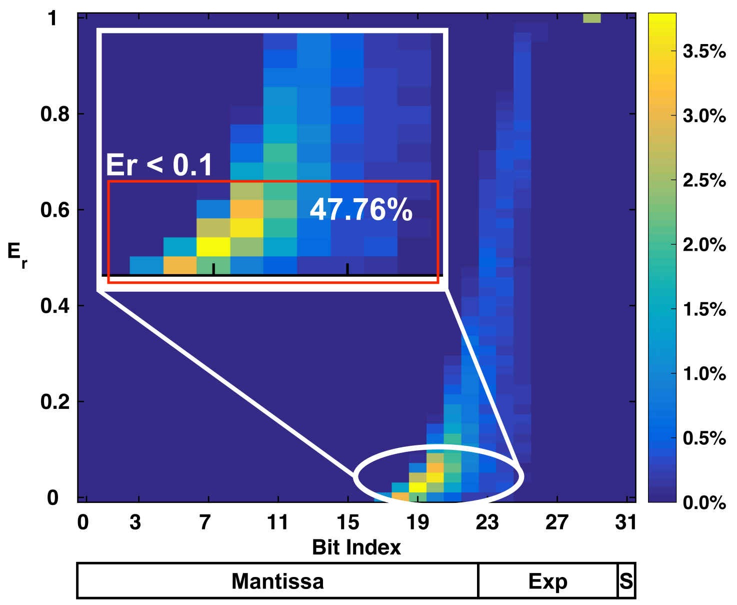


Figure 3.4: Percentage of arithmetically similar but microarchitecturally different loaded values in `Bilateral Filter`.

and `power_on_cuda`), all the inputs should be checked. That is, the warp is eligible for approximation only when every input has similar values. For example, `temp_on_cuda` has similar values, so does `power_on_cuda`.

Figure 3.4 shows the percentage of load instructions that are arithmetically close but microarchitecturally different across all 32 values of the warp, and the bits (from MSB) at which they start to differ. We find that nearly 50% of load instructions are arithmetically similar ($E_r < 0.1$) but microarchitecturally different, and they typically start to differ from the 8th – 12th bits from the MSB. This behavior is frequent for FP values. That is, prior work would not consider that each of these load instructions returns similar 32 values as it recognizes that only the first 7th – 11th bits from the MSB are the same. A low-cost implementation of logic calculating E_r for FP values is described in Section 3.3.2.

While we need to check similarity of every input, all the inputs are not loaded simultaneously. Instead, the inputs are loaded by multiple load instructions from memory. The loaded values may also be used in different approximable regions. Therefore, we need a mechanism to keep such information and history of the loaded values, which will be described in detail in Section 3.3.1. Based on the result of checking value similarity of a given input, each load instruction can be associated with a single bit to indicate whether the values loaded by a load instruction are similar or not. When a GPU encounters an approximable code region, which is directed by the programmer and marked by the compiler, it checks the history of the associated loaded values. If the history shows that all loaded values are similar, approximation is triggered for the current code region.

3.2.3 Approximated Execution

If a warp enters an approximate region of code and it is eligible for approximated execution, only the anchor threads of the warp are executed, but the rest of the non-anchor threads are not executed by clock-gating the SIMT execution lanes corresponding to these threads similar to prior work [13, 3].

For example, stores to shared memory must interpolate the value and write the value of all active lanes to memory, and related address calculation must always be executed, because the shared data may be used by the anchor thread later. Some instructions associated with critical control flow may need to be executed precisely. We use a compiler approach similar to prior work [13] to mark such instructions explicitly. Furthermore, to improve execution efficiency and thus performance, we propose LnL warp fusion that fuses repeated executions of an approximable instruction from multiple warps into an execution by a single warp, exploiting SIMT execution lanes unused by skipped threads of these instructions. The details are described in Section 3.4.

3.2.4 Approximating Output Values

At the end of an approximable code region (e.g., a store instruction), if the warp is approximated, we must approximate the values that are supposed to be computed by skipped threads in the warp. Since the results of threads at the end of an approximate code region are to be written into the destination registers and then the global memory using store instruction, we can approximate the results of skipped threads just before executing these store instructions. Note that some destination registers inside of approximated code regions can be used outside of the approximated code regions. A compiler can detect such cases and insert dummy interpolate instructions at the

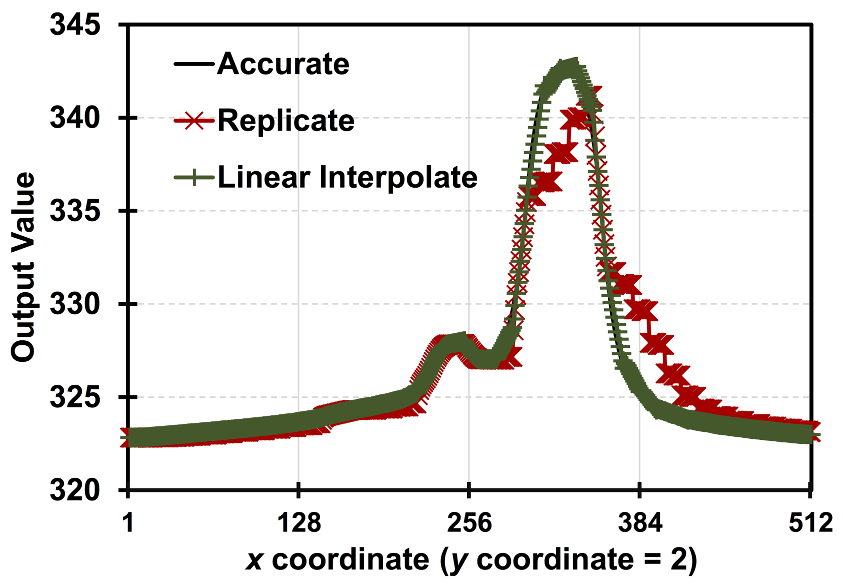


Figure 3.5: Accuracy comparison of two approximation techniques: replication versus simple linear interpolation of computed values from anchor threads.

end of the approximate code regions. Our analysis shows that such cases do not exist in the evaluated benchmarks.

The easiest way to approximate the output values of skipped threads is replicating the output value of the anchor thread in each thread group. However, we observe that such a simple approximation technique often leads to large approximation errors. Figure 3.5 compares the output values of `HotSpot` (temperature of a die) after we apply two approximation techniques: (1) a simple replication of the computed value of an anchor thread and (2) a linear interpolation of two anchor threads for the output values of skipped threads. We see that the replication technique can lead to large errors, especially when the output values rapidly change compared to neighboring values, whereas the linear interpolation technique renders negligible errors. This in turn allows LnL to use a more aggressive criterion when deciding whether or not an approximable code region should be approximated. We will discuss a low-cost way of supporting such a linear interpolation technique that leverages some existing features of modern GPUs in Section 3.3.3.

3.3 Architectural Support

In this section, we describe the necessary architecture changes for LnL described in Section 3.2. More specifically, we first describe the necessary Instruction Set Architecture (ISA) extension to annotate an approximable code region and a simple hardware structure to track whether approximate execution of the approximable code region can be triggered. Second, we elaborate a low-cost logic that can efficiently calculate the (approximated) E_r among FP values returned by each load instruction. Finally, we detail how we can estimate the results for the skipped threads, using the results computed by anchor threads practically with no extra hardware cost.

3.3.1 Annotating Load-Triggered Regions

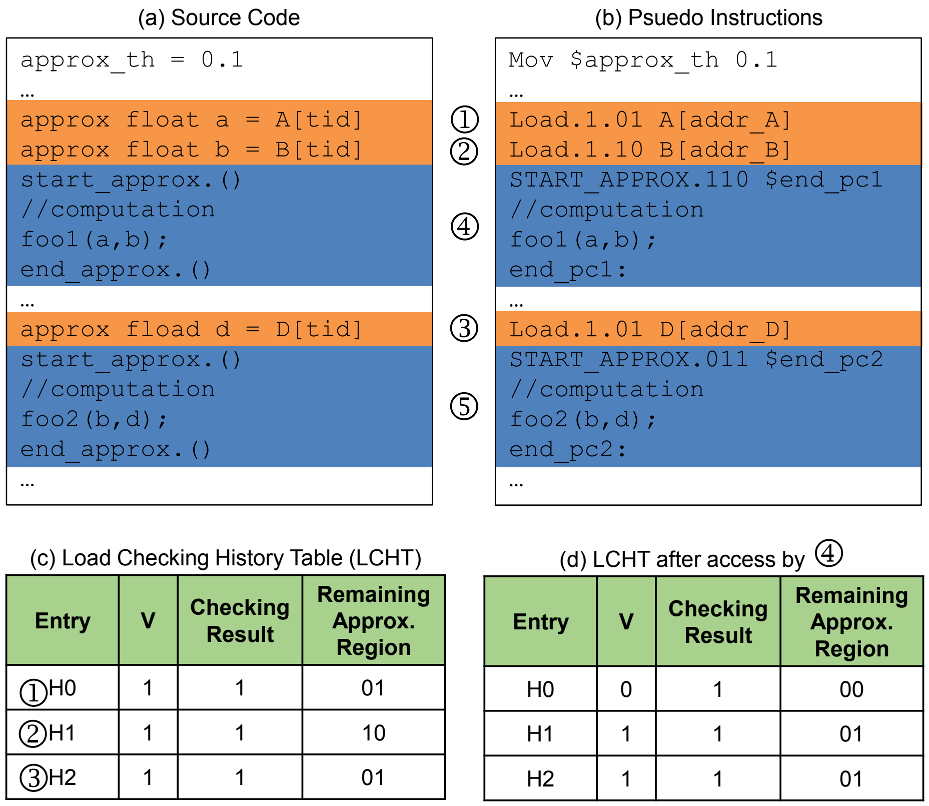
To facilitate load-triggered approximation overviewed in Section 3.1, we describe the ISA modification, hardware support and support for source code annotation of approximable code regions in this section.

Load instruction modification. In CUDA, we append the key word `approx` to indicate whether or not given loads should be checked for approximation. During compilation, this will be mapped to a LnL load instruction, which includes two additional fields: a 1-bit flag to indicate whether or not the returned values should be checked (`check` bit), and a few bits to represent the number of approximable regions that will use values from the load instruction (`approx_block_num` bits). These bits are set by the compiler, which analyzes the lifetime of the registers holding the loaded values [62]. With the proposed modification, a load instruction will look like:

```
LD.check.approx_block_num $dst, [$src+offset]
```

If the `check` bit is `0`, the load instruction will be executed as a normal load and `approx_block_num` bits are ignored. Otherwise, the loaded values are checked and `approx_block_num` indicates how many approximable code regions will use the loaded values.

In Figure 3.6(b) ①, ②, and ③ denote our new load instructions. In this sample code block, we have three global loads (for `A[]`, `B[]`, and `D[]`), with all three associated load instructions annotated to check for similarity of



```
approx_th = 0.1
...
approx float a = A[tid]
approx float b = B[tid]
start_approx.()
//computation
foo1(a,b);
end_approx.()
...
approx fload d = D[tid]
start_approx.()
//computation
foo2(b,d);
end_approx.()
...
```

```
Mov $approx_th 0.1
...
Load.1.01 A[addr_A]
Load.1.10 B[addr_B]
START_APPROX.110 $end_pc1
//computation
foo1(a,b);
end_pc1:
...
Load.1.01 D[addr_D]
START_APPROX.011 $end_pc2
//computation
foo2(b,d);
end_pc2:
...
```

(c) Load Checking History Table (LCHT)

Entry	V	Checking Result	Remaining Approx. Region
①H0	1	1	01
②H1	1	1	10
③H2	1	1	01

(d) LCHT after access by ④

Entry	V	Checking Result	Remaining Approx. Region
H0	0	1	00
H1	1	1	01
H2	1	1	01

Figure 3.6: This example shows the ISA and hardware support for annotating a load-triggered approximate code region. (a) In the source code, `approx_th` is the load checking threshold value. Key word `approx`, function call `start_approx()` and `end_approx()` are used to mark which load should be checked and where approximation should start/end. `foo1()` and `foo2()` represent the computation, not real function calls. (b) shows our new ISA to support annotation. (c) and (d) show the operations on the load checking history table (LCHT).

loaded values. Note that we allocate 2 bits for `approx_block_num` as most load values are only used by a few approximable regions, which will be shown in Section 3.5.3. If there are more than four approximable regions using the loaded values, the compiler can ignore some of the approximable regions based on some criteria such as potential of energy reduction, and treat such regions as unapproximable regions.

Hardware to track checked load instructions. If loaded value checking is enabled, the information of the value checking result and the number of approximable regions that use this load will be stored in a Load Checking History Table (LCHT) as shown in Figure 3.6(c). Each entry of the table has one valid bit `V`, one bit for checking result and two bits for the number of remaining dependent approximable regions. For example, in our figure, load instruction ① has one dependent approximate region, and load instruction ② has two dependent approximate regions. Furthermore, every load has been detected to return similar values and thus the checking result bit is set to 1. The checking result bit is set by the logic checking the similarity of loaded values detailed in Section 3.3.2. The information in the LCHT will be used to decide whether or not the block should be approximated.

Annotation for approximable code regions. We introduce annotation instructions to mark the starting and ending points of an approximable code region. In CUDA, the function calls `start_approx()` and `end_approx()` mark the start and end point of approximable regions, respectively. This CUDA instruction is translated to the following annotation instruction during compilation:

```
START_APPROX.loads_to_check $end_pc
```

The two fields in the instruction, `loads_to_check` and `$end_pc`, indicate which load-checking results should be used and where the current approximable region ends. The `loads_to_check` bits correspond to entries in the LCHT. For example, in Figure 3.6, our LCHT has three entries. Therefore, our `START_APPROX` instructions (④ and ⑤) both have three bits for `loads_to_check`, each corresponding to entries H0, H1, and H2, respectively. If all the associated loads indicate that the warp can be approximated, the warp will start approximated execution from the next instruction. Meanwhile, it stores the value of `$end_pc` in a special register of the warp. The warp will stop approximated execution when its program counter (PC) reaches `$end_pc`.

When a `START_APPROX` instruction accesses an entry in the LCHT, the number of remaining approximable code region that will use the loaded value will decrement by one. If the number becomes 0, the entry will be deleted from the LCHT. This is illustrated in Figure 3.6(d), which depicts the LCHT after an access by approximate region ④. This approximate region depends on load instruction ① and ②, and will check LCHT entries H0 and H1 (thus having the `loads_to_check` bits as `110`). Upon checking, the bits representing the number of remaining approximable code region will be decremented by one. Entry H0 will decrement to 0, and will be invalidated from the

LCHT. We assume that the LCHT entry management can be determined statically during compilation of the kernel. Both H1 and H2 now have one remaining dependent approximate region (⑤).

In our example, the first `START_APPROX` identifies that all dependent loads has checking result bits of 1, therefore, this code region can be approximated. This load-triggered approximate region will then approximately execute `foo1()` until it reaches `end_pc1`, the `end_pc` value of the approximate region.

Annotation for load checking threshold. In the source code, we need to first set the threshold of the relative differences among values returned by a load instruction, which will be stored in a special register `$approx_th`. The threshold value is not stored in FP format in the register. Instead, it is converted into a fixed-point format so we can do the comparison easily when checking the loaded values. The algorithm we use to convert the FP value is consistent with the scheme we use to check the loaded value described in Section 3.3.2. However, we assume the size of the thread groups must be specified when launching the kernels and cannot be changed during execution.

Backward compatibility of ISA extension. Extending the ISA for LnL is relatively straightforward for GPUs. In NVIDIA GPUs, the binary of CUDA kernels can contain multiple versions of the PTX (pseudo ISA of CUDA) code. Using the right version of the PTX code (with or without approximation enabled), the CUDA driver is capable of compiling PTX code to the SASS (native ISA) code at runtime if necessary [63]. Therefore, if we add new native/PTX instructions for a GPU, the compiler can store multiple versions of PTX code in the binary (with and without the new PTX instructions) and is able to generate new machine code using the appropriate version of PTX code. In this way, we can easily keep the backward compatibility and take advantage of the new instructions.

3.3.2 Checking Loaded Value Similarity

To check similarity of values returned by a load instruction, we implement the checking logic in each load/store unit. When the 32×32-bit values are returned from memory, LnL uses an anchor-thread mask (derived from the active-thread mask) to choose the loaded values of the anchor threads (Figure 3.1(c)). In each thread group, LnL then compare the 32-bit value of an anchor thread of a given load instruction with that of a non-anchor thread of the load instruction by a comparison unit to determine whether the two values are similar or not. As this comparison needs to be done for each of all non-anchor threads in each thread group and across thread groups, LnL needs 31 comparison units to handle the largest thread group (one anchor thread compared with 31 other threads) in a load/store unit. Nonetheless, LnL still implements 32 comparison units to have one comparison unit per thread. This implementation allows a one-to-one mapping between the thread and comparison unit, which simplifies the interconnection and selection/routing logic.

To calculate the relative difference between two FP values (i.e., E_r), LnL needs FP subtraction and division units, which are too expensive even if LnL only checks values of load instructions in an approximable region. Thus, we propose a low-cost comparison unit that can approximate E_r. Figure 3.7 depicts the detail of a low-cost LnL comparison unit. To limit approximation errors, LnL limits loaded value similarity to values which have the same sign bit and only differ in the exponent field by 1. If the sign bits of two FP values differ or the exponent field differs by at least 2, which is checked by the logic illustrated in the left side of Figure 3.7, LnL immediately terminates the comparison, as E_r is larger than 100% or 50%, respectively.

If two compared FP values have the same sign bit and the difference of exponent values differ by at most 1, we can easily convert the FP values to fixed-point (FxP) ones to simplify the logic for a comparison unit. For FP-to-FxP conversion, LnL first puts the implicit “1” bit back into the mantissa and use the mantissa as the fractional part. For de-normalization, LnL only needs to shift the mantissa value with smaller exponent by 1 bit. As LnL only needs to right-shift the mantissa values by 1 bit at most, the logic for this step is very simple. Then the sifted mantissa value becomes a FxP value to approximate E_r. LnL simply drops the sign exponent bits of the FP values, as it ends up comparing two values with the same sign. Finally, LnL calculates E_r of two FxP values with integer subtraction and division. As precise integer division is expensive, LnL also approximates it by approximating the reciprocal of a given divisor with three-interval piecewise linear approximation. Then LnL can replace division with multiplication.

The detailed steps of approximating the reciprocal of a given value are as

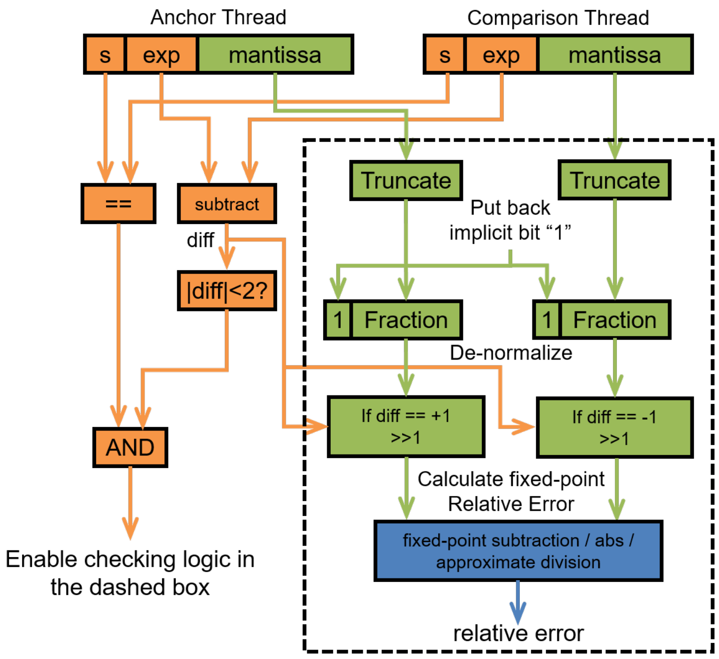


Figure 3.7: Low-cost comparison logic to approximate E_r.

follows. In our FP-to-FxP conversion scheme, at least one of two FxP values starts with 1 for its MSB as both are treated as unsigned numbers, and LnL takes such a value as a divisor. Because the impact of divisor bits on the precision of a reciprocal value exponentially decreases from the MSB to LSB, LnL only checks the first eight bits of the divisor, i.e., one integer bit (= 1) and seven fractional bits. The seven fractional bits determine one of three interval of 128 possible values: ①, ②, and ③ in Figure 3.8. Depending on the fractional bits (note that the value of the the divisor has very limited range), the reciprocal of a divisor can be approximated by:

$$\text{①}: \frac{1}{\texttt{DIV}} = \texttt{8'hFF - \{DIV[6:0],1'b0\},}\quad \texttt{DIV[6:0]} \leq \texttt{7'b0010000}$$
$$\text{②}: \frac{1}{\texttt{DIV}} = \texttt{8'hF6 - \{1'b0, DIV[6:0]\},}\quad \texttt{DIV[6:0]} \leq \texttt{7'b1100000}$$
$$\text{③}: \frac{1}{\texttt{DIV}} = \texttt{8'hDF - \{2'b0, DIV[6:1]\},}\quad \texttt{DIV[6:0]} > \texttt{7'b1100000}$$

That is, a division is replaced by subtracting a divisor from a constant where each constant is determined to minimize errors in approximating the reciprocal for a given interval. Figure 3.8 shows that the error for any one of the 128 possible divisor values is less than 3.4%. Note that LnL needs only

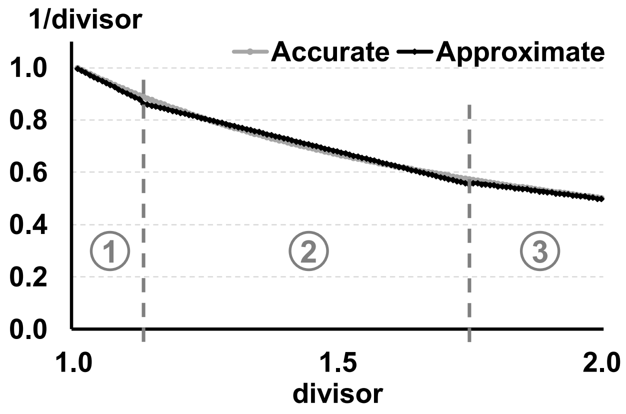


Figure 3.8: Algorithm and accuracy of the approximate reciprocal unit.

small 8-bit subtraction and multiplication units as its takes only 8 bits of a mantissa value.

Each comparison unit generates one bit after comparing two values. If all the comparison units generate 1's, LnL determines that the load instruction for this warp returned similar values. We use an all-one detector for all thread groups to check if the warp is approximable or not. We assume the threshold value is stored in a special register when launching the kernel, so we do not need to explicitly specify the threshold value in the load instruction.

The comparison units should be placed between the write-back port of load/store unit and the register file, and they can be easily pipelined to avoid affecting the throughput or clock frequency. We show the synthesis result in Section 3.5.2. In Fermi, a single-precision FP subtraction and division takes 16 cycles and 1038 cycles, respectively [64], whereas the comparison unit only takes three cycles to get the relative error as shown in Section 4.5.

3.3.3 Approximating Output Values

When the approximated execution is completed, the warp needs to store the output values in memory. The easiest method to approximate the output values is to replicate the output values of the anchor threads. However, we showed that such a simple method can lead to large errors (Section 3.2.4). Therefore, we propose to linearly interpolate the output values for the skipped threads between two anchor threads based on their thread ID. We desire relatively precise interpolation, as numerical imprecision introduced by interpolation will directly affect the output values. Fortunately, we can utilize the existing hardware on GPUs to avoid expensive FP arithmetic units for this purpose.

Almost all modern GPUs comprise dedicated hardware providing texture filtering capability for graphics application [65]. For 1-D textures, the filtering is basically a 1-D linear interpolation on the FP pixel values. For example, when we want to access a pixel whose coordinate `x` does not directly correspond to a pixel in the texture, the texture filtering hardware performs the following operation [63]:

```
tex[x] = (i+1-x)*T[i]+(x-i)T[i+1]
```

where `T[i]` and `T[i+1]` are the pixels enclosing the sample point `x`, i.e., integer `i` satisfies: `i<x<i+1`. To approximate the output values, LnL can simply treat a store of a warp as accessing a 1-D texture, with the value of an anchor thread as a pixel value `T[n]` and the thread ID of the threads as the 1-D coordinates `x`. Note that the thread IDs need to be mapped to the coordinate to perform the interpolation using texture filtering unit. However, this mapping only depends on the thread group size and therefore, can be done at kernel launch time. Since the texture unit is already connect to the SMs, we expect minimal hardware modification to accommodate the interpolation.

3.4 Load-Triggered Warp Fusion

3.4.1 Warp Fusion Overview

When a warp is approximated, SIMT lanes corresponding to non-anchor threads are unused. However, the GPU front-end still treats approximated warps as normal warps. That is, an instruction of an approximated warp has at most only half of the threads active, but it still takes one issue slot. Since there are at most half of the threads active in an approximated warp, we have enough execution units to execute two or more approximated warps at the same time. Therefore, we propose to enhance LnL to fuse repeated executions of an approximable instruction by multiple warps into a single execution by a single fused warp to improve the front-end efficiency and

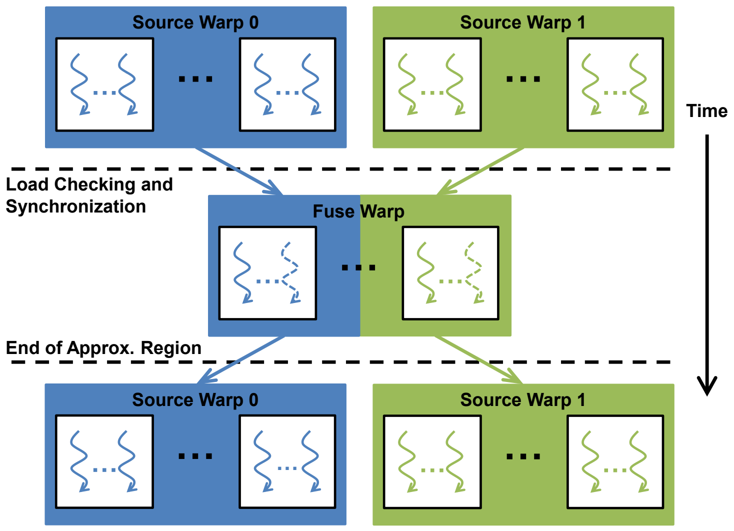


Figure 3.9: An example for fusing two warps. Arrows with dashed lines represent inactive non-anchor threads.

performance.

In an SM, if two consecutive active warps, in terms of warp IDs, are both eligible for approximation, we can fuse these two warps (referred to as source warps), into one warp (referred to as fused warp). Two, four, or eight warps can be fused in a fixed way with simple hardware support, but we only consider fusing two consecutive warps for simplicity. In Figure 3.9, we show an example of warp fusion. LnL first checks if both source warps are approximable or not at each synchronization point. If both source warps are approximable, LnL fuses them as a single fused warp for the whole approximable region. This is possible because LnL determines whether or not a warp is approximable before starting the execution of the approximated region. Finally, LnL splits the fused warp back into two source warps when leaving the approximable region.

To start warp fusion, we need to know if both source warps are approximate or not at the approximable region entry point. Therefore, we use our `START_APPROX` instruction as the barrier synchronization point. If both source warps are approximable, they will be marked as a fused warp. For a fused warp, the front-end fetches, decodes, and issues instructions for the source warps only once. When an instruction from such a fused warp is executed, it operates on the operands from all active lanes of the two source warps and executes them as if they are from a single warp. Hence, all the per-warp hardware structures such as register file (RF) and SIMT stack (a hardware stack for managing divergent control flow) are kept unchanged. In this way, we effectively fuse two source warps into one fused warp.

During the execution of the fused warps, if there is control divergence, the push/pop operation on both SIMT stacks will be based on all threads from both source warps. For instance, if one source warp can re-converge while the other source warp cannot, the re-converged source warp cannot pop the corresponding entry from its SIMT stack, until the other source warp also re-converges. We have to do this because a fuse warp must keep both source warps synchronized during the entire fused execution phase. When two non-divergent warps are fused at the beginning but one warp diverges later, the fused warp is considered as divergent and the non-divergent half will also go thorough the other path.

When the approximable region ends, the fused warp will be split into individual source warps again and continue execution normally. To split a fused warp, we only need to reset the bits that mark the source warps as fused, then continue fetching, decoding and executing the instructions of source warps normally. We can simply do this because we maintained all the architectural states of both source warps when executing them as a fused warp, such as Program Counters (PCs), SIMT stacks and the scoreboards.

3.4.2 Hardware Support

To support warp fusion, some hardware structures in the baseline architecture need to be adapted as shown in Figure 3.10, including fetch, instruction buffer (I-Buffer) and operand collector. All adaptations are simple and introduce negligible costs.

Fetch. In the baseline architecture, the fetch stage uses a table for the PCs and warp IDs of the active warps. In each cycle, the fetch scheduler selects an eligible warp to fetch instructions. Enabling warp fusion requires two additional fields in each entry of the PC table. Along with the warp ID and the PC, we need a fused bit, `f`, to indicate if the warp is currently fused or not, and a pointer, `p`, to point to the table entry corresponding to the other source warp of the fused warp pair. Since the PC table only needs to have entries for active warps, it only needs 48 entries for Fermi [38] and one

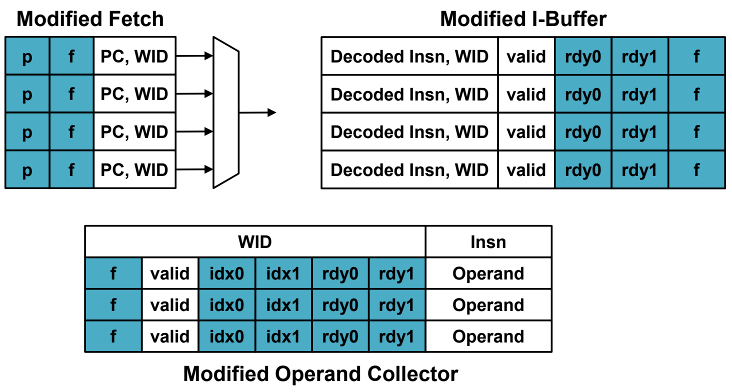


Figure 3.10: Hardware modified for warp fusion. Colored fields are modified fields. The symbols `f`, `p`, `rdy0/1` and `idx0/1` denote the fused bit, pointer to the fused warp, ready bit and index bits, respectively.

pointer only requires six bits.

For all the fused warps, the fetch stage can fetch only one of the source warps to eliminate the need for fetching instructions for fused warps repeatedly. For simplicity we assume it always fetches for the source warp with an even warp ID. However, the PCs of both source warps are updated normally to make sure both warps can continue the execution immediately after the fused warps are split.

I-Buffer. I-Buffer is used to buffer decoded instructions of each active warp. In the baseline architecture, each warp has two entries in the I-Buffer. Each entry holds the decoded instruction, valid/ready bits and warp ID. The ready bit is set by the scoreboard logic when all dependencies of the instruction are resolved, and the ready bits must be set before the instruction can be issued. In the warp fusion architecture, a fused bit `f` and a second ready bit are added to each entry of I-Buffer. The two ready bits, `rdy0` and `rdy1`, are for the two source warps of the fused warp. If a warp is not a fused warp, only the first ready bit is checked.

If a warp is a fused one, the warp can be issued only when both ready bits are set. The fused warps will be executing the same instructions so the decoded instructions of the fused warps will be the same. Therefore, we only need to store one copy of the decoded instructions in I-Buffer. However, as we mentioned above, we still need to calculate the index of the physical registers for both source warps. Because we only fuse consecutive source warps, the warp IDs of the two source warp are n and $n + 1$. Considering the way the physical ID is calculated:

$$\texttt{physical ID} = \texttt{warp ID} \times \texttt{\# of regs per warp} + \texttt{logical ID}$$

we only need to get the physical ID, `idx0`, for the registers of one source warp (the one with smaller warp ID) using the logic already in baseline architecture. Then the physical ID of the other source warp, `idx1`, can be obtained by adding the value of *# of regs per warp* to `idx0`.

Operand Collector (OC). The operand collector also requires small modification to efficiently support warp fusion. Like I-Buffer, we only need to add a fused bit `f`, a second ready bit and another index bit field in the operand collectors. For fused warp, it only needs one modified OC to buffer its operands, instead one OC for each source warp. To achieve this goal without doubling the size of the data field, we need to pack the operand data before writing them to OC. A simple way to pack the data is grouping threads into pairs by thread index: `(0, 1), (2, 3), ..., (30, 31)`. Note that the threads in a thread pair are always in the same thread group, so at most one of them will be active. Using 16 of 2-to-1 multiplexers, we can efficiently pack the data to half the size of the data field of OC and each source warp uses half of the data field of the OC. Since one bit is needed for each thread pair to indicate which one thread is chosen, we need to add 32 more bits for each 1024-bit data field. Similar to I-Buffer, when the warp is not fused, only the first ready bit is used. However, both ready bits for a fused warp must be set before the warp can be issued.

3.5 Evaluation

3.5.1 Methodology

We use GPGPU-Sim 3.2.2 [55] with GPUWattch [3] to evaluate the effectiveness of LnL. Both GPGPU-Sim and GPUWattch are configured to model the Fermi architecture of NVIDIA GTX 480 [38]. The key configuration parameters are listed in Table 3.1. We use benchmarks from the Rodinia benchmark suite [54] and CUDA SDK 4 [66]. As we explained in Section 3.2, the proposed approximation scheme is for applications that use a structured grid

Table 3.1: GPGPU-Sim configuration.

# of SMs	15	SM Frequency	1.4 GHz
Warps per SM	48	Schedulers per SM	2
Warp Size	32	Reg File per SM	128 KB
CTAs per SM	8	$L1 per SM	16 KB
Mem Channels	6	$L2 Size	768 KB

Table 3.2: List of benchmarks. Phy. and Img. denote physics simulation and image processing, respectively.

Benchmark	**Domain and Grid Size**	**Abbr.**
`HotSpot`	Phy., 512×512	`HS`
`Bilateral Filter`	Img., 1024×1024	`BF`
`Convolution Separable`	Img., 1024×1024	`CS`
`Convolution Texture`	Img., 1024×1024	`CT`
`Sobel Filter`	Img., 1024×1024	`SF`

and have the "similar in, similar out" property, including physics simulation and image processing. The benchmarks we evaluated are listed in Table 3.2.

In our evaluation, we use thread group size n of 4, 8 and 16 for all benchmarks. On the other hand, the threshold values for the benchmarks are selected individually such that the average output errors of the benchmarks are within 10%. For `BF`, `CS`, `CT` and `HS` we check the relative errors, and for `SF` we check the absolute errors.

3.5.2 Area and Power Overheads

The load checking logic shown in Figure 3.7 is described with Verilog HDL and then synthesized using a low-power TSMC 40 nm library. For each SM, we only need one set of checking logic units since each SM has only one load/store unit. One set of checking logic consists of 32 checking units assuming warp size is 32. The delay, area and power of a comparison unit is 1.2 ns, 586 μm^2 and 0.3 mW, respectively. Since there are 15 SMs in the modeled GPU, a total of 480 comparison units are added. This leads to the total area of 0.3 mm^2 and the total power of 144 mW. Although the baseline GPU is implemented with a 45 nm technology, the checking logic

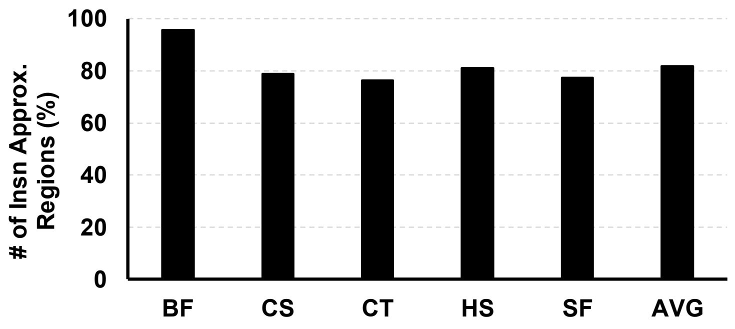


Figure 3.11: Percentage of dynamic instructions covered by approximable regions. AVG denotes the average.

can still fit into three pipeline stages even after the scaling. Thus, we apply three extra cycles to each load/store unit performing the value checking for load operations. We also modeled the memory structures of the original and modified PC tables in the fetch stage, I-Buffers and operand collectors using CACTI 6.5 [67]. In summary, both the total area and power costs are less than 1% compared to the baseline GPU.

3.5.3 Load Statistics and Approximable Region Coverage

Figure 3.11 shows the percentage of the dynamic instructions covered by the approximable regions among the total number of dynamic instructions. This is the upper bound of the actual number of approximated instructions. From Figure 3.11 we can see that the approximable regions of the benchmarks can cover a significant fraction of the dynamic instructions and provide good opportunities to apply approximation. The maximal number of loads to check for one approximable region is only three, and one loaded value is used by only one approximable region among all the benchmarks. Note that we do not need to check all the loaded values. Instead, we only need to check the loads that are strongly correlated to the output based on the algorithm or profiling.

However, having a wide coverage of the approximable region is only the necessary condition to gain high energy efficiency. In Figure 3.12, we show the number of actually approximated instructions. From Figure 3.12 we can see that the percentage of approximated instructions is up to 79%, 71%, 64%, 68% and 59% for `BF`, `CS`, `CT`, `HS` and `SF`, respectively. The number of approximated instructions also shows a clear trend with respect to threshold

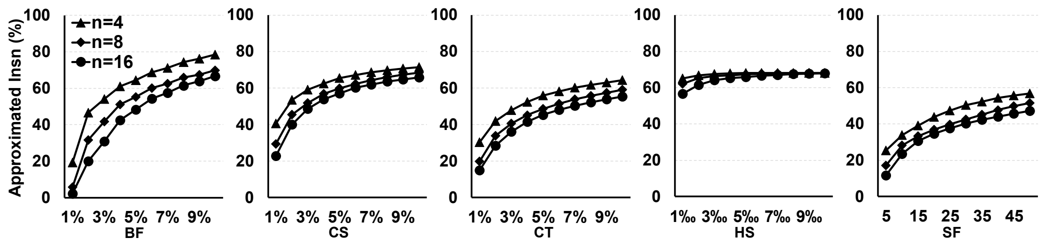


Figure 3.12: Percentage of approximated instructions. Thread group size n = 4, 8, 16 for all benchmarks. Horizontal axis is the threshold values. Note that `SF` uses absolute difference, not relative error E_r, for threshold values.

values and the thread group size. When the threshold value increases, the number of approximated instructions increases; when the thread group size decreases, the number of approximated instructions increases. Such a trend is very intuitive. When the threshold value gets larger, more instructions will be eligible for approximation; and when thread group decreases, there are fewer threads to compare in the thread group and it is more likely that the differences among the threads are less than the given threshold. Note that though `HS` does follow the trend, the number of approximated instructions quickly saturates when the threshold value increases. Because the loaded values of `HS` are either extremely similar or extremely different. A small threshold value can capture most of the cases having similar values.

3.5.4 Output Error

Figure 3.13 shows the error of the output. We use the Root-Mean-Square Error (RMSE) relative to the geometric mean of the output to characterize the error introduced by applying approximation using LnL. The highest output error is only 8% for all the benchmarks and configurations. For `CT` and `SF`, the error increases almost linearly with the threshold values, while the output error of other benchmarks increases much slower and seems to be bounded. However, it does not mean we can arbitrarily increase the threshold value for the benchmarks with sub-linear error growth. When the threshold value gets larger than a critical value, the error will quickly increase and the quality becomes unacceptable. Such behavior is similar to the jump in output error of `HS` with a thread group size of $n = 16$.

From the benchmark results we can conclude that small thread group size

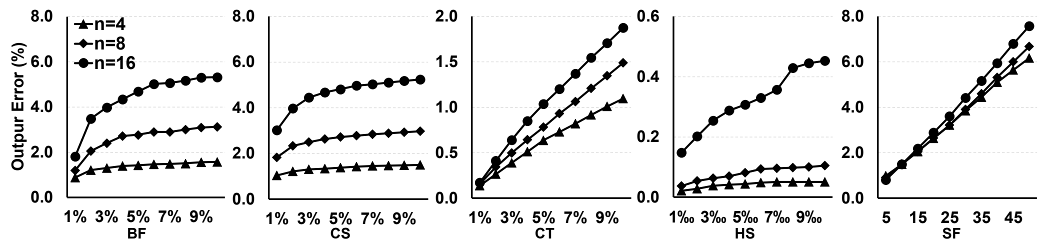


Figure 3.13: Output error: $\frac{\text{RMSE}}{\text{Mean Value}}$.

is almost always beneficial for reducing output error. The reason is that when the thread group size is small, fewer threads will be disabled when a warp instruction is approximated. With more accuracy output points, the estimation or interpolation of the results gets easier and introduces fewer errors. However, since we have more anchor threads when using a small thread group, the energy efficiency can be negatively affected as we can see in Section 3.5.5.

3.5.5 Energy Efficiency Improvement

Figure 3.14 shows the energy efficiency of LnL using IPC/W normalized to baseline. The energy efficiency of LnL is up to 2.2× of the baseline. On average, LnL improves the energy efficiency and performance by 62% and 23%, respectively, when using the best configurations. For `CS`, `CT` and `HS`, the best configuration for energy efficiency uses the largest threshold value and the largest thread group size, while for `BF` and `SF`, the best configuration uses the largest threshold value and the smallest thread group size. Note that, the performance of `BF`, `CT` and `SF` actually exceeds the throughput limit of the front-end thanks to warp fusion. However, the performance improvement is very small considering the high percentage of approximated instructions. The major reason is that, due to the memory access granularity of GPUs, the disabled threads cannot reduce the memory bandwidth requirement. An approximated instruction is similar to a divergent or irregular memory access in terms of the utilization. Because the memory bandwidth bottleneck still exists, the performance improvement is limited to 23%.

Different from the clear trend of the number of approximated instructions and output error, a small thread group size is not always the best configuration for energy efficiency. As we mentioned in Section 3.5.3, though smaller

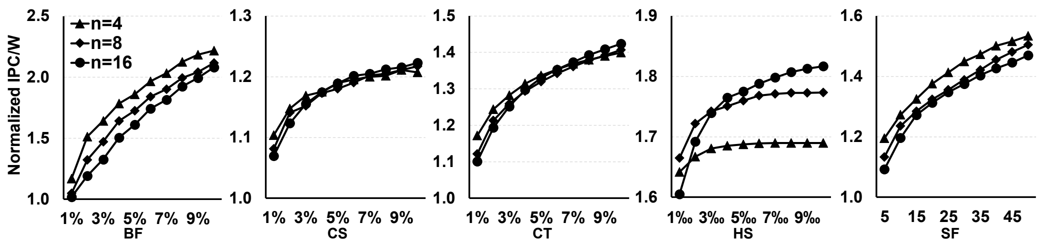


Figure 3.14: Energy efficiency normalized to baseline.

thread group size can potentially increase the number of instructions eligible for approximation, there is less benefit from applying approximation on each instruction. When the thread group size is small, there will be more thread groups in a warp, meaning more anchor threads and fewer disabled threads. Therefore, whether or not a small thread group size is good depends on which of the two factors is dominate. For instance, the number of approximated instructions of `HS` quickly saturates when the threshold value gets larger. With the number of approximated instructions being similar, the energy efficiency is proportional to the size of the thread group: the larger the thread group is, the higher the energy efficiency is. The crossing lines in Figure 3.12 are the consequence of both the number of approximated instructions and energy efficiency improvement per approximated instruction.

3.6 Conclusion

Prior approximated execution architecture [13] requires GPUs to check the value similarity for every instruction. Furthermore, it does not improve performance like LnL because it does not reduce the total number of fetched, decoded, scheduled and executed instructions. In this work, we first proposed Lock and Load (LnL) where approximate computing is triggered by similarity of values returned by load instructions in a warp and then performed for an approximable code region followed by the load instructions. This not only reduces the overhead of checking the eligibility of approximation for every instruction but also allows us to deploy more sophisticated techniques for checking the eligibility of approximation and only approximating the output values for all the skipped threads at the end. Second, we enhance LnL to fuse consecutive approximable warps when both are executing approximable code and improve the performance because it reduces the number of fetched, decoded, scheduled and executed instructions. Our experiment shows that LnL can improve energy efficiency and performance by 62% and 23%, respectively with only 1% power/space cost.

CHAPTER 4

AXMEMO: GENERAL-PURPOSE APPROXIMATE MEMOIZATION

4.1 Background

So far we have only focused on reducing redundant computation on GPUs. In this chapter, we switch our focus to CPUs, since CPU is still the one of the most widely used types of processor. In terms of exploiting data similarity and computation redundancy, the biggest difference between GPU and CPU is the architecture and execution model. (Note that while most modern CPUs support SIMD instructions, we focus on the none-SIMD instructions. Since the SIMD pipelines in CPU can easily utilized the techniques proposed in previous Chapters with minor or no modification.)

The difference in the architecture makes the distribution of energy spent on instructions drastically different. In comparison to GPUs, modern CPUs spend rather a large fraction of their time and energy on fetching, decoding and scheduling operations rather than executing them. Even for a double-precision fused multiply-add instruction, the energy spent on actual computation (execution unit) can be as low as 3% of the total energy of the instruction [4]. Another key difference between GPU and CPU is about their execution model. The difference is similar to the difference between "spatial" and "temporal" locality. In GPU, multiple data and computation instances exist in the SIMD pipeline at the same time in the different SIMD lanes, thus we exploit this "spatial" locality to find redundant computation. However, we do not have such nice "spatial" locality among the computation instances on CPU. Instead, the computation is done one after one, hence "temporal" locality. To exploit computation redundancy, we have to somehow record previous computation inputs and results on CPU for later reuse.

One of the effective techniques can be memoization that replaces operations with lookup operations to a previously recorded results table. This technique exploits the computation redundancy in the code. During the execution of an application, computation blocks may take the exact same inputs and generate the same outputs as previous instances of the computation. Such redundancy is caused by either repetitive input patterns or the nature of the algorithm, which is prevalent in the cyber-physical domain due to the processing of natural inputs. Exact memoization has been explored at a fine-grained instructions level [31, 32], a more coarse-grained function level [27, 35] and even a task level [29]. Except for the last work, the rest of these inspiring techniques do not exploit approximation. The technique in [29] is a pure software technique and does not explore the benefits of hardware support for hashing or reuse of the spare cache space for memoization. This work fills the void in providing the hardware support for approximate memoization of relatively large blocks of code. To replace large code blocks with memoization (lookups) in various applications, we need to handle a variety of input types and counts. The varied and potentially large number of inputs raise several challenges for the memoization process.

First, the proposed memoization hash needs to generate unique lookup indices without imposing significant delay or energy. The hashing mechanism should also support varying number of inputs. Second, the memoization process needs to maintain a high hit rate for the lookup operations. As more inputs are added, the probability of having an exact match decreases. To address these challenges, we first develop a novel use of Cyclic Redundancy Checking (CRC) to hash the varying number of inputs. To improve memoization hit rate, AxMemo employs a two-level lookup table, which utilizes the spare storage in the second level caches. These solutions enable AxMemo to efficiently memoize relatively large code regions with variable input sizes and types using the same hardware mechanisms. Experimentation with ten different benchmarks from various domains show that AxMemo provides 2.64× average speedup and 2.58× average energy reduction. These benefits come at the cost of 0.2% of average quality loss and 2.1% area overhead. These results show that AxMemo is an effective approximate computing solution.

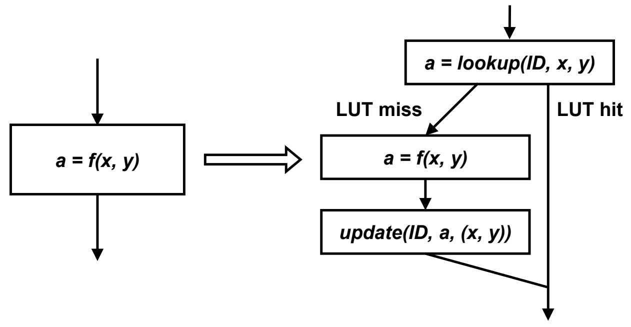


Figure 4.1: Control flow transformation of AxMemo. ID is the lookup table (LUT) ID.

4.1.1 AxMemo Overview

Before describing the memoization scheme of AxMemo, it is worth mentioning that AxMemo can be used only to memoize computation blocks equivalent to code sections that: (1) always produce the same results or return values given the same inputs, and (2) do not have any observable side effects, such as writing to a file. Computation blocks not satisfying either of these properties are ineligible for memoization.

Given a valid computation block, we transform the block into a branch structure. As shown in Figure 4.1, we first look up the LUT with inputs of the original computation block. If more than one computation block is memoized, we need to have multiple LUTs. To distinguish multiple LUTs, each LUT has a LUT ID. If there is a LUT hit, we copy the output of the LUT to the registers, skip the original computation and continue execution. Otherwise, we execute the original code and update the LUT, then continue program execution.

The transformation itself is simple and straightforward. However, as we discussed in Section 4.1, one of the challenges is how to perform the lookup efficiently. The different numbers/sizes of inputs for different applications make simply concatenating the inputs and using it as a lookup tag infeasible, especially when the computation block has many inputs. For instance, one of the target computation blocks in our benchmarks (`Sobel`) needs nine floating-point numbers as input. Concatenating them means we need a tag of 36 bytes for each LUT entry. It not only demands to significantly increase the LUT capacity but also causes timing overhead since we need to have a comparator wider than 200 bits for a tag comparison. The worse part is that applications with a few inputs waste such resources. Therefore, a small and fixed-size tag is the key to efficient lookup and memoization.

On the other hand, to maintain high a hit rate for many inputs, we combine memoization with approximation. Approximation exploits the fault-tolerant characteristics of some applications. By relaxing the requirement of producing precise results and allowing an error bound in the output, applications in the field of image processing and computer vision for example, can achieve notable performance and/or energy efficiency benefits. Therefore, we allow similar inputs to be a match to increase hit rate in AxMemo.

4.2 AxMemo Memoization Unit

Most of the memoization operations are performed by the memoization unit. In AxMemo, the memoization units are private to each CPU core. Figure 4.2 shows the overview of the memoization unit: the memoization unit mainly consists of a Hashing Unit, Hash Value Registers and a LUT. To efficiently perform the lookup with a fixed-size tag, we hash the inputs of the original computation block, dubbed the memoization inputs, and use the hash value for the LUT tag. The hash values are first stored in Hash Value Registers. When a lookup or update request is received, the hash value is read from the Hash Value Registers using LUT ID (`LUT_ID`) and thread ID (`TID`) as an address to index an entry of a LUT. Combined with the `LUT_ID`, the hash value will be used to perform the lookup/update operations in the LUT. The LUT in the memoization unit is considered as L1 LUT, and we can also have an *optional* inclusive L2 LUT to improve the hit rate.

4.2.1 Hash Key Generation and Approximation

Hash key generation. We propose to use the cyclic redundancy check (CRC) algorithm [68] for the hashing. CRC is a widely used error detection algorithm. An n-bit CRC algorithm can take an input of arbitrary size and generate an n-bit CRC value. The following properties of CRC that make it suitable for the memoization scheme:

1. It does not need to have all the input data to start hashing. It takes a

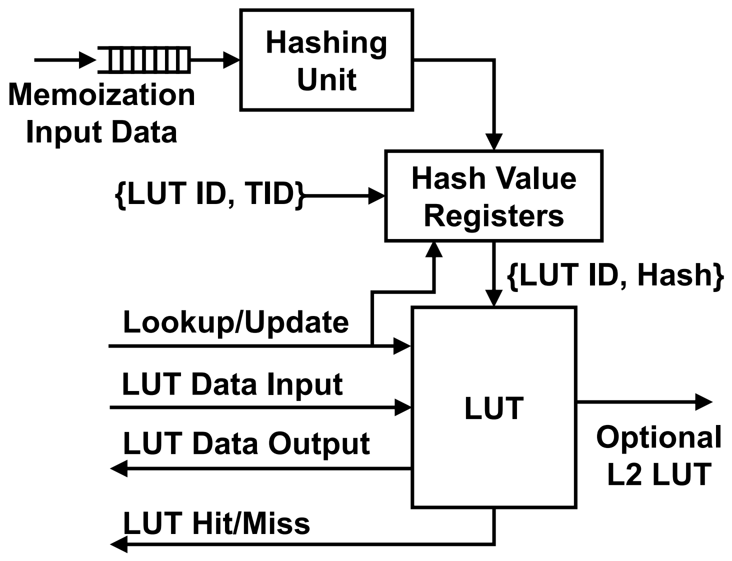


Figure 4.2: Overview of the memoization unit. Major components include hashing unit, hash value registers and LUT. TID denotes thread ID. Memoization inputs are the inputs of the computation block being replaced, such as x, y in Figure 4.1.

stream of inputs and "accumulates" them into the output. This is an important feature because it can help with hiding the latency of hash value calculation.

2. Every bit of the inputs will affect the CRC output, not like the sampling-based algorithm such as the one used in [29].

3. A hardware implementation of CRC is cheap.

4. The CRC can work in many sizes: 16-bit CRC, 32-bit CRC, 64-bit CRC etc. We can use different CRC sizes for different designs.

As shown in Figure 4.3, a simple implementation of CRC is similar to linear-feedback shift register (LFSR). Compared with LFSR, however, CRC uses the XOR of the input bit and the feedback bit as the input to the first register, instead of using the feedback bit directly. A serial CRC implementation processes 1 bit of input every clock cycle. An alternative n-bit parallel implementation can process n bits per clock cycle, which reduce the latency to $1/n$ of the serial version. The n-bit parallel implementation of m-bit CRC needs a $2^n \times m$-bit RAM. This small RAM stores constants that are required for the n-bit parallel implementation. Note that any hashing scheme incurs collision, i.e., two sets of inputs with different values generate the same hash value, but it is acceptable for approximable applications as long as it is not frequent.

Approximation for memoization. Even without considering hash collision, using a CRC value as a LUT tag still requires all the memoization inputs to be an exact match to have a LUT hit. As we discussed in Section 4.1, we exploit the correlation between the similarity of the inputs and the similarity of the outputs and apply approximation to the inputs in AxMemo. The approximation can potentially increase the LUT hit rate, thus the performance of the application.

We use a simple approach to apply the approximation, truncating some least significant bits (LSBs) of the memoization inputs before sending them to the hashing unit. The level of approximation, i.e. the number of truncated bits, is programmer controllable for each variable. Mathematically, the truncation rounds the input down by a given relative precision (for floating-point variables) or absolute precision (for integer variables). Though we only evaluated truncation, a more sophisticated approach can be applied since the approximation does not affect hashing unit.

In our experiment, we use the compiler tools to profile the applications with a sample input set. Then we use the statistics to determine the number of bits to be truncated for each memoization input. Alternatively, we can use a dynamic approach. A certain percentage of the execution time can be

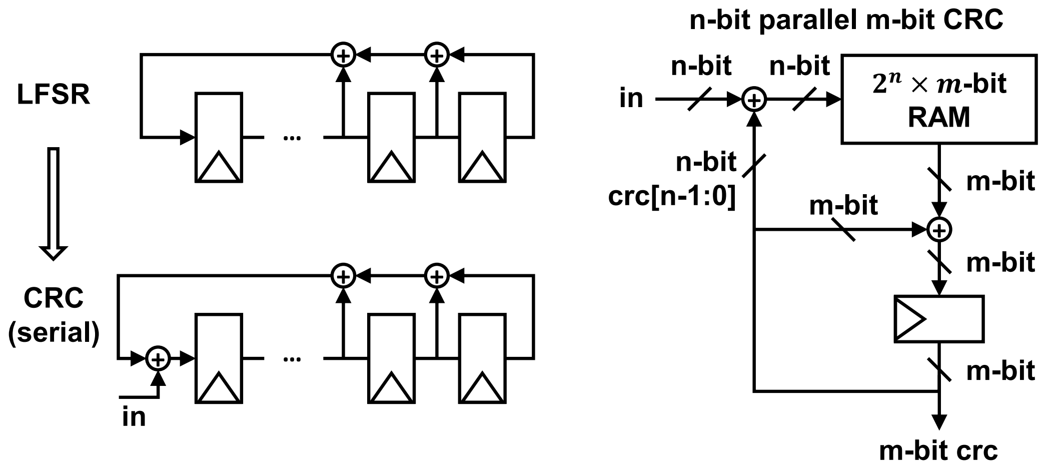


Figure 4.3: Implementations of cyclic redundancy checking (CRC) unit. The linear-feedback shift register (LFSR) unit and serial CRC unit are not showing actual configurations.

allocated for profiling at runtime periodically. During the profiling phase, the memoization unit always returns miss to the processor even if there is a hit so we can use the computation results and the LUT output to calculate error and adjust the approximation level accordingly during the execution. See Section 4.4 for more detail.

4.2.2 Hash Value Registers

The Hash Value Registers (HVRs) are used to store the hash values and they are not just buffers for temporarily storing the CRC values that are ready. When the processor sends the memoization inputs of different LUTs to the memoization unit in an interleaved way, the HVRs store the intermediate results of CRC and serve as the hardware context of the CRC calculation. The HVRs are addressed by the LUT ID and the thread ID {`LUT_ID, TID`} as shown in Figure 4.4.

Note that thread ID is the hardware thread ID used to identify the threads in processors support simultaneous multi-threading (SMT). The number of needed CRC registers depends on the maximum number of allowed LUTs and maximum number of SMT threads. Supposing an architecture with up to eight LUTs in one thread and two SMT threads, the CRC registers should contain at least 16 $\times$ 32-bit registers for 32-bit CRC. For out-of-order processors, {`LUT_ID, TID`} is equivalent to the architectural name of the Hash Value Register. To support the instruction-level parallelism, more "physical" Hash Value Registers are needed and they should also be "renamed".

4.2.3 Lookup Table (LUT) Structure

The LUT has an organization similar to a normal set-associative cache, as depicted in Figure 4.4. The LUT is composed of LUT entries organized in sets, which are equivalent to sets of cache lines. Each LUT entry has one LUT tag field and one LUT data field, or simply LUT tag and LUT data. Compared to a cache line, the LUT tag field is equivalent to the address and the LUT data field is equivalent to the data. The LUT data is 4-byte by default, and we can configure it to 8-byte by combining two LUT entries. The 8-byte LUT data is necessary to support 8-byte data types and can also

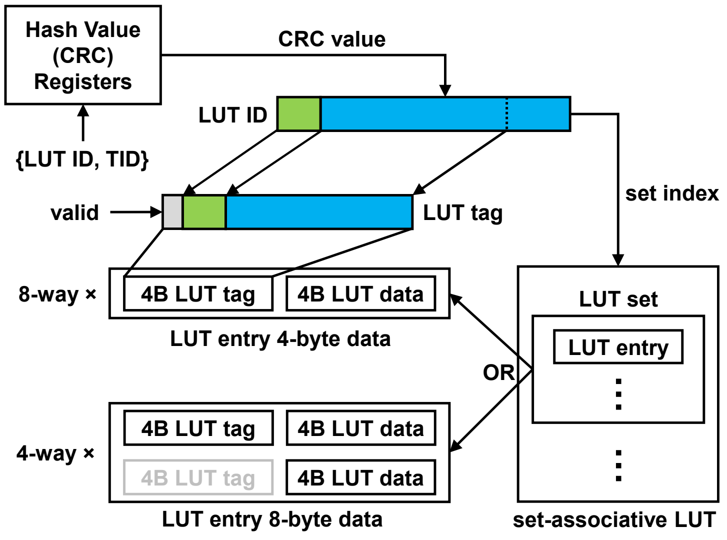


Figure 4.4: LUT organization similar to a set-associative cache. Each set can be configured to either one 4-way or one 8-way set. The CRC value is combined with LUT ID and used for LUT tag.

be useful in the cases that the LUT logically has many outputs. In such cases, we can pack as many outputs into the 8-byte LUT data field as possible to reduce number of LUT accesses.

Note that we may implement L1 LUT and *optional* L2 LUT differently. We use a dedicated SRAM array for L1 LUT while allocating part of the last-level cache for L2 LUT. L1 LUT is limited to small sizes ($\leq$16 KB) and L2 LUT can use up to half of the last-level cache. Implementing a small separate L1 LUT can avoid interference with valuable L1 cache space and timing. On the other hand, using part of the last-level cache for L2 LUT can avoid a large overhead to implement a large LUT in the baseline processor.

In AxMemo, one LUT set can be configured as either 8-way 4-byte LUT tags with 4-byte LUT data, or 4-way 4-byte LUT tags with 8-byte LUT data. In the latter case, the half of the LUT tags are not used. Since some lower bits of the CRC value `CRC` are used to index a set, we do not need to store the whole 32-bit `CRC` value in the LUT tag array. The upper bits of the LUT tag array are used for a valid bit and `LUT_ID` bits. Such configurations ensure there is enough space for the valid bit and a 3-bit `LUT_ID`. With `LUT_ID` included in the LUT tag, we can store multiple logical LUTs in one unified LUT.

We chose the LUT entry size and associativity because such configuration allows one set of the LUT entries to just fit into a 64-byte last-level cache line, when the LUT tag and LUT data are both considered as data for the cache. This allows us to use the last-level cache as L2 LUT most efficiently. For simplicity, we assign a fixed number of ways in the last-level cache to the L2 LUT in our experiment.

4.2.4 LUT Lookup and Update Operation

When a CRC value `CRC` is ready, the memoization unit can start a LUT lookup or update upon a request from the CPU. The CPU sends lookup or update requests along with the `LUT_ID` and its `TID` to the memoization unit.

LUT lookup operation. When receiving a lookup request, the memoization unit first checks if there is any pending CRC calculation for this LUT. Since our implementation only allows the lookup request to be sent after the last memoization input is sent to the memoization unit, we only need to check if there is data from this thread for the LUT waiting in the small input queue of the memoization unit. If there is any pending calculation, the memoization unit stalls the request until the calculation is completed. Then the lookup is performed in the LUT to find a LUT entry whose LUT tag matches with {`LUT_ID, CRC`}. A condition code, which is used for branch instructions, is also set based on lookup result, i.e. hit or miss. The condition code is used later by the program to determine whether or not the computation should be skipped.

Upon a LUT hit, the memoization unit returns the LUT data to the CPU register specified by the lookup request and the computation is skipped. Upon a LUT miss, the program executes the original computation and the memoization unit immediately starts to allocate an LUT entry for the update request and will update the LUT once the computation is done. When no invalid entry is available, the L1 LUT entries are invalidated (without L2 LUT) or evicted to L2 LUT (with L2 LUT) using the least recently used (LRU) policy. The same allocation policy is used for L2 LUT. Different from data in normal cache, the LUT entry will not be eventually written back to main memory. The L2 LUT entry will always be invalidated when the it needs to be evicted.

LUT update operation. Once the original computation is done, the CPU will send an update request to the corresponding LUT entry. An update is very similar to a lookup: it first accesses the CRC register to get `CRC`, then it writes {`LUT_ID, CRC`} and the input data to the corresponding LUT entry and sets the valid bit. As mentioned earlier, the allocation of the LUT entry happens in parallel with the original computation, most update can perform the write immediately.

For multi-core processors, there is no coherence required for the LUTs, because the same LUT tag should always have the same LUT data without hash collision, which makes coherence unnecessary. If the same LUT tag has different LUT data in different LUT arrays, it means a collision occurred. In such a case, it is meaningless to force the LUTs to stay coherent since we cannot tell which data value is more precise.

4.3 Instruction Set Architecture Design

To enable memoization, we need to extend the ARM-v8a ISA with the following five instructions. All of them can be encoded into 32-bit instructions:

1. `ld_crc dst, [addr], LUT_ID, n`: This instruction loads memory content at `addr` to register `dst`. It also sends the loaded data (with the last `n` bits truncated) and the value of `LUT_ID` to the memoization unit. The AxMemo compiler replaces the normal load with this instruction for the variables that are marked as input to the memoization region.

2. `reg_crc src, LUT_ID, n`: This instruction reads a register `src`, truncates the last `n` bits and sends the truncated value to the LUT. The destination LUT is identified by the value of `LUT_ID`. In some benchmarks, such as `FFT`, all the inputs to the memoization region are not load instructions. As such, we include this instruction in AxMemo ISA to support these scenarios.

3. `lookup dst, LUT_ID`: This instruction performs the LUT lookup in the memoization units. The target LUT is identified by the value of `LUT_ID`. It also sets the condition code for branch instructions based on the lookup result. If the access to the memoization unit is a hit, `lookup` writes the returned data to the destination register `dst`. We use `lookup` together with a normal branch instruction to skip the calculation, when

the access to the memoization unit—with LUT identified by `LUT_ID` is a hit.

4. `update src, LUT_ID`: This instruction sends the value of register `src` to memoization unit and insert it to the LUT. The LUT entry is allocated after a lookup miss. It also sends `LUT_ID` to the memoization unit as the identifier for LUT.

5. `invalidate LUT_ID`: This instruction invalidates all the entries of the LUT, which is identified by `LUT_ID`. This instruction is only used at the end of the program execution, or when the program needs to reuse the LUT associated with `LUT_ID` for other logical LUT. `invalidate` is called infrequently ($\approx$1% of total number of dynamic instructions) during the execution of the application and only consumed a few processor cycles because we use dedicated hardware for invalidating all LUT entries (Section 4.1).

For `ld_crc` and `reg_crc`, the programmer may enable approximation by specifying a non-zero value for `n`. The value of `n` determines how many LSBs should be truncated. The programmer may disable approximation (truncating) by specifying a value of zero for `n`. All the instructions, which access the memoization unit, sends the target LUT (specified by instructions with `LUT_ID`) to the memoization unit as identifiers of the memoization request. To guarantee that all the input data are sent to the CRC unit in the same program order, the `lookup` must only be issued after all the input data are sent to the target CRC unit and stored in the LUT. To support this case, we impose a dependency, which is equivalent to first reads a dummy register and then write into the dummy register (i.e. it is both the source register and the destination register), on the `ld_crc`, `reg_crc`, and `lookup` instructions. The imposed dependency by this consecutive write and read makes these instructions to follow the exact same program order as defined in the program.

4.4 Compiler Support for AxMemo

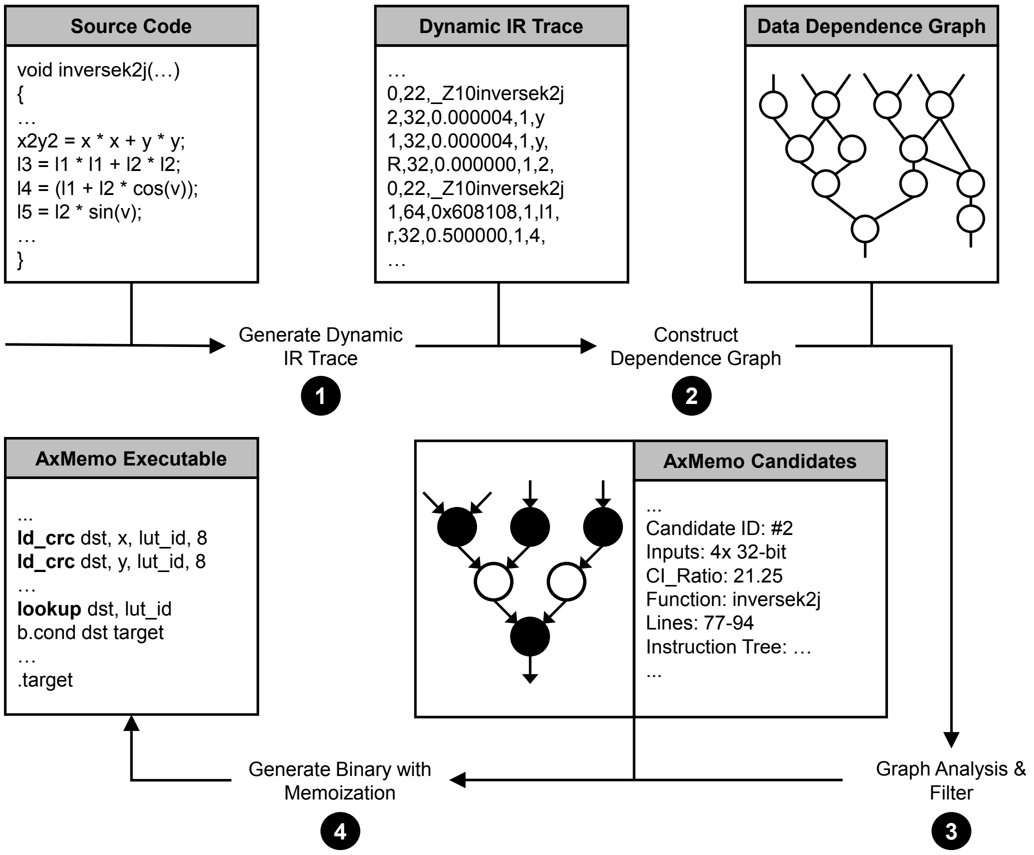
Compiler-guided code analysis. To facilitate the use of AxMemo for generic programs, compilers and dynamic analysis tools are needed to identify suitable computation blocks as candidates for memoization. This task requires examining the program's dataflow characteristics and input patterns in a search for sections of code that are memoizable and yield promising speedup. Two key factors determine the effectiveness of AxMemo for such sections: (1) execution time spent on them must be substantial enough to result in notable performance gains if memoized; (2) they must have few, approximable inputs to yield a justifiable hit rate and error bound. Both factors can be evaluated if we construct a dynamic data dependence graph (DDDG) of the program detailed at the instruction level with all intermediate input data recorded. With this motivation, we devise a compilation and analysis workflow shown in Figure 4.5 to identify and memoize candidate computation blocks for AxMemo. ❶ We use LLVM-Tracer [69] to generate a dynamic LLVM intermediate representation (IR) trace of the program by executing it on a sample input set. ❷ We construct a DDDG from this trace using ALADDIN [70] with some modifications.

A DDDG $G = (V, E)$ is a directed acyclic graph whose vertices represent LLVM IR pseudo-instructions and edges represent data dependencies between vertices. For example, an edge $v \rightarrow w \in E$ indicates that the output of instruction v is used as an input operand to instruction w. In addition, each vertex of the DDDG is weighted by its estimated latency. Note that LLVM IR allows an arbitrary number of registers hence the DDDG captures true dependencies of the program. An AxMemo transformable candidate subgraph $S = (V_s, E_s)$ is a subgraph of the DDDG G that can be memoized with AxMemo without interfering original program flow. S has a set of input vertices $V_i \subset V_s$ and a set of output vertices $V_o \subset V_s$ satisfying:

1. If $v \rightarrow w \in E$ where $v \in V \backslash V_s$ and $w \in V_s$, then $w \in V_i$
2. If $v \rightarrow w \in E$ where $v \in V_s$ and $w \in V \backslash V_s$, then $v \in V_o$

In other words, S reflects a program block at the instruction granularity whose inputs are operands of vertices in V_i and outputs are results of vertices in V_o. Figure 4.6 shows an example subgraph of the DDDG of the benchmark `Blackscholes`.

Recall that the potential speedup of an AxMemo transformed block depends on the execution time and input constraint factors detailed earlier. Thus, to capture the desirability of a candidate subgraph S, we define its compute-to-input ratio (CI_Ratio) in Equation 4.1:



```
void inversek2j(…)
{
…
x2y2 = x * x + y * y;
l3 = l1 * l1 + l2 * l2;
l4 = (l1 + l2 * cos(v));
l5 = l2 * sin(v);
…
}
```

```
…
0,22,_Z10inversek2j
2,32,0.000004,1,y
1,32,0.000004,1,y,
R,32,0.000000,1,2,
0,22,_Z10inversek2j
1,64,0x608108,1,l1,
r,32,0.500000,1,4,
…
```

```
...
ld_crc dst, x, lut_id, 8
ld_crc dst, y, lut_id, 8
…
lookup dst, lut_id
b.cond dst target
…
.target
```

```
...
Candidate ID: #2
Inputs: 4x 32-bit
CI_Ratio: 21.25
Function: inversek2j
Lines: 77-94
Instruction Tree: …
...
```

Figure 4.5: Compilation and analysis flow to identify optimal code sections for AxMemo.

$$I\_Ratio = \frac{\sum_{v \in V_s} v \times \text{weight}}{\#\text{ of Inputs}} \tag{4.1}$$

The higher this ratio, the more execution cycles we can replace with a lookup and/or the higher hit rate we can expect. In step ❸, our analysis task simplifies to finding candidate subgraphs S in the DDDG G with a high CI_Ratio but not exceeding the number of inputs allowed by AxMemo. Our algorithm consists of running a directed breadth first search rooted at each vertex of the transpose of G. For each vertex $v \in V$, we find the AxMemo transformable subgraph S with v as the sole output vertex (i.e. $V_o = \{v\}$) having the highest CI_Ratio and add it to our candidate list if that ratio exceeds our predefined threshold. To aid this search, programmers may specify specific functions for analysis rather than the entire program as sections involving system calls or I/O operations are non-memoizable. After the search process, we often identify more than 10^4 candidates depending

```
double normalize (
     double in,
     double min,
     double max,
     double min_n
     double max_n
)
{
   return (((in - min) / (max - min)) *
         (max_n - min_n)) + min_n;
}
```

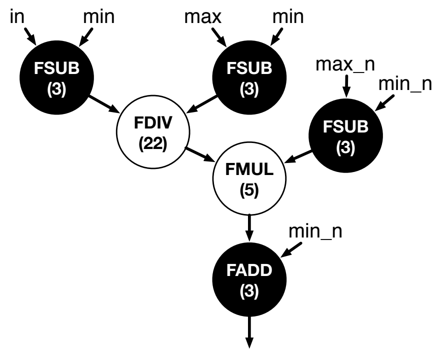


Figure 4.6: Example subgraph from the dynamic data dependence graph of `Blackscholes`. Black vertices are input/output vertices. The number in parenthesis indicates the weight (latency) of each vertex.

on the application and its input size. This is no surprise as our graph is constructed from a dynamic trace, thus many subgraphs will have identical structure if they belong to a loop body or repeated function call. Therefore, in the last step of ❸ we filter out candidate subgraphs if they are structurally equivalent to or subsets of other candidates based on their static instruction IDs from compiled assembly. Finally, we merge the remaining subgraphs with high overlap to create larger subgraphs with multiple outputs and select among them one or more final candidates with the highest total vertex weight in all their recurrences.

Table 4.1 shows our analysis on applications from the AxBench and Rodinia benchmark suites. The first two columns respectively denote the total number of candidate subgraphs identified and the number of unique subgraphs after filtering in step ❸. The third column is the average CI_Ratio among all filtered candidate subgraphs. The last column, *Memoization Coverage*, is measured by $\sum v_s \times \text{weight} / \sum v \times \text{weight}$ where $v_s \in V_s$ are vertices belonging to candidate subgraphs and $v \in V$ are all vertices of the DDDG. In other words, coverage is the vertex weight ratio of candidate sections to the entire graph. This fraction gives an estimate of the potential computation time that can be eliminated by memoization.

These results predict that benchmarks such as `Blackscholes` and `FFT`, both boasting a high CI_Ratio and coverage, will achieve significant speedup if hit rates are respectable after truncation. On the other hand, the obvious outlier `Jmeint`'s main function, which calculates triangle intersection using 16 floating-point inputs, resulted in an undesirable CI_Ratio and was not selected as a candidate, hence the low coverage for that benchmark. An important caveat of memoization coverage is that it does not always directly translate to an upper bound on performance speedup. Our DDDG model weighs each pseudo-instruction individually by its cycle time, yet modern processors are able to execute multiple instructions concurrently with multiple functional units.

Code Generation. With the final candidate computation blocks for memoization selected, step ❹ begins by selecting the number bits of the inputs to be truncated such that we can achieve a high hit rate while keeping output error within a given bound. To do so, we profile applications by observing error rates when truncating inputs by different numbers of bits. For all benchmarks evaluated, we truncate bits while constraining output error to less than 0.1% with the exception of `JPEG`, for which we use 1% instead since the output is an image. After determining the number of truncated bits, AxMemo instructions are inserted into selected code sections of the assembly and recompiled.

4.5 Evaluation

Benchmarks. Table 4.2 summarizes the benchmarks from AxBench [71] and Rodinia [54] that we evaluated. The benchmarks from AxBench cover a wide range of applications suitable for approximation, including financial

Table 4.1: The dynamic data dependence graph (DDDG) analysis of AxBench [71] and Rodinia [54] Benchmarks.

	Benchmark	Total # of Dynamic Subgraphs	# of Unique Subgraphs for Memoization	Compute / # of Inputs Ratio	Memoization Coverage
AxBench	Blackscholes	61,114	8	48.41	75.24%
	FFT	5,376	3	43.85	93.83%
	Inversek2j	840	4	38.13	67.91%
	Jmeint	516	4	9.87	53.10%
	JPEG	260	6	15.49	19.3%
	K-means	387	4	9.01	75.31%
	Sobel	32,288	2	23.81	35.3%
Rodinia	Hotspot	15,429	43	16.35	45.4%
	LavaMD	24,614	16	13.77	65.28%
	SRAD	110,003	2	21.58	45.20%

Table 4.2: Evaluated benchmarks.

	Benchmark	Domain	Description	Acronym
AxBench	Blackscholes	Financial Analysis	Calculates the price of European-style options	**BLK**
	FFT	Signal Processing	Radix-2 Cooley-Turkey FFT	**FFT**
	Inversek2j	Robotics	Calculates the coordinated of a two-joint arm	**I2J**
	Jmeint	3D-Gaming	Detects the intersection of two triangles	**JM**
	JPEG	Compression	Compresses an image using JPEG standard	**JPEG**
	K-means	Machine Learning	K-mean clustering on an image	**KM**
	Sobel	Image Processing	Applies Sobel filter on an RGB image	**SBL**
Rodinia	Hotspot	Physics Simulation	Simulates the temperature of an IC chip	**HS**
	LavaMD	Molecular Dynamics	Simulates interaction of particles with charge	**MD**
	SRAD	Medical Imaging	Image denoising	**SRAD**

analysis, signal processing, robotics, and machine learning.

To further evaluate the benefits of AxMemo across other domains, we include three other benchmarks—in the domain of physics simulation, molecular dynamics, and medical imaging—from Rodinia [54]. The input datasets used for evaluation are provided directly by the benchmark suites, as listed in Table 4.3. The third column of Table 4.3 lists the total size of memoization inputs in bytes for each benchmark. As defined in Section 4.2, memoization inputs are all inputs to the memoized computation block. The large sizes of the memoization inputs demonstrate the necessity for using CRC values as LUT tags. The last column in Table 4.3 indicates the number of truncated bits per input for each benchmark. This number is selected based on compiler analysis and profiling to achieve the highest hit rate while satisfying output error constraints as described in Section 4.4. Our compiler analysis and experiment both show that 32-bit CRC is generally large enough to avoid collision. We use the complete AxMemo compilation workflow (see Section 4.4) to generate memoization ready binaries for the evaluated benchmarks.

Quality metric and monitoring. Since we apply bit truncation on inputs and the hash function may result in collision, applications may suffer a degradation in output error. To assess output quality when the memoization is enabled, we use output error (Equation 4.2) defined as follows [29]:

$$E_r = \frac{\sum_i (\hat{X}_i - X_i)^2}{\sum_i X_i^2} \tag{4.2}$$

In the equation, X represents correct results from the unmodified source code

Table 4.3: Input dataset of benchmarks and memoization configuration. All datasets used are default and provided by the benchmark suite. Memoization input size denotes the total input size in bytes for each (logical) LUT. The number of tuples corresponds to the number of memoized blocks of the benchmark. The actual LUT tag in hardware is generated by hashing.

	Benchmark	Input Dataset	Memoization Input Size (bytes)	# of Truncated bits
AxBench	BLK	200K options	24	0
	FFT	4,096 floating-point data points	4	0
	I2J	1.24 million pairs of angles	8	8
	JM	Coordinates of 145K pairs of triangles	36	6
	JPEG	512x512 pixel images	(16, 16)	(2, 7)
	KM	512x512 pixel images	12	16
	SBL	512x512 pixel images	36	16
Rodinia	HS	512x512 maps of power and temperature	16	8
	MD	16x100 particles with random initial position	12	0
	SRAD	458x502 pixel medical images	24	18

and $\hat{X}$ represents results with AxMemo enabled. For the `JPEG` benchmark where the output is an encoded image file, we apply Equation 4.2 on individual pixel values instead. The output of `Jmeint` is a Boolean value indicating whether two 3-D triangles intersect. As such, we use misclassification rate (percentage of incorrect classifications). To ensure the output quality, we also implement quality monitoring scheme to prevent large output error. During the execution, every 1 out of 100 LUT hits is ignored. The LUT performs the lookup normally but returns a miss instead of hit. The LUT output will be later used to compare against the data sent by processor for updating the LUT. For each comparison, a simple relative error is calculated. For every 100 comparison, the statistics of the relative error is checked. If more than 10% of the relative errors are larger than 10%, the memoization is disabled.

4.5.1 Experimental Setup

Cycle-accurate simulator. All experiments in this section are assessed with the gem5 simulator [72] to evaluate the benefits of AxMemo. We use one of the gem5's default configuration which accurately models a high-performance in-order (HPI) ARM processor, as we target processors for low-power applications. The HPI processor, released by ARM, includes detailed configurations and timing parameters to model a modern in-order processor

Table 4.4: Major microarchitectural parameters for the ARM high-performance in-order (HPI) processor using `ARM-v8a` ISA.

Number of Cores, Frequency	Two cores, 2 GHz
Issue Width	Two, in-order
Number of Integer Units / Core	Two ALUs, One Multiplier, One Divider
Number of Floating-Point Units /	One
Number of Load/Store Units / Core	One
L1 Instruction Cache	32 KB, 2-way set associative, 1-cycle hit latency
L1 Data Cache	32 KB, 4-way set associative, 1-cycle hit latency
L2 Shared Cache	1 MB, 16-way set associative, 13-cycle hit latency
Memory Configuration	4 GB, 1600 MHz, DRR3, two channels

Table 4.5: Timing parameters for the AxMemo ISA extensions (Section 4.3).

AxMemo Instruction	Latency
ld_crc `dst, [addr], LUT_ID, n`	One cycle for each byte of data. This instruction does not stall CPU unless the input queue of the memoization unit is full
reg_crc `src, LUT_ID, n`	One cycle for each byte of data. This instruction does not stall CPU unless the input queue of the memoization unit is full
lookup `dst, LUT_ID`	Two cycles for L1 LUT and 13 cycles for L2 LUT. In case there is an already issued CRC operation to the same LUT, this instruction wait until the undergoing CRC operation finishes
update `src, LUT_ID`	Two cycles
invalidate `LUT_ID`	One cycle per each way-associativity

implementation using `ARM-v8a` ISA. Although we evaluate in-order processor, AxMemo can also be implemented in out-of-order processors as we explained in Sections 4.2 and 4.3. Table 4.4 summarizes key microarchitectural parameters of the configuration used for AxMemo. We modified the gem5 simulator to include all proposed ISA extensions and additional hardware necessary for AxMemo. One memoization unit is appended to each core of the processor. Table 4.5 shows timing parameters for the proposed instructions. We extract these parameters from synthesis results shown in Table 4.6. We use `CLANG/LLVM 3.4` and `GCC 4.8` to compile, analyze, and generate benchmark binaries equipped with memoization. We also enable maximum compiler optimization for all applications prior to transforming memoized sections.

Hardware synthesis. We implement all proposed microarchitectural units, including the 32-bit CRC unit, Hash Value Registers (16×32-bit) and LUTs of various sizes in Verilog. The 8-bit parallel 32-bit CRC unit uses an iterative algorithm and processes 8 bits of the input each cycle. To match the throughput of the CRC unit with the most common case of a 4-byte input, we unrolled the 32-bit CRC unit four times and apply pipelining. We use the

`Synopsys Design Compiler (K-2015.06-SP3-1)` with a FreePDK 45 nm technology model. To remain consistent with the process technology used to model other components, we properly scaled down synthesis results of the proposed microarchitectural units to 32 nm. Since all synthesized hardware components have a latency smaller than 0.5 ns, we do *not* need to reduce the baseline core clock frequency in our simulations (Table 4.4). The area overhead, energy consumption, and the timing parameters of the synthesized units are shown in Table 4.6. With the largest L1 LUT (`16 KB`), the added memoization units for all the cores consumes up to 0.17 mm^2 (2.08%) area per the HPI processor with an estimated area of 8.0 mm^2, estimated using McPAT version 1.3 [73] also with 32 nm technology (note that L2 LUT is partitioned from L2 cache). The quality monitoring unit uses comparison logic proposed in [37] (see Section 3.3.2), the area overhead is only 16.8 µm^2 and power overhead is 7.5 µW with a latency of 1.0 ns.

LUT hardware configurations. To better understand the benefits of AxMemo, we perform our experiments using various LUT configurations, all built from the same base design and a 32-bit CRC unit. We only vary the size of the LUTs and the number of levels of LUTs for each configuration. For the experiments using one-level LUTs with sizes (including both tag and data) ranging between `4 KB`, `8 KB`, and `16 KB`. In this case, we use small-sized LUTs to limit the areal overhead of dedicated SRAM arrays necessary for larger L1 LUTs (synthesis results listed in Table 4.6). The memoization unit can have an *optional* L2 LUT. The L2 LUT is partitioned from last-level cache (L2 cache in this case) and does not require dedicated SRAM. The size of L2 LUTs is either `256 KB` or `512 KB` which are quarter and half the size of L2 cache, respectively. When we evaluate a configuration with L2 LUT, we fix the L1 LUT size to `8 KB`. This design decision is to strike a balance between the performance and cost of hardware resource.

Energy modeling. We use CACTI 6.5 [67] to estimate access energy and latency of the LUTs and the 1 KB SRAM in the CRC unit. The total dynamic energy of the processor is estimated using McPAT version 1.3 [73]. We configure McPAT according to the gem5 configuration for HPI processor and use statistics from gem5 to calculate the number of accesses to each component. Finally, we feed number of accesses to McPAT to calculate the application's energy consumption.

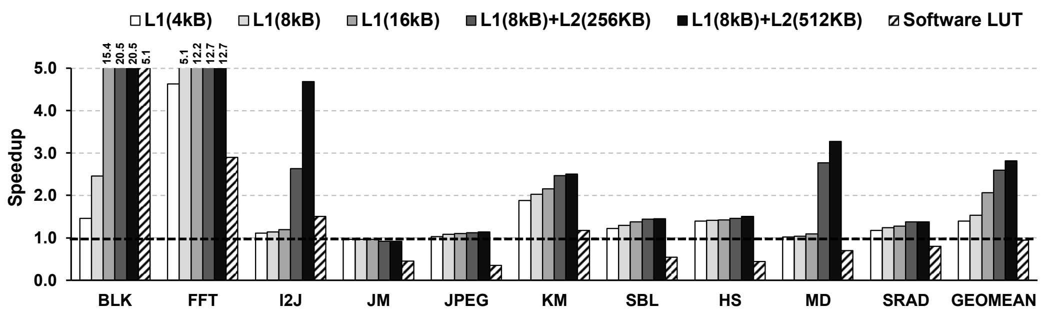
Table 4.6: Area, energy and timing analysis for `32 nm` technology node. CRC32 denotes 32-bit CRC. CRC32 unit shown here is already unrolled and pipelined. All LUTs are 8-way set-associative (4-byte LUT data).

	Area (mm2)	**Energy (pJ)**	**Latency (ns)**
CRC32 Unit	0.0146	2.9143	0.4133
Hash Register	0.0018	0.2634	0.1121
LUT (4 KB)	0.0217	3.2556	0.1768
LUT (8 KB)	0.0364	4.4221	0.2175
LUT (16 KB)	0.0673	7.2340	0.2658

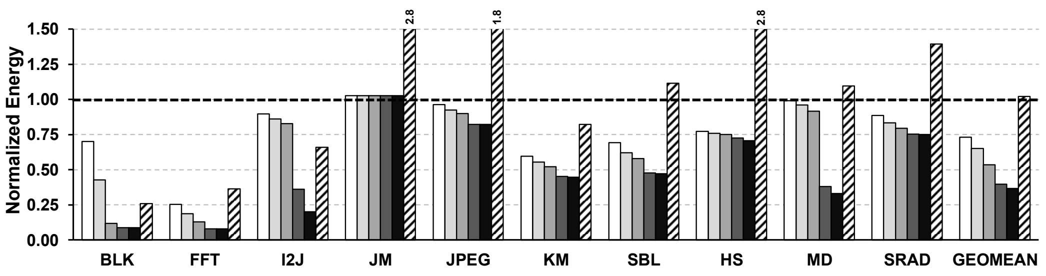
4.5.2 Experimental Results

We evaluate the benefits of AxMemo across a diverse set of benchmarks, including financial analysis, robotics, molecular dynamics, and machine learning. In all evaluations, the baseline is a regular ARM HPI processor with the same configuration but not equipped with memoization hardware. All results from hereon are normalized to this baseline. For each evaluation, we also perform a sensitivity study of the results for various AxMemo configurations: `L1(4 KB)`, `L1(8 KB)`, `L1(16 KB)`, `L1(8 KB)+L2(256 KB)`, `L1(8 KB)+ L2(512 KB)`. Bit truncation is applied identically for memoization inputs regardless of configuration. In addition to comparing against the baseline, we implement a software LUT for memoization as another contender. The software implementation of CRC uses an 8-bit parallel algorithm (Figure 4.3 in Section 4.2). To calculate the CRC value of a 4-byte input, the software implementation requires $4 \times 3 = 12$ instructions (1 AND, 1 LOAD and 1 XOR for each byte). We then use the CRC value to index entries of the LUT array using $CRC\%2^N$, where 2^N is the total number of entries in the LUT array. To reduce the cost of the software implementation, we use a simple array as the LUT. In contrast to the hardware implementation, the cost of increasing the number of LUT entries in software is significantly lower. As such, we simply increase the size of LUT arrays to the point where speedup plateaus. Based on these trials, we fix the number of LUT entries for the software implementation to 2^{28}, equivalent to 1GB for `4 B` data types. Recall that we use the remainder operation to obtain the array index, which means only the last 28 bits of CRC value are used to index the array. This may cause additional collision in the software LUT compared to AxMemo, which compares the whole 32-bit CRC value.

Performance and energy benefits with AxMemo. Figure 4.7a shows



(a) AxMemo Speedup



(b) AxMemo Energy normalized to baseline.

Figure 4.7: (a) Speedup and (b) normalized energy using different configurations of LUTs. L1 LUT is fixed at 8KB when there is an L2 LUT. Software LUT is the software memoization implementation.

full application speedup with AxMemo for different LUT configurations normalized to the non-memoized ARM HPI baseline, where the application executes normally on the CPU. Of all the benchmarks, `Blackscholes` enjoys the highest speedup of 20.5× for the `L1(8 KB)+ L2(512 KB)` configuration. The main reason for such a high speedup in `Blackscholes` is that almost the entire computation kernel of the benchmark, consisting of no less than 40 instructions, is replaced with a single LUT access. Out of ten benchmarks, only `Jmeint` exhibits virtually zero speedup for all tested AxMemo configurations. This is due to the low lookup hit rate for `Jmeint` (less than 0.1%), an indicator of low computation reuse available for AxMemo to exploit. We study the source of these benefits in the following paragraphs. On average, AxMemo delivers 1.40× and 2.82× speedup for the `L1(4 KB)` and `L1(8 KB)+L2(512 KB)` configurations respectively. Some benchmarks, such as `Blackscholes`, do not benefit from a large LUT. These benchmarks only need small LUT to capture most of their computation reuse. This is similar to the case when data cache becomes much larger than the working set of an application. In contrast to AxMemo, the software implementation, on average, suffers a slowdown by 0.94×. The underwhelming results of the software LUT implementation is largely due to the significant overhead of CRC calculation in software. Out of ten benchmarks, only `Blackscholes`, `FFT`, and `Inversek2j` enjoy speedup from the software implementation. For these three benchmarks, the sheer amount of computation replaced by lookup hits outweighs software overhead from performing memoization. Figure 4.7b shows the energy of each benchmark normalized to the baseline with no memoization. Similar to the trend of speedup results, the highest energy saving is achieved for `Blackscholes`, `FFT` and `Inversek2j`, whose energy is reduced to 0.09× , 0.08× and 0.20× of baseline, respectively. The `L1(4 KB)` and `L1(8 KB)+L2(512 KB)` configurations respectively reduce energy consumption to 0.73× and 0.37× of baseline on average for all the benchmarks. Once again, the software LUT implementation increases energy consumption by ∼2%, which we still attribute to the large overhead of CRC calculation in software.

Dynamic instruction count. Figure 4.8 shows the total dynamic instruction count of each evaluated benchmark normalized to the baseline without memoization. We show the breakdown of the dynamic instruction count between memoization instructions and normal instructions. We consider `ldr_crc` instructions not as a memoization instructions, but as a normal instruction because they simply substitute the original load. On average, AxMemo effectively reduces the number of dynamic instructions by 20.0% and 50.1% for `L1(4 KB)` and `L1(8 KB)+L2(512 KB)`, respectively. We observe the largest reduction in the number of dynamic instructions for `FFT`, where an overwhelming section of the application is replaced by memoization and hit rate is higher than 90%. This coincides with earlier compiler analysis showing a high memoization coverage for `FFT` in Section 4.4. The software memoization implementation, on the other hand, increases dynamic instruction count by ≈2.0×, causing most applications to see a slowdown and virtually zero energy reduction using the software implementation (see Figure 4.7). The total number of dynamic instructions is a proper indicator for the number of benefits that AxMemo can deliver. However, different instructions have different latencies, so dynamic instruction count alone does not determine exact speedup or energy savings.

Lookup-Table (LUT) hit rate. Figure 4.9 shows the LUT hit rate of the

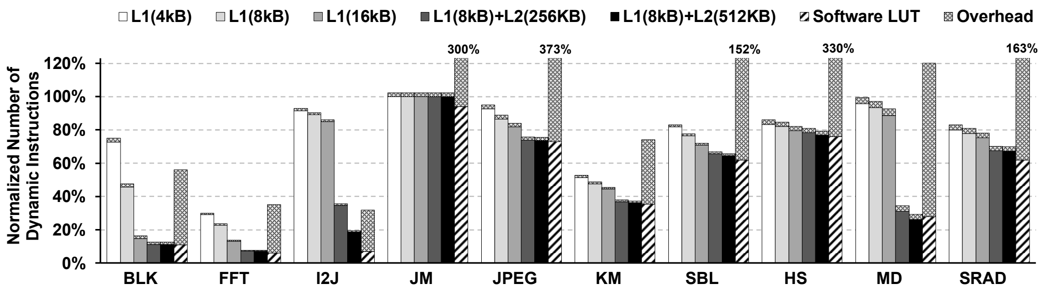


Figure 4.8: Dynamic instruction count normalized to the baseline with no memoization. The top bar in the stack (dotted bars) represents memoization instructions, which include the AxMemo instructions (Section 4.3) and the additional branch instructions.

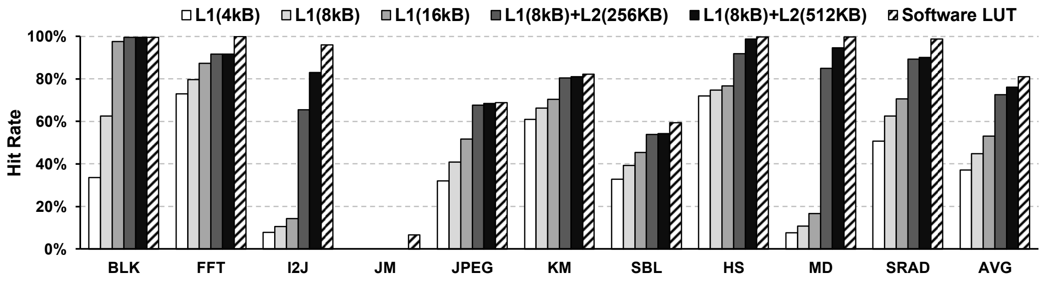


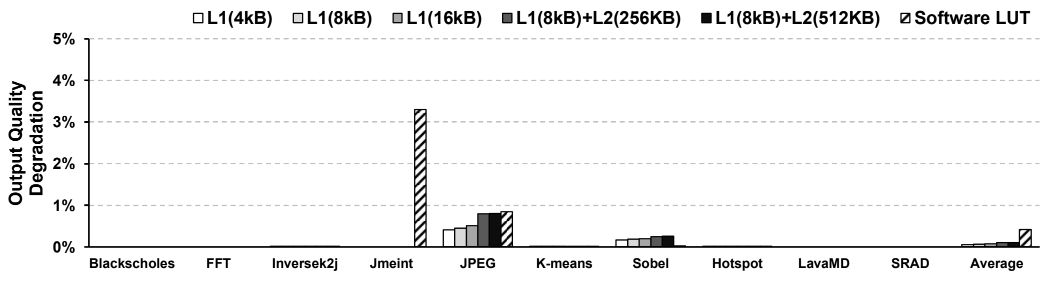
Figure 4.9: The LUT hit rate of various LUT configurations, including AxMemo and the software LUT implementation. **AVG** is the arithmetic mean.

evaluated benchmarks across different AxMemo configurations. For AxMemo configurations with multiple levels of LUT, namely `L1(8KB)+L2(256)` and `L1(8KB)+L2(512)`, we calculate the total lookup hit rate across both levels. The last bar for each benchmark shows the software implementation of our proposed memoization approach. On average, across all benchmarks, `L1(4KB)` (the smallest LUT size) and `L1(8KB)+L2(512KB)` (the largest LUT size) provide a 37.1% and 76.1% total hit rate respectively. Increasing from the smallest LUT size to the largest LUT size yields a 39.1% improvement in the lookup hit rate on average. This result shows the effectiveness of AxMemo's multi-level LUT design in improving hit rate. Note that the L2 LUT is inclusive, therefore adding the L2 LUT is effective in increasing the total hit rate but has minimal impact on the L1 LUT hit rate. The software LUT implementation, delivers an average hit rate of 81.1% across all benchmarks, slightly better than the 76.1% of the `L1(8KB)+L2(512KB)` configuration. As previously noted, the small overhead of increasing memory used in software allowed us to increase the LUT array's size to 1 GB, after which further increases no longer improve speedup. Figure 4.9 indicates that all the benchmarks except `Jmeint` exhibit significant computation reuse. The reason for `Jmeint`'s failure is its lack of repetitive or similar input patterns to the memoized computation block.

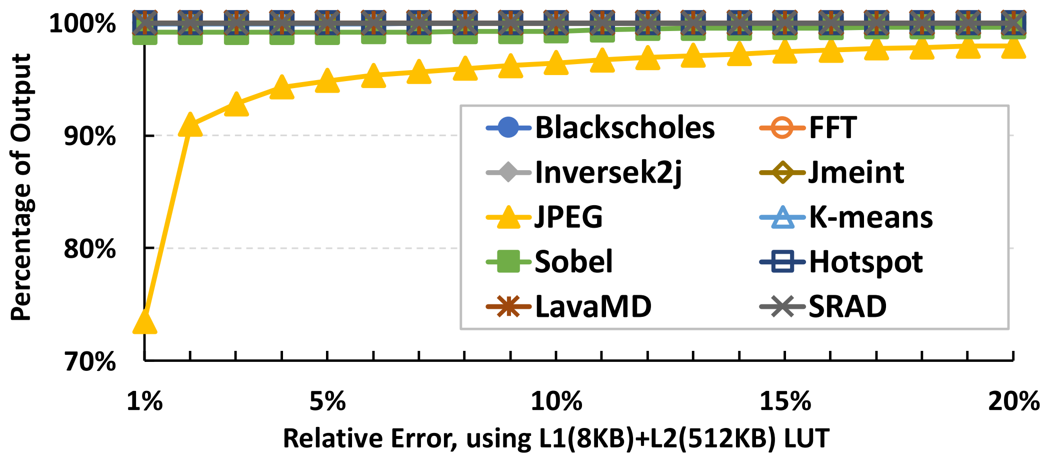
Output quality degradation. To assess the output quality degradation, we use the quality metric defined in Equation 4.2. Figure 4.10 shows the final output quality degradation of the evaluated applications across all the AxMemo configurations. To provide more detailed information about the quality, we also show the cumulative distribution function of the element-wise relative error of the output in Figure 4.10. On average, the output error E_r across all configurations of AxMemo falls below 1%. The main reason for this low output error with AxMemo is its virtually zero hashing collision rate. Furthermore, since we use the compiler analysis to determine the number of truncated bits in Section 4.4, the effects of truncation on the output quality will be minimal. Finally, errors from the LUT outputs do *not* always affect final values of the application. The last bar in Figure 4.10 shows output quality degradation for the software implementation, which has a *non-zero* collision rate (1% on average and up to 6.6%). As we previously detailed, the reason is that the four most significant bits of the hash value are discarded when indexing the LUT array.

Effectiveness of approximation. We use bit truncation as an approximation technique. To show the effectiveness of approximation, we assess the speedup and energy penalties of AxMemo when no truncation is performed. Figure 4.11a and Figure 4.11b show these speedup and normalized energy compared to AxMemo with truncation. On average, approximation improves the speedup and energy reduction by 12.1% (max. 32.8%) and 17.4% (max. 130%), respectively. Three benchmarks: `JPEG`, `Sobel`, and `SRAD` suffer from slowdowns and no longer enjoy energy savings without approximation. Without approximation, the average LUT hit rate drops from 76.1% to 47.2% across all benchmarks. Therefore, we conclude that input truncation is an effective approximation technique for AxMemo.

Comparison with prior work. The closest work to AxMemo is software-based approximate task-level memoization (ATM) [29]. To generate the hash key, ATM first concatenates the application's inputs into a single 1-D vec-



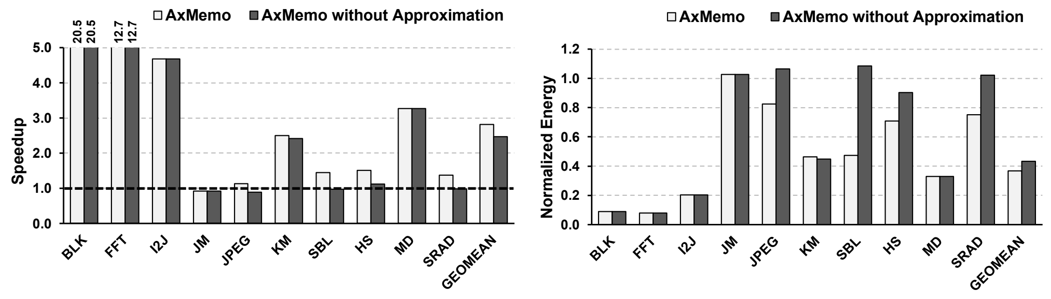
(a) Output error E_r



(b) Cumulative distribution function of element-wise relative error

Figure 4.10: Output quality. Software LUT has a higher error rate due to its higher collision rate.

tor of N bytes. Then, they create a vector of N indices, each of which points to one byte in the concatenated input vector. Finally, the indices are shuffled and used as the key for lookup accesses. Nonetheless, comparing AxMemo with ATM, we note that the only publicly available benchmark and datasets used both by ATM and AxMemo is `Blackscholes` from PARSEC benchmark suite [74]. As such, we have a head-to-head comparison with ATM for `Blackscholes` using a similar evaluation setup. The speedup for `Blackscholes` using their best implementation [29] is 2.5× *less* than the speedup delivered by AxMemo (8.2× vs. 20.1×). For the other "shared" benchmark `K-means`, though not exactly the same implementation, ATM also performs worse than AxMemo (2.0× vs. 2.5×).



(a) Speedup (b) Normalized Energy

Figure 4.11: Speedup and normalized energy of AxMemo, without approximation and with approximation. In both cases, AxMemo uses 8 KB L1 LUT and 512KB L2 LUT.

4.6 Conclusion

We propose AxMemo, an approximate memoization scheme. AxMemo focuses on replacing a long sequence of instructions with a few lookup table accesses, therefore reducing the total number of instructions executed. To enable memoization for large computation blocks with a different number of inputs and data types, AxMemo uses CRC to generate hash values to perform lookups. Furthermore, we apply approximation by truncating inputs to improve the LUT hit rate and observe output errors of only 0.2% on average. AxMemo is implemented with simple hardware with an area overhead of 2.08% and requires minimal software changes backed by our proposed compiler support. Our experiment results demonstrate that an AxMemo provides speedups up to 2.82× and energy reduction up to 63%, yielding a 7.67× energy efficiency improvement compared to the baseline processor.

CHAPTER 5

CONCLUSIONS

With the failure of Dennard scaling, novel approaches are needed to further improve the efficiency and performance of processors. In this dissertation, we proposed various techniques to reduce redundant computation on GPU or CPU, including G-Scalar, Lock and Load and AxMemo architectures.

We first proposed G-Scalar on GPU, a generalized scalar execution architecture along with a low-cost register value compression technique. G-Scalar can support scalar execution of not only conventional non-divergent arithmetic/logic but also divergent and special-function instructions. Furthermore, when GPUs adopt our low-cost register value compression technique, G-Scalar is practically free, as it is architected to share most of hardware resources with our register value compression technique, and reuse existing hardware resources of SIMT execution pipelines for scalar execution instead of implementing dedicated scalar execution pipelines. Our evaluation shows that G-Scalar, which consumes only 1% more chip space than the baseline GPU, can double the number of instructions eligible for scalar execution. This in turn improves power efficiency of GPUs by 24% and 15% compared with the baseline and previous scalar execution architectures, respectively. Lastly, our register value compression technique alone can reduce the power consumption of the register file by 54%.

Then we extend G-Scalar to similar values and proposed Lock and Load (LnL) where approximate computing is triggered by similarity of values returned by load instructions in a warp and then performed for an approximable code region followed by the load instructions. This not only reduces the overhead of checking the eligibility of approximation for every instruction but also allows us to deploy more sophisticated techniques for checking the eligibility of approximation and only approximating the output values for all the skipped threads at the end. Second, we enhance LnL to fuse consecutive approximable warps when both are executing approximable code and improve the performance because it reduces the number of fetched, decoded, scheduled and executed instructions. Our experiment shows that LnL can improve energy efficiency and performance by 62% and 23%, respectively with only 1% power/space cost.

Finally, we proposed AxMemo based on the same idea for LnL. AxMemo focuses on replacing a long sequence of instructions with a few lookup table accesses, therefore reducing the total number of instructions executed on CPU. To enable memoization for large computation blocks with a different number of inputs and data types, AxMemo uses CRC to generate hash values to perform lookups. We further apply approximation by truncating inputs, which improves the LUT hit rate. We observe output errors of only 0.2% on average with the approximation applied to AxMemo. AxMemo is implemented with simple hardware with an area overhead of 2.08% and requires minimal software changes backed by our proposed compiler support. Our experiment results demonstrate that an AxMemo provides speedups up to 2.82× and energy reduction up to 63%, yielding a 7.67× energy efficiency improvement compared to the baseline processor.

All three techniques focus on reducing the number of needed computations, by exploiting data similarity and computation redundancy. These techniques are proven to be effective for a variety of appellations that show such characteristics.

REFERENCES

[1] "The International Technology Roadmap for Semiconductors." [Online]. Available: http://www.itrs2.net/

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2011.

[3] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in GPGPUs," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2013.

[4] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep. 2011. [Online]. Available: http://dx.doi.org/10.1109/MM.2011.89

[5] S. Z. Gilani, N. S. Kim, and M. J. Schulte, "Power-efficient computing for compute-intensive GPGPU applications," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[6] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou, "Exploiting uniform vector instructions for GPGPU performance, energy efficiency, and opportunistic reliability enhancement," in *International ACM Conference on International Conference on Supercomputing (ICS)*, 2013, pp. 433–442.

[7] Y. Yang, P. Xiang, M. Mantor, N. Rubin, L. Hsu, Q. Dong, and H. Zhou, "A case for a flexible scalar unit in SIMT architecture," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014, pp. 93–102.

[8] A. Yilmazer, Z. Chen, and D. Kaeli, "Scalar waving: Improving the efficiency of SIMD execution on GPUs," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[9] K. Asanovic, S. W. Keckler, Y. Lee, R. Krashinsky, and V. Grover, "Convergence and scalarization for data-parallel architectures," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[10] S. Collange, D. Defour, and Y. Zhang, "Dynamic detection of uniform and affine vectors in GPGPU computations," in *Proceedings of the 2009 International Conference on Parallel Processing (Euro-Par'09)*, 2009.

[11] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten, "Microarchitectural mechanisms to exploit value structure in SIMT architectures," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2013.

[12] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning approximation for graphics engines," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[13] D. Wong, N. S. Kim, and M. Annavaram, "Approximating warps with intra-warp operand value similarity," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[14] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[15] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[16] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2014.

[17] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[18] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural acceleration for GPU throughput processors," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[19] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *Conference on Design, Automation and Test in Europe (DATE)*, 2010.

[20] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[21] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011.

[22] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2010.

[23] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Eliminating redundant fragment shader executions on a mobile GPU via hardware memoization," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2014.

[24] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications," in *IEEE Transactions on Mmultimedida*, vol. 15, no. 2, February 2013.

[25] B. Grigorian and G. Reinman, "Accelerating divergent applications on simd architectures using neural networks," in *IEEE/ACM International Symposium on Computer Design (ICCD)*, 2014.

[26] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[27] S. E. Richardson, "Caching function results: Faster arithmetic by avoiding unnecessary computation," Mountain View, CA, USA, Tech. Rep., 1992.

[28] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing, "LookNN: Neural network with no multiplication," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[29] I. Brumar, M. Casas, M. Moreto, M. Valero, and G. S. Sohi, "ATM: Approximate task memoization in the runtime system," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[30] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas, "SoftSig: Software-exposed hardware signatures for code analysis and optimization," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ser. ASPLOS XIII, 2008.

[31] D. A. Connors and W.-M. W. Hwu, "Compiler-directed dynamic computation reuse: Rationale and initial results," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1999.

[32] T. Tsumura, I. Suzuki, Y. Ikeuchi, H. Matsuo, H. Nakashima, and Y. Nakashima, "Design and evaluation of an auto-memoization processor," in *Proceedings of the 25th IASTED International Multi-Conference: Parallel and Distributed Computing and Networks*, 2007.

[33] M. Imani, D. Peroni, and T. Rosing, "NVALT: Nonvolatile approximate lookup table for GPU acceleration," *IEEE Embedded Systems Letters*, vol. 10, no. 1, 2018.

[34] S. Sinha and W. Zhang, "Low-power FPGA design using memoization-based approximate computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 8, 2016.

[35] G. Zhang and D. Sanchez, "Leveraging hardware caches for memoization," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 59–63, 2018.

[36] Z. Liu, S. Gilani, M. Annavaram, and N. S. Kim, "G-Scalar: Cost-effective generalized scalar execution architecture for power-efficient GPUs," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[37] Z. Liu, D. Wong, and N. S. Kim, "Load-triggered warp approximation on GPU," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2018.

[38] NVIDIA, "Fermi Architecture Whitepaper." [Online]. Available: http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf

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

[40] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: Enabling power efficient GPUs through register compression," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2015.

[41] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler, "A variable warp size architecture," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2015, pp. 489–501.

[42] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr., "Divergence analysis and optimizations," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[43] N. Brunie, S. Collange, and G. Diamos, "Simultaneous branch and warp interweaving for sustained GPU performance," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2012, pp. 49–60.

[44] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[45] M. Rhu and M. Erez, "Maximizing SIMD resource utilization in GPGPUs with SIMD lane permutation," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2013, pp. 356–367.

[46] M. Rhu and M. Erez, "The dual-path execution model for efficient GPU control flow," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 591–602.

[47] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 377–388.

[48] J. Zhan, M. Poremba, Y. Xu, and Y. Xie, "NoΔ: Leveraging delta compression for end-to-end memory access in NoC based multicores," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014.

[49] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 172–184.

[50] D. J. Palframan, N. S. Kim, and M. H. Lipasti, "COP: To compress and protect main memory," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2015, pp. 682–693.

[51] AMD, "AMD Graphics Cores Next (GCN) Architecture White Paper." [Online]. Available: https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf

[52] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2011, pp. 235–246.

[53] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, Mar 2012.

[54] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

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

[56] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for GPU architectures using McPAT," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 19, no. 3, pp. 26:1–26:24, June 2014.

[57] H. Yamauchi, "A discussion on SRAM circuit design trend in deeper nanometer-scale technologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 5, pp. 763–774, May 2010.

[58] H. Zhang, M. Putic, and J. Lach, "Low power GPGPU computation with imprecise hardware," in *Design Automation Conference (DAC)*, 2014.

[59] NVIDIA, "Pascal Architecture Whitepaper." [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[60] S.-Y. Lee and C.-J. Wu, "Characterizing the latency hiding ability of GPUs," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[61] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[62] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "GPU register file virtualization," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[63] NVIDIA, "CUDA Toolkit Documentation." [Online]. Available: http://docs.nvidia.com/cuda/parallel-thread-execution/

[64] M. Andersch, J. Lucas, M. Alvarez-Mesa, and B. Juurlink, "Analyzing GPGPU pipeline latency." [Online]. Available: http://lpgpu.org/wp/wp-content/uploads/2013/05/poster_andresch_acaces2014.pdf

[65] G. Ziegler, "Textures and Surfaces." [Online]. Available: http://on-demand.gputechconf.com/gtc-express/2011/presentations/texture_webinar_aug_2011.pdf

[66] NVIDIA, "CUDA Toolkit 4.0." [Online]. Available: https://developer.nvidia.com/cuda-toolkit-40

[67] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," in *HP Technical Report HPL-2009-85*.

[68] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," in *Proceedings of the IRE*, vol. 49, 1961.

[69] Y. S. Shao and D. Brooks, "ISA-independent workload characterization and its implications for specialized architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[70] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2014.

[71] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, vol. 34, no. 2, pp. 60–68, 2017.

[72] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Compututer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[73] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669172 pp. 469–480.

[74] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.