Power-efficient Computing for Compute-intensive GPGPU Applications

Syed Zohaib Gilani[†], **Nam Sung Kim**[†], **Michael J. Schulte**[‡]
[†]The University of Wisconsin-Madison, WI, U.S.A. [‡]Advanced Micro Devices, TX, U.S.A.
gilani@wisc.edu, nskim@engr.wisc.edu, michael.schulte@amd.com

Abstract

The peak compute performance of GPUs has been increased by integrating more compute resources and operating them at higher frequency. However, such approaches significantly increase power consumption of GPUs, limiting further performance increase due to the power constraint. Facing such a challenge, we propose three techniques to improve power efficiency and performance of GPUs in this paper. First, we observe that many GPGPU applications are integer-intensive. For such applications, we combine a pair of dependent integer instructions into a composite instruction that can be executed by an enhanced fused multiply-add unit. Second, we observe that computations for many instructions are duplicated across multiple threads. We dynamically detect such instructions and execute them in a separate scalar unit. Finally, we observe that 16 or fewer bits are sufficient for accurate representation of operands and results of many instructions. Thus, we split the 32-bit datapath into two 16-bit datapath slices that can concurrently issue and execute up to two such instructions per cycle. All three proposed techniques can considerably increase utilization of compute resources, improving power efficiency and performance by 20% and 15%, respectively.

1. Introduction

GPUs are massively parallel processors with a large number of execution units and high-bandwidth memory channels to concurrently run thousands of hardware threads. Although GPUs are originally designed for graphics applications, they are also used to accelerate GPGPU applications that can efficiently utilize the rich compute resources of GPUs.

As applications demand higher GPU performance, the manufacturers have integrated more compute resources and operated them at higher frequency, which has been mainly driven by technology scaling. However, such an approach significantly increases power consumption and thus it is not scalable under a power constraint. Note that the power reduction through technology scaling has been diminishing [1] while the power constraint that has not scaled [2]. Thus, we must considerably improve the power efficiency of GPUs to further increase the performance.

The compute resources of GPUs are organized into groups, such as streaming multiprocessors (SMs) in NVIDIA® GPUs. Each SM supports a single-instruction multiple-thread (SIMT) pipeline (i.e., execution pipeline) that consists of execution units (EUs), main register file (MRF), fetch/decode/scheduling (FDS) logic, and on-chip shared memory. For compute-intensive applications, SMs can consume more than 85% of the total GPU dynamic power and the FDS logic, MRF and EUs can dissipate more than 45% of the total SM dynamic power [3]. In this paper, therefore, we focus on these power-hungry components in SMs to improve power efficiency and performance of GPUs executing GPGPU applications.

Composite instructions: GPUs are typically optimized for floating-point-intensive applications. Thus, the execution pipeline of GPUs is usually comprised of a larger number floating-point (FP) fused multiply-add (FMA) units [4, 5, 6, 7]. However, the GPUs are also employed for accelerating integer-intensive applications, such as data compression, data encryption, and medical image processing. Consequently, the FP FMA units are often enhanced to perform integer (INT) arithmetic, bitwise, and logical operations, allowing the GPUs to utilize the same execution pipeline for INT instructions. Exploiting such FP FMA units, we can fuse a pair dependent INT instructions into a composite instruction that can be efficiently executed by an enhanced FMA unit. The composite instructions, which are formed by the compiler, reduce the total number of fetched/executed instructions, thereby improving both power efficiency and performance of GPUs.

Scalar unit: Many GPGPU applications exhibit considerable computational redundancy, which arises when all the threads in SIMT groups (i.e., a warp) produce the same result. Figure 1 shows that redundant computations constitute from 10% to 50% of the total fetched/executed instructions for the evaluated benchmarks (cf. the "Redundant" bars in Figure 1); refer to Section 6 for the detailed experimental methodology. The sources of this computational redundancy are (i) duplicated control instructions across threads in SIMT groups, (ii) operations with constants, (iii) memory address calculations, and (iv) inherent redundancy in pixel data in image and video processing applications. To exploit such computational redundancy within a warp, we dynamically detect an instruction that produces the same result across all the threads in a warp at runtime. Then we issue such an instruction to a separate *scalar pipeline* where its source and destination registers are kept in a separate *scalar register file* (SRF); a scalar unit is comprised of a scalar pipeline and a SRF. The scalar unit can improve both power efficiency and performance of GPUs by eliminating redundant computations and allowing the SIMT pipeline to concurrently execute another instruction. The baseline GPU assumed in this study (i.e., NVIDIA® Quadro FX5800) can issue up to two instructions (one non-transcendental to the FMA units and one transcendental to the special function units (SFUs)) per thread every cycle [8]. Note that we can utilize the same dual-issue capability for our technique to

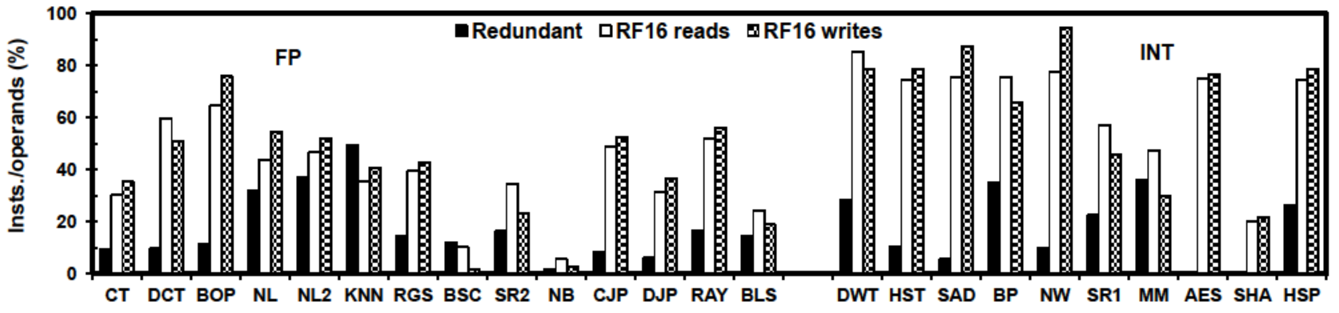

Figure 1: Percentage of total instructions that performed redundant computations and the percentage of RF accesses that utilize 16 bits or less.

issue computationally redundant instructions to the scalar pipeline for power efficiency and performance improvement.

Sliced datapath: GPUs typically employ the 32-bit datapath. However, we observe that 16 or fewer bits are sufficient for accurate representations of operands and results of many instructions. Figure 1 shows that a large percentage of values read from and written to the MRF require 16 or fewer bits for their accurate representations (cf. the "RF16 reads" and "RF16 writes" bars in Figure 1). Since the most significant bits (MSBs) of these values only contain the sign-extended bits, the accesses of the complete 32-bit registers are not necessary. This can be exploited to reduce MRF access energy or improve GPU performance by spliting the 32-bit datapath into two 16-bit datapath slices; using only one 16-bit datapath slice reduces MRF access energy while issuing two (16-bit precision) instructions, each of which is sufficient with the 16-bit datapath for accurate computations improves GPU performance. While 16-bit data types are supported by some GPUs, they are used only to reduce memory bandwidth pressure [9]. Moreover, the use of such data types must be explicitly specified by the programmer. In contrast, our technique dynamically detects such instructions and schedules them to the 16-bit datapath slice(s) to reduce MRF access energy or improve GPU performance.

The reminder of this paper is organized as follows. Section 2 depicts the baseline GPU architecture. Sections 2, 4, and 5 present our techniques proposed to improve the power efficiency and performance of GPUs. Section 6 evaluates the performance and power impact of the proposed techniques. Section 7 discusses the impact of the proposed techniques on future GPUs. Section 8 describes related work. Section 9 concludes this study.

2. Baseline GPU Architecture

In this study, we assume a baseline GPU architecture similar to the NVIDIA® Quadro FX5800 [8]. The baseline GPU, which is illustrated in Figure 2(a), consists of 30 SMs. Each SM contains an MRF (16384 32-bit registers), a warp scheduler, 8 EUs, 2 SFUs, and on-chip shared memory. The compute resources of an SM are organized into two SIMT clusters, as shown in Figure 2(b). Each SIMT cluster has 4 EUs and 4 MRF banks. Each MRF bank is dual-ported (1-read and 1-write ports) and has 512 128-bit entries. The multi-banked MRF architecture allows each SM to sustain the bandwidth of 24 32-bit reads and 8 32-bit writes every cycle per SM without employing multi-ported register file that can consume considerably more area and power. This bandwidth ensures that 8 single-precision FP FMA instructions, each of which needs to read 3 32-bit source operands and write 1 32-bit result, can be issued every cycle per SM. All the registers for a thread reside in the same bank and each thread performs multiple MRF accesses to read all of its source operands [8].

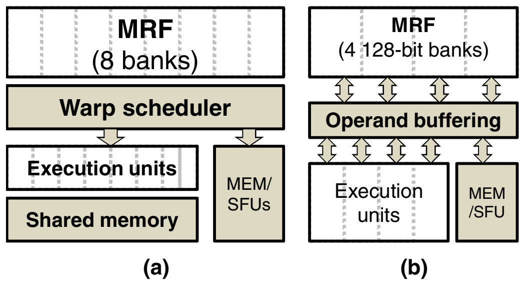

Figure 2: Baseline GPU architecture: (a) streaming multiprocessor (SM) and (b) SIMT cluster.

Each SM schedules warps (i.e., threads in groups of 32 threads). Each warp is issued over 4 cycles with 8 EUs. The EUs are typically comprised of FP FMA units enhanced to also execute INT instructions [4, 5, 6, 7]. We assume that the pipeline latency of the FMA units is 8 cycles. Consequently, most instructions (excluding double-precision and transcendental instructions) have 8-cycle read-after-write (RAW) latency [4, 5, 6]. Finally, the warp scheduler can issue up to two instructions to the FMA units and the SFUs per cycle [8].

3. Composite Instructions

3.1 Enhanced FMA Unit

An FMA unit with INT execution enhancements is illustrated in Figure 3(a). For FP FMA operations, the FMA unit multiplies the significands of the FP operands using the signifand multiplier (SMUL) and aligns the product to the addend using the alignment shifter (ASFT). The alignment

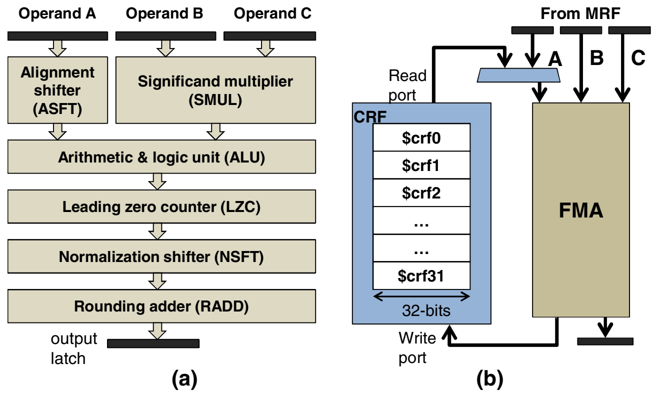

Figure 3: SM execution unit: (a) baseline FMA unit and (b) composite register file (CRF) integrated with an FMA unit.

shifter ensures that both the product and the addend have the same exponent. The shifted product is added to the addend using the ALU and the result is left-shifted using the normalization shifter (NSFT) until all the leading zeros are shifted out. The leading zero counter (LZC) is used to determine the amount of the normalization shift. The normalized result is rounded using the rounding adder (RADD) and is re-normalized if required (in case of the overflow during rounding).

The ALU can be also used to perform INT arithmetic, logical, and bit-wise operations. The ASFT and NSFT can be also used to perform right and left shifts on INT operands, respectively. The LZC, NSFT, and RADD are also used to convert FP to INT data types and vice versa. Finally, 32-bit INT multiplications can be performed using the SMUL. These INT execution enhancements are already supported by the baseline GPU to utilize the same FMA unit (and execution pipeline) for both INT and FP operations [4, 5, 6, 7].

3.2 Motivation and Approach

We propose to combine pairs of dependent INT instructions into composite instructions to reduce the number of fetched/executed instructions and increase the utilization of the FMA units. The main motivation behind this approach is that the energy-per-instruction (EPI) in GPUs is not dominated by the EUs; considerable energy is consumed by the instruction cache/buffer, decoder, warp/instruction schedulers, register file, pipeline flip-flops and clock tree. We measure the energy consumption of the NVIDIA® FX5600 GPU using INT and FP micro-benchmarks. Our measurement indicates that executing two INT ALU instructions consumes about 74% more energy than a single FMA instruction on average. Thus, the energy reduction due to the reduced number of fetched/executed instructions offsets the energy overhead of using the FMA units enhanced to execute composite instructions.

Dependent instructions to form a composite instruction are chosen such that they can easily be mapped to the compute resources within the FMA unit. The composite instructions are added to the instruction set architecture (ISA) of the GPU. Adding such instructions to a GPU ISA is simpler than adding them to a CPU ISA. This is because the GPUs typically employ an intermediate language (e.g., NVIDIA's PTX) and the code based on it is converted to the a platform-specific ISA by the compiler in the GPU driver at runtime. The identification of dependent instruction pairs and the formation of composite instructions are performed statically at compile time. The motivation is to increase the amount of completed work per instruction (i.e., to reduce the number of fetched/executed instructions per work) while utilizing the same execution pipeline as much as possible.

The MRF bandwidth is optimized to read three 32-bit operands every cycle per FMA unit and most INT operations require only two or fewer operands per instruction. As a result, we can combine two dependent INT operations such that they fully utilize the MRF read bandwidth. A composite register file (CRF) is added to each FMA unit to hold the intermediate results of composite instructions (i.e., the results of predecessor instructions in the combined pairs). Dependent operations in composite instructions read the source operands from the CRF. Figure 3(b) shows a CRF providing 32 32-bit registers and integrated with the FMA pipeline. The CRF is also dual-ported (1-read and 1-write ports) like MRF banks and it has a dedicated slot, which is indexed by the warp identification number (warp ID), for storing one intermediate result per thread. The 32-entry CRF ensures that one intermediate result can be stored for each active warp; the maximum number of active warp is 32 in the baseline GPU architecture. The use of the CRF also reduces the number of MRF accesses; the MRF consumes far more energy per access than the CRF because it has 16384 32-bit registers per SM.

3.3 Implementation Details

The composite instructions, which can be supported by the FMA unit, are tabulated in Table 1, along with the FMA resources that are used for each composite instruction. We enhance the FMA unit to implement bidirectional shifters for both ASFT and NSFT. This increases the number of supported composite instructions by allowing both the producer and consumer instructions to be either left or right

Table 1: Composite instructions and the corresponding FMA resources utilized.

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

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

(a)

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

(b)

Figure 4: Formation of composite instruction: (a) original instruction sequence and (b) new sequence with composite instructions.

shifts. In order to form composite instructions, the compiler can perform a pass over the code based on an intermediate language such as NVIDIA® PTX. The compiler pass identifies the pairs of INT operation listed in Table 1. If the intermediate results are required by other instructions, the compiler ensures that they are read from the CRF.

During the formation of composite instructions, the compiler ensures that only one of instructions in a composite instruction performs a memory access to avoid pipeline stalls due to bank conflicts of the shared memory. Moreover, branch boundaries are never crossed during the search for instructions that can be combined. Finally, the compiler pass also ensures that the CRF entry is no longer needed by any dependent instruction before allowing another composite instruction to overwrite it.

Figure 4(a) shows a code snippet from the DCT benchmark. The compiler is able to detect two composite instructions from the given code sequence (highlighted in the figure). Each pair of combined instructions is transformed into a single composite instruction as illustrated in Figure 4(b). The two composite instructions utilize the FMA resources ALU-NSFT and ALU-RADD, respectively. For the second composite instruction (i.e., PC: 0x228 in Figure 4 (b)), the intermediate result is stored in the CRF entry indexed by the corresponding warp ID. The instruction at PC 0x230 thus reads its operand from the CRF.

4. Scalar Unit Exploiting Computational Redundancy

4.1 Motivation and Approach

Many applications execute a significant percentage of instructions in which all the threads in a warp produce the same result. There are several reasons for this computational redundancy. First, each thread typically consists of two types of computations: control and data. While the data computations can be different across threads in a warp, the control computations (e.g. loop increments, memory address calculations) are often the same across all the threads in a wrap. Second, even for data processing, some applications exhibit data redundancy. For example, in image and video processing applications, the values of pixels in an image region processed by a warp may not not change if the region exhibits the same color. Moreover, computations involving

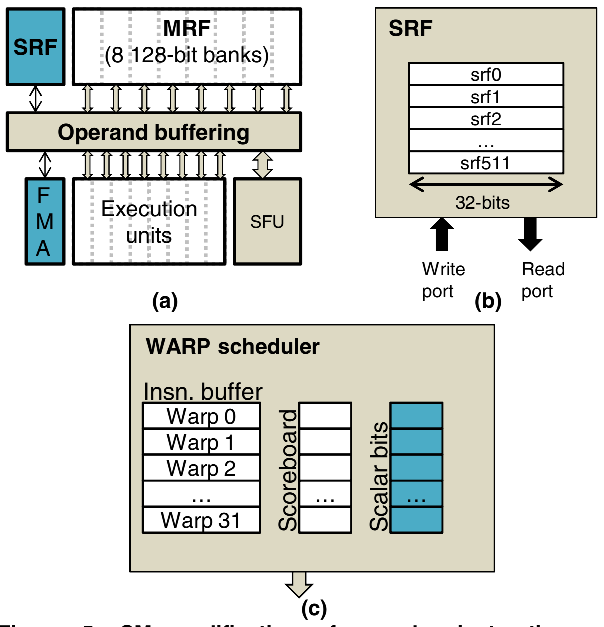

Figure 5: SM modifications for scalar instructions: (a) modified SM, (b) scalar register file (SRF), and (c) scheduler modifications.

constants also increase the probability of redundant computations since one of the operands is fixed across all the threads. Consequently, several image processing benchmarks that utilize many constant values have a high percentage of redundant computations (e.g. *knn* in Figure 1).

We can exploit the computational redundancy to increase instruction throughput by adding a separate scalar unit to an SM. The scalar unit, which consists of an SRF and an FMA unit, is highlighted in Figure 5(a). If each input operand of an instruction is the same for all threads in a warp, the instruction is issued to the scalar unit; otherwise it utilizes the SIMT pipeline. Although our baseline GPU does not have a scalar unit, Intel's Larrabee and AMD's recently announced GPUs also include scalar units [10, 7], which can be independently power-gated for applications that do not exhibit significant computational redundancy. Therefore, the area and power overhead of our approach is negligible for such GPUs.

4.2 Implementation

In order to detect redundant operations, we keep track of a set of warp registers (i.e., 32 32-bit physical registers) that store the same value for all 32 physical registers. This is achieved by adding a comparison stage before the write-back stage in the SIMT pipeline. The comparison stage checks whether or not the register write-back values across all the threads in a warp are the same. If all the values are the same, the scalar bit is set for the corresponding register in the warp scheduler (cf. Figure 5(c)) and a single 32-bit result is written to the SRF.

The SRF has a single bank with 512 32-bit entries as illustrated in Figure 5(b). Operands for a scalar instruction

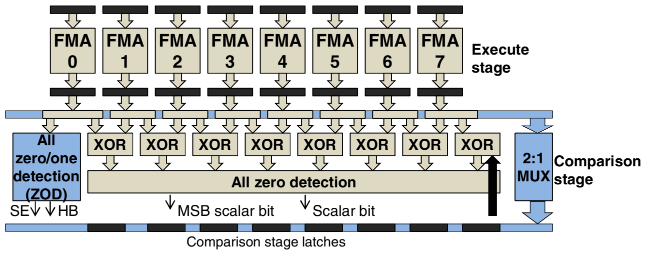

Figure 6: Comparison stage added to the execution pipeline.

are read over multiple cycles, similar to the SIMT pipeline. This keeps both the SIMT and scalar pipelines in synchronization and also allows the dual-ported (1-read and 1-write ports) SRF to provide the sufficient bandwidth for scalar instructions. Threads in a kernel have common logical register values, but each thread's registers are mapped to different physical registers in the MRF banks. The SRF registers are addressed using the logical register value of a thread. More specifically, the SRF is indexed by *warp ID* × *kernel register count* + *logical register ID*. Since the value of *kernel register count* is known at compile time, the first indexing term (i.e., *warp ID* × *kernel register count*) can be preloaded into an SM during the kernel launch phase. During warp scheduling, the scalar bits for all source operands are checked in parallel by the score-boarding logic to determine which pipeline (SIMT or scalar) the instruction should be issued to. If all the source operands of an instruction reside in the SRF, it is issued to the scalar unit; otherwise it is issued to the SIMT pipeline.

The comparison stage added to the SIMT pipeline is illustrated in Figure 6. A single stage of XOR gates is used to compare write-back values from adjacent SIMT lanes. Since a 32-thread warp is scheduled in four SIMT groups, each of which consists of eight threads, one XOR block is employed to compare the current write-back value with one of the write-back values from the previous cycle; the write-back value of the previous cycle is read from the pipeline latches. The results from the XOR gates are applied to the all-zero detection circuit, which can have a delay equivalent to one or two gates using a dynamic circuit [11]. The all-zero detector sets the scalar bit to zero if there is any binary one present among its input bits, indicating that the write-back values are not the same; otherwise, the scalar bit is set to one. The registers for threads in a warp are replaced with a single scalar register if the scalar bit is set for all four SIMT groups of a warp.

The scalar unit provides three advantages. First, a considerable percentage of high energy MRF accesses are replaced with low energy SRF accesses; the SRF (one 32-bit 512-entry bank) consumes much less energy per access than the MRF (eight 128-bit 512-entry banks) because the SRF is considerably smaller than the MRF. Second, the computation is carried out only for a thread instead of 32 threads in the warp, reducing execution energy as well. Third, a significant performance improvement can be obtained though issuing two instructions to both the SIMT and scalar pipelines together. The baseline SM can already issue up to two instructions per warp (one non-transcendental instruction to the FMA units and one transcendental instruction to the SFUs) [8]. With our technique, we utilize the same dual-issue logic to issue one (scalar) instruction to the scalar unit and one (SIMT) instruction to the SIMT units.

5. Sliced Datapath

5.1 Motivation and Approach

The MRF can consume up to 15% of the SM dynamic power [12]. On the other hand, we observe that 16 or fewer bits are sufficient for accurate representations of operands and results of many instructions. Figure 1 shows that as many as 80-90% of values read from and written to the MRF require only 16 or fewer bits for accurate representations of the values for some INT applications; the 16 MSBs of such values contain only the sign-extended bits. Consequently, the 16 MSBs of registers storing such values do not impact the accuracy of computations, wasting energy for each access. This can be exploited by spliting the 32-bit datapath into two 16-bit datapath slices, allowing accesses of only the 16 least significant bits (LSBs) of registers to decrease MRF access energy or issues of two (16-bit precision) instructions, each of which is sufficient with the 16-bit datapath for accurate computations to increase GPU performance.

5.2 Implementation

We split the 128-bit wide MRF bank illustrated in Figure 7(a) into two 64-bit wide bank slices as shown in Figure 7(b); the 16 MSBs and 16 LSBs of each 32-bit register are

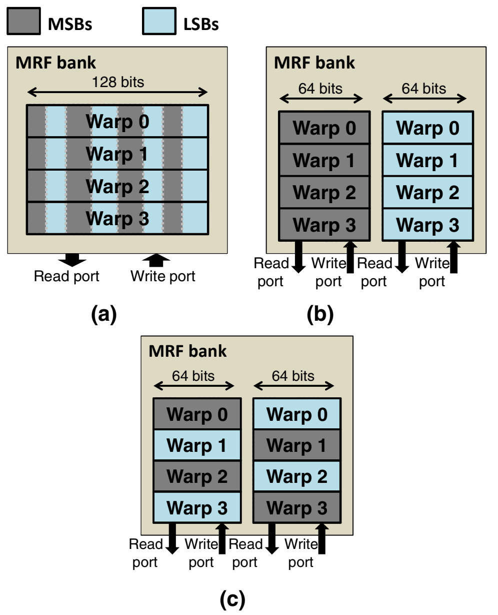

Figure 7: MRF bank sub-division to improve GPU performance: (a) baseline MRF bank (b) sliced MRF bank (c) sliced MRF bank with modified data placements.

stored across these two bank slices. Consequently, energy consumption for read and write accesses that require only the 16 LSBs can be reduced by nearly 50%. However, in order to determine whether or not operands requires only 16-bit accesses, two additional bits are added to every warp register (i.e., 32 32-bit physical registers). A half bit-width (HB) bit indicates that only the 16 LSBs are needed to be read from the 32-bit registers and a sign-extension (SE) bit specifies that the sign of values stored in the 32-bit registers. These bits are stored in the warp scheduler, similar to the scalar bits introduced in Section 4. When an instruction is scheduled, the HB and SE bits are checked to determine whether or not both MSB and LSB bank slices are required to be accessed.

The all zero/one detection (ZOD) unit in the comparison stage checks the 16 MSBs of each write-back value to see whether or not they are sign-extended bits. At the same time, the logic that computes the scalar bit introduced in Section 4 also determines the MSB scalar bit, which is set when the 16 MSBs are the same across all the write-back values from a warp. If the MSB scalar bit is set, the HB bit is set along with an appropriate SE bit value based on the sign of the values.

The sliced datapath can also be utilized to increase the GPU performance. Note that the baseline architecture can provide the bandwidth of three 32-bit reads and one 32-bit write [12]. On the other hand, (16-bit precision) instructions that are sufficient with the 16-bit datapath utilize only a half of the read and write bandwidth of the baseline architecture. Therefore, the GPU can dual-issue such instructions from two different warps using the sliced FMA units that can support two 16-bit precision computations in parallel [13, 14]. The two 16-bit datapath slices can be combined to form a single 32-bit datapath.

Although 16-bit precision instructions only utilize a half of the read and write bandwidth of the baseline architecture, issuing two such instructions from two different warps can be challenging the sliced MRF organization depicted in Figure 7(b). This is because all the 16 LSBs reside in the same MRF bank, likely leading to MRF bank conflicts. In order to reduce such bank conflicts, we can swap the placement of 16 LSBs with that of the 16 MSBs for every other warp. This is achieved by adding multiplexers in the comparison stage. The modified data placements in the sliced MRF are illustrated in Figure 7(c). Such placements enable the GPU to issue two 16-bit precision instructions from odd and even warps in the same cycle. Instead of using a unified warp scheduler, we also split the warp scheduler into two smaller warp schedulers, each of which is half as complex as the warp scheduler in the baseline GPU architecture; one schedules instructions from odd warps and the other from even warps. If both schedulers have 16-bit precision instructions that are ready to be issued to the 16-bit datapath in a given cycle, they are issued simultaneously. An additional instruction decoder is also needed to issue two such instructions from the two warp schedulers.

When the result of an instruction with 16-bit operands requires the complete 32-bit precision and thus 32-bit write bandwidth, the pipeline is stalled for a cycle. In practice, however, such stalls are rare; our experiment shows only 0.1% of EU accesses cause stalls on average. Note that these stalls do not reduce the performance of the GPU below that of the baseline GPU since each SM can still execute 8 instructions per cycle even if all the instructions with 16-bit operands incur stalls. In the absence of stalls, the performance of instructions untilizing the 16-bit datapath can be increased to 16 instructions per cycle per SM.

Although the large register file banks in the baseline GPU can lead to congestion in the RF layout, our sliced MRF banks do not increase this layout congestion. This is because the layout congestion is caused by the large number of bit-lines in the SRAM arrays of the register file while we do not increase the number of bit-lines or their routing. We add 2:1 multiplexers for 16-bit MSBs in operand buffers of EUs to select a bank slice for either the 16-bit LSBs (for 16-bit precision instructions with 16-bit precision) or the 16-bit MSBs (for 32-bit precision instructions).

6. Evaluation

We use GPGPU-Sim [15] to evaluate the performance impact of our proposed techniques. The simulator is configured to model a GPU similar to the NVIDIA® Quadro FX5800. The simulator configuration is summarized in Table 2. GPGPU-Sim is enhanced to maintain access rates for different architectural components. These access rates, which are used in the GPU power model proposed by Hong *et al.* [3], help to estimate the impact of our techniques on power consumption. The benchmarks used in this study and their acronyms are tabulated in Table 3.

6.1 Performance Impact

Figure 9 shows the percentage of total dynamic instructions that benefit from each of our proposed techniques. For the FP-intensive benchmarks, the percentage of composite

Table 2: Simulator configuration (see [15] for details).

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

Table 3: Benchmarks and their acronyms.

Benchmark	Acronym	Benchmark	Acronym
Texture-based convolution [9]	CT	Ray tracing [25]	RAY
Discrete cosine transform [9]	DCT	Simple CUDA BLAS [9]	BLS
Binomial options pricing [9]	BOP	Discrete wavelet transform	DWT
Image denoising [25]	NL	Image histogram [9]	HST
Image denoising [25]	NL2	Sum-of-absolute-differences [25]	SAD
Image denoising [25]	KNN	Back-propagation [26]	BP
Recursive Gaussian filter [9]	RGS	Needleman-Wunsch [26]	NW
Black-Scholes options pricing [9]	BLK	SRAD computation 2 [26]	SR1
SRAD computation 1 [26]	SR2	Matrix multiplication [9]	MM
n-body simulations [9]	NB	AES encryption [25]	AES
JPEG encoding [25]	CJP	SHA encryption [25]	SHA
JPEG decoding [25]	DJP	Hot-spot [26]	HSP

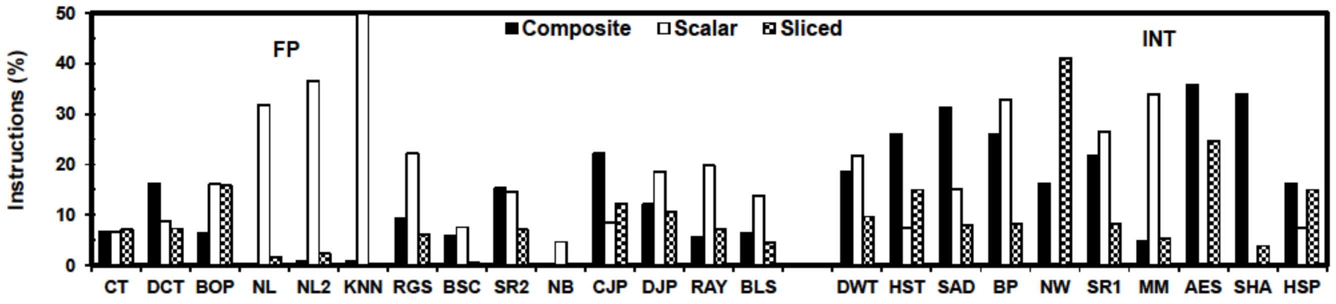

Figure 9: Percentage of total instructions that benefit from each of our proposed techniques.

instructions (cf. "Composite" bars in Figure 9) is less than that for the INT-intensive benchmarks. This is because composite instructions can only be formed for INT instructions and the FP-intenisve benchmarks exhbit a low percentage of INT instructions. For encryption benchmarks, up to 35% of instructions can be combined to form composite instructions. These benchmarks perform numerous shift and add operations that can be easily mapped into composite instructions.

Image processing, machine learning and control-processing intensive kernels such as NLM, NLM2, KNN, SRAD2, DJPG, BP, DWT and MM perform a large percentage of scalar operations (cf. "Scalar" bars in Figure 9). The reasons for their high computational redundancy have been described in Section 4. Encryption algorithms (SHA and AES) exhibit the least amount of computational redundancy. This is because these algorithms perform many bit-level manipulations that are different for each thread. The number of scalar instructions depends upon the control and data redundancy within warp instructions. The percentage of control and data redundancy is application-dependent. For MM, BSC, BOP, BLS more than 90% of the computational redundancy is due to control or memory-address calculations whereas for NL, NL2, KNN, RGS, HSP data duplication is the main reason for computational redundancy.

Although the percentage of 16-bit operands read from and written to the MRF is fairly high (cf. Figure 1), the percentage of 16-bit precision instructions is less (cf. "Sliced" bars in Figure 9). This is because instructions can be dual-issued to the 16-bit datapath slices only if two 16-bit precision instructions from odd and even warps are ready simultaneously in the same scheduling cycle.

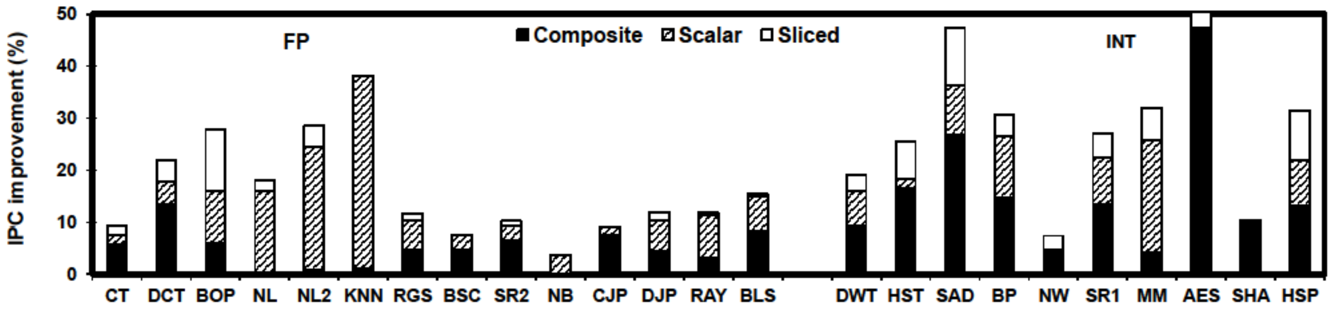

Figure 8: Performance impact of proposed techniques.

Figure 8 shows the performance impact of our approaches for FP and INT benchmarks. Overall, the INT benchmarks exhibit higher speedups than the FP benchmarks. The speedup obtained generally correlates with the percentage of instructions that benefited from our techniques. In some cases (e.g. SHA), however, the memory-intensive nature of the application reduces the overall performance benefit. NW has a high percentage of 16-bit precision instructions, but it also has a considerable percentage of branch divergence. Its performance improvement is thus limited by branch divergence serialization.

Moreover, as we apply our techniques successively, the performance improvement of some benchmarks starts to saturate because memory latencies and arithmetic latencies start to become more dominant. Consequently, for many benchmarks (e.g. RGS, CJPG, DJPG, SRAD2, RAY, BLAS and SRAD1), the relative performance improvement using scalar units and 16-bit datapath slices are less than the percentage of instructions that can benefit from them. Overall, our techniques provide a geometric mean speedup of 25% and 12% for INT and FP benchmarks, respectively.

6.2 SM Dynamic Energy Consumption

Figure 10 shows the breakdown of SM dynamic energy consumption of the baseline, composite, scalar, and sliced architectures denoted by "Base," "Comp," "Scalar," and "Sliced." "Comp" supports only composite instructions. "Scalar" adopts scalar units along with the support of composite instructions. "Sliced" employs all three of our techniques: composite instructions, scalar units, and sliced datapath. The constant energy shown in the breakdown models several components of an SM (e.g. frame buffers) that consume a relatively constant amount of power as long as the SM remains active [3]. Overall, all three proposed techniques collectively offer a geometric mean SM energy reduction of 21%. The energy reduction is higher for INT benchmarks (28%) than FP benchmarks (16%) since our techniques benefit the INT benchmarks more than FP benchmarks.

FDS energy: Composite instructions can reduce the number of fetched/executed instructions and thus the energy consumption of FDS logic. The percentage reduction of fetched/executed instructions is equal to the percentage of executed composite instructions (cf. Figure 9). Overall, composite instructions reduce the FDS energy by 17%. The INT benchmarks have a higher percentage of composite instructions than FP benchmarks, leading to more FDS energy reduction.

MRF energy: The dynamic and leakage energy consumption of MRF, CRF, SRF and sliced MRF is obtained using CACTI [16]. A similar approach to estimating MRF power consumption was taken by Gebhart *et al.* [12]. The leakage power and dynamic energy per access of CRF and SRF are included in the power model to estimate the overheads of our approaches. However, these structures do not contribute significantly to the overall SM power consumption.

Table 4 tabulates the energy cost of the MRF, CRF and SRF as obtained using CACTI. The read and write energy cost per 32-bit register of CRF is nearly ten times less than that of the MRF due to its smaller size. Similarly, the SRF reduces the energy per access by nearly 20 times. Finally, utilizing the sliced datapath for operands with 16 or less bits

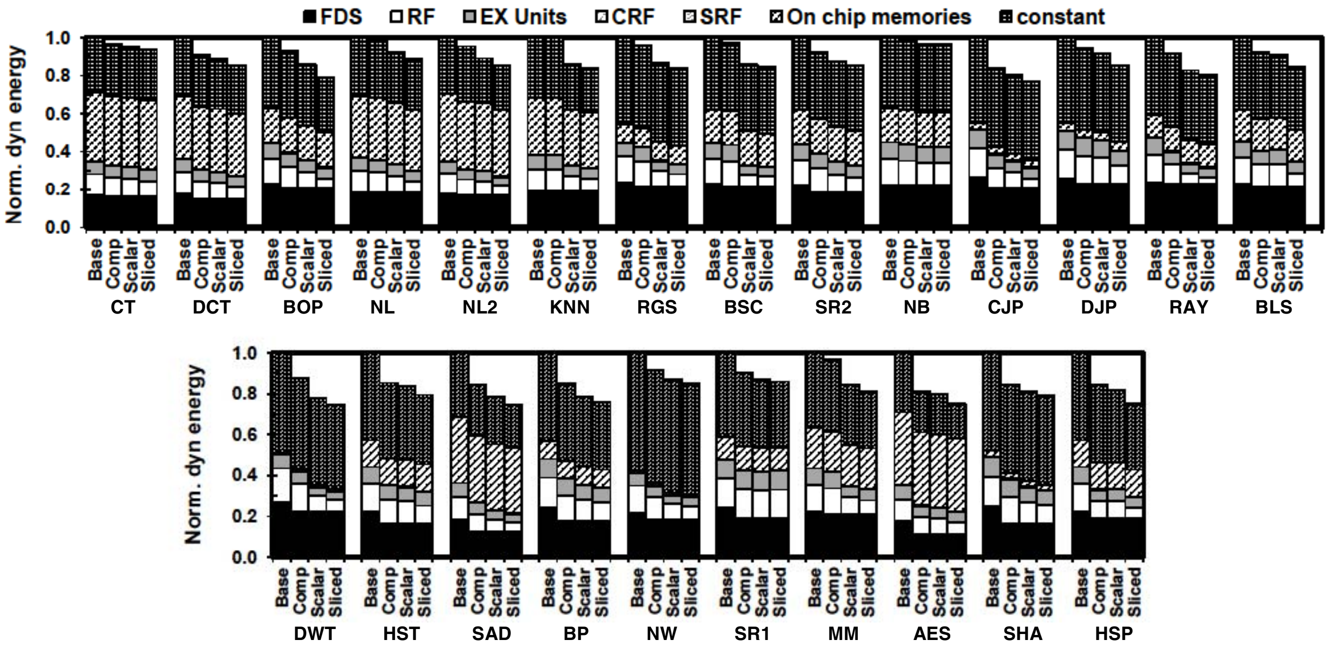

Figure 10: Dynamic energy consumption per SM: FP (top) and INT (bottom).

Table 4: Energy per access and leakage power of RFs.

	MRF	CRF	SRF	Sliced MRF
Organization	8 banks 128 bits 512-entry	32 banks 32 bits 32-entry	1 bank 32 bits 512 entries	16 banks 64 bits 512 entries
Read energy	2.8×10^{-2}nJ	7.1×10^{-4} nJ	1.4×10^{-3} nJ	1.4×10^{-2} nJ
Write energy	2.8×10^{-2} nJ	7.4×10^{-4} nJ	1.4×10^{-3} nJ	1.4×10^{-2} nJ
Lkg. power	56.48mW	8.96 mW	1.68 mW	56.64 mW

Table 5: Energy overhead of additional logic.

	Dynamic energy	Leakage energy
Execution stage	28.48pJ	9.76pJ
Additional logic	0.72pJ	0.01pJ
Overhead	4.3%	0.3%

reduces the access energy by nearly 50%.

All of our proposed techniques reduce MRF energy consumption. Composite instructions reduce MRF accesses by accessing the CRF for intermediate results. The scalar instructions utilize the SRF for redundant computations instead of the MRF, and the sliced instructions reduce MRF energy consumption by reducing the bank widths and accessing half bit-width banks for a large percentage of operands. Overall, our proposed techniques reduce the MRF energy consumption by nearly 50%.

EU energy: We enhance the FMA units to include input multiplexers for operands read from the CRF and swapping the 16-bit LSBs and MSBs of operands read from the sliced MRF for odd warps. We also implement bidirectional shifters for ASFT and NSFT. The estimated energy overhead of these enhancements over the baseline FMA units is 2%. We obtain this estimate after synthesizing Verilog HDL descriptions of the baseline FMA unit and a modified FMA unit using a TSMC 45nm standard cell library.

For the architecture with a scalar unit and sliced datapath, we also include a comparison stage before the write-back stage in the SIMT pipeline. The overhead of this additional logic including the additional pipeline flip-flops is estimated by synthesizing its Verilog HDL descriptions using a TSMC 45nm standard cell library. The energy overhead of additional logic is summarized in Table 5 along with the energy consumption of the execution units [17]. The comparison stage has a dynamic energy overhead of 4.3% over the execution stage (8 FMA units).

All three proposed techniques collectively reduce the EU energy consumption. Composite instructions reduce the number of instructions issued to the EUs. The scalar unit reduces the dynamic energy consumption of SIMT EUs nearly a factor of 32 for redundant computations. The sliced architecture also reduces the dynamic energy of instructions with HB operands by nearly 50%. Overall, we reduce the EU energy by 41% on average.

6.3 GPU Power Consumption

Figure 11 shows the impact of our techniques on the total GPU power. The total GPU power consumption includes the dynamic power consumption of SMs, global memory power consumption, and the GPU leakage power. The leakage overhead of additional RFs and logic is also included in GPU power consumption. At the GPU level, we only increase the power consumption by less than 1% on average. However, this power increase comes with a significant performance improvement, resulting in much higher power efficiency than the baseline GPU. Our proposed techniques offer greater optimization opportunities (and thus higher performance) for INT benchmarks than for FP benchmarks, leading to more GPU power consumption (~1% and ~0.5% for INT and FP benchmarks, respectively).

6.4 GPU Power Efficiency

The primary goal of this paper is to improve the power efficiency of GPU architectures for GPGPU applications. The impact of our proposed techniques on power efficiency is shown in Figure 12. Power efficiency is determined by instruction per cycle (IPC) of the GPU divided by the total GPU power consumption (IPC/Watt). With small power overhead of our techniques, the performance improvements achieved improve the power efficiency of the GPU significantly. For INT and FP benchmarks, our techniques collectively improve power efficiency by 27% and 15%, respectively, on average; our techniques achieve 20% improvement in power efficiency across both INT and FP benchmarks.

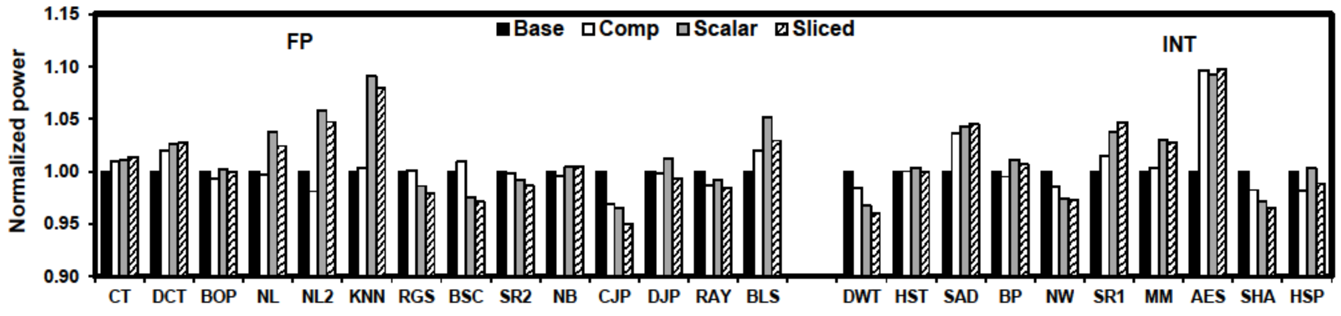

Figure 11: GPU power consumption.

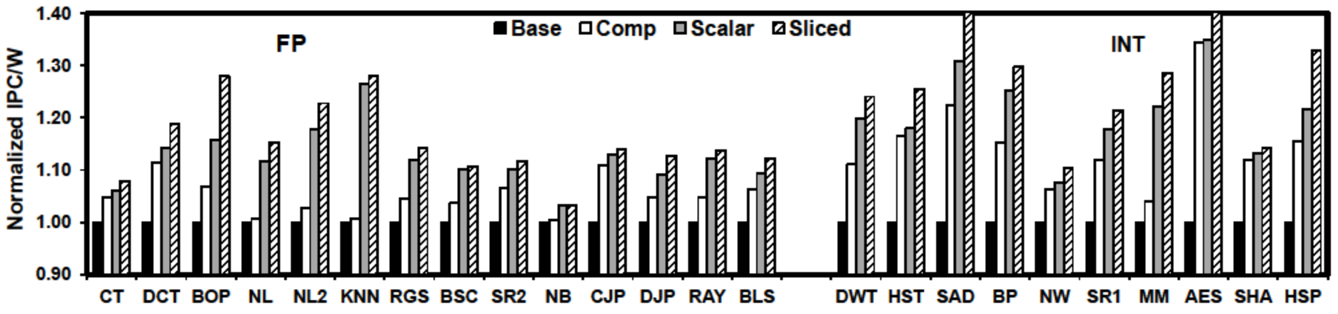

Figure 12: GPU IPC/Watt.

7. Discussion

Although our proposed techniques are evaluated using a GPU architecture similar to the NVIDIA FX5800 (GT200 architecture), they can be also employed with more recent GPU architectures such as Fermi and Kepler. Due to the lack of publically available performance simulators and power models for these newer architectures, we limit our evaluation to the GT200 architecture. However, any GPU architecture can adopt and benefit from our techniques with some modifications, because all of our techniques exploit common characteristics of GPGPU applications. Finally, all of our techniques utilize the dual-issue capability of the baseline GPU [8] to issue up to two instructions per cycle. Consequently, the evaluated GPU does not issue more than two instructions per cycle even if it has more than two instructions that can be issued at any given cycle with one or more of our techniques.

7.1 Application to Future GPU Architecture

In this section, we describe how our proposed approaches can benefit other GPU architectures such as Fermi and Kepler [18, 19].

Composite instructions: The primary motivation for employing composite instructions is to exploit common instruction patterns to reduce the number of fetched/executed instructions. We use the FMA pipeline to combine pairs of dependent INT instructions into composite instructions that can easily be mapped to the existing FMA resources at a negligible power overhead. Note that we do not proposed to implement a new datapath that is suitable for one or a few kernels evaluated in this paper. Instead, we exploit the existing FMA datapath to cover a large percentage of INT instruction patterns existing in many kernels.

The main benefit of composite instructions comes from the fact that the GPU can effectively execute two instructions whenever it issues a composite instruction. This allows the GPU to execute another instruction (which can be another composite instruction) in the next issue cycle and increase its effective IPC.

Since FMA units are implemented in most GPUs, composite instructions can be easily adopted by these GPUs. Supporting composite instructions for these GPUs requires a simple compiler pass that identifies appropriate dependent instruction pairs and replaces them with composite instructions. The replacement can either be done in the compiled code or by the GPU driver that converts the immediate representation in the compiled code to GPU's architecture-specific ISA.

Both Fermi and Kepler include separate INT and FP execution pipelines. For both architectures, the performance improvement with composite instructions is likely to be even higher compared to the GT200 architecture used for our evaluation. This is because normal INT instructions and composite instructions can be issued to the INT and FP pipelines in parallel.

Scalar unit: We employ a scalar unit to exploit the computational redundancy in kernels to improve performance. Computational redundancy is an architecture-independent characteristic and will exist in any GPU architecture. The extent of computational redundancy depends upon the application domains, the particular kernels, and the particular input data set. In order to take advantage of computational redundancy, we maintain a scalar bit with each SIMT register. If the scalar bits of all the operands of an SIMT instruction are set, the instruction is executed in the scalar pipeline. Otherwise, it is executed in the SIMT pipeline.

The performance improvement with the scalar unit is achieved by utilizing the dual-issue capability of the GPU to issue instructions to the SIMT and scalar pipelines simultaneously. The baseline GPU architecture schedules a warp over four cycles (i.e., 8-wide SIMT). A computationally redundant instruction, which is executed in the scalar pipeline, only requires a single cycle instead of four. Nonetheless, we do not exploit this to issue more scalar instructions in the remaining three cycles. This is done for two reasons. First, for the GT200 architecture, this increases the complexity of FDS logic. This is because, instead of scheduling warps every four cycles, the FDS logic should also be able to schedule new warps every cycle if they execute in the scalar pipeline. Second, we employ a single write-port SRF to store the results of computationally redundant instructions. For cases where both the SIMT and scalar pipelines need to write the result to the SRF, a single write port can be shared; the scalar and SIMT pipelines can use the write port in the first and fourth cycles, respectively, after the comparison stage detects the same result across all the SIMT lanes.

The Kepler architecture schedules a complete warp (32 threads) in a single cycle. The comparison stage (cf. Figure 6) can easily be extended to compare results across all 32 SIMT lanes. Since our technique only performs comparisons between adjacent lanes (using XOR gates), an increase in the SIMT width does not increase any critical paths for that stage; the all zero detection logic is made using dynamic gates and its delay does not increase with the increased SIMT width.

However, since the Kepler architecture issues the complete warp in a single cycle, for cases where both the SIMT and scalar pipelines need to write the result to the SRF an additional write port will be required in the SRF. However, the SRF has a very lower power impact and will not drastically change the power efficiency of the approach in Kepler.

Lastly, the performance advantage with the scalar pipeline is likely to be higher with Kepler. This is because the Kepler scheduler can schedule warps every cycle. This will allow the scalar pipeline to be utilized more efficiently, as new scalar instructions can be issued every cycle compared to the baseline GPU which can only schedule scalar instructions every four cycles. Assuming one issue per instruction, the Kepler architecture can issue 2 instructions per cycle (one scalar and one SIMT instruction per cycle) instead of 2 instructions per 4 cycles in the QuadroFX5600 GPU. This can allow the performance improvement of our proposed approach to increase by 4X.

Sliced datapath: Sliced datapath is motivated by the fact that many operands have sign-extended bits that do not efficiently utilize the register file bandwidth and execution resources of GPUs. We partition the 32-bit datapath into two 16-bit datapath slices. The dual-issue capability of the GPU is exploited to issue up to two 16-bit precision instructions to the 16-bit datapath slices. This approach requires the GPU to maintain to a MSB scalar bit per a warp register. The MSB scalar bit is computed during the computation of the scalar bit to exploiting computational redundancy. Thus, the increased SIMT width of Kepler does not impact the implementation complexity of this technique.

7.2 Impact on Critical Paths

The timing impact of all hardware requirements for the three techniques is evaluated after Verilog HDL descriptions are synthesized. A warp scheduler typically consists of scoreboarding flip-flops and round-robin schedulers and its compexity increases linearly with the number of warps. The total number of warps scheduled by the two warp schedulers in the sliced datapath approach is the same as the warp scheduler in the baseline GPU. Therefore, our techniques does not increase the complexity (i.e., critical path delay).

8. Related Work

Gebhart *et al.* propose to reduce the MRF accesses using register file caches and smaller and low power register file structures managed at compile time [12, 20]. Although we also include smaller register files (i.e., CRF and SRF) in our techniques, their purpose is not limited to reducing MRF accesses. They also allow us to considerably improve the performance of the GPU and also help reduce the energy consumption of the EUs and FDS logic.

Kim *et al.* propose a macro-op scheduling technique for out-of-order (OOO) processors to reduce the cycle time constraints from the scheduling logic [21]. Dependent INT instructions are dynamically detected and scheduled as a group. Each instruction within the macro-op is still issued separately. In contrast, our technique identifies pairs of dependent instructions to form composite instructions (and thus reduce the number of instructions at compile time. Moreover, these composite instructions are issued to the FMA units as a single instruction, reducing energy consumption of EUs and improving performance.

Brooks *et al.* propose an operation packing technique for OOO processors. They employ a sub-word parallel approach for improving the performance of reduced bit-width operands in OOO processors [22]. In contrast, our approach provides a sliced datapath architecture that can execute two sliced instructions from different threads simultaneously. The absence of data dependencies between different threads allows more opportunities for performance improvements. Moreover, the scheduling logic in an OOO processor is often timing critical [21]. Efficiently finding instructions that can be executed in a sub-word parallel form is likely to increase the critical path of the processor. On the other hand, GPUs schedule instructions at a much lower clock rate. A single scheduled warp issues over four cycles before the next warp is scheduled. This additional scheduling time allows GPUs to potentially make more complex scheduling decisions without impacting performance. Moreover, in our technique, we split the warp scheduler and MRF banks in our sliced datapath architecutre. This provides the opportunity to find independent instructions across threads instead only from the same thread (as done in [21]). Also, due to the much lower complexity of a GPU decoder (e.g., compared to x86 decoders), we can easily replicate it to issue two different instructions within the same scheduling cycle. Finally, the sliced datapath architecture can more efficiently utilize RF bandwidth, which is a critical resource in GPUs.

Hameed *et al.* utilize customizable processors to implement application-specific functional units to improve performance [23]. Our techniques focus on improving performance on general-purpose applications. Moreover, instead of adding customized functional units, we utilize the existing resources of the EUs to efficiently map common INT instructions. Ergin *et al.,* utilize the reduced bit-width operations to reduce register file pressure in OOO processors. Our techniques, however, strive to improve instruction throughput [24].

9. Conclusion

GPUs have traditionally improved their peak compute performance by integrating more compute resources and operating them at higher frequency. However, modern high performance GPUs are power-constrained and it is becoming more challenging to increase their compute resources or frequencies within their power budgets. Consequently, power efficient approaches need to be developed to improve performance for future power-constrained GPUs. Facing such a challenge, in this paper, we proposed three techniques that improved both power efficiency and performance of GPUs. We exploit the resources of the execution pipeline to execute composite instructions, reducing the power consumption of critical resources (FDS, MRF, and EUs) and significantly improving performance. We also exploit computational redundancy with a scalar unit, which provides additional power efficiency and performance improvements. Finally, a sliced datapath architecture is employed to take advantage of 16-bit precision operands common in many GPGPU applications. Individually, composite instructions, scalar units, and sliced datapaths provide geometric mean performance improvements of 9%, 7%, and 4%, respectively. Overall, our techniques offers geometric mean power efficiency and performance improvements by 20% and 15%, respectively.

Acknowledgement

This work is supported in part by NSF grants (CCF-095360, CCF-1016262, CNS-1217102), an SRC grant (Task ID: 2080.001), and generous gift grants from AMD.

References

[1] ITRS, 2011. [Online]. Available: http://www.itrs.net

[2] J. Lee, *et al.*, “Improving Throughput of Power-Constrained GPUs Using Dynamic Voltage/Frequency and Core Scaling,” in *ACM PACT*, 2011.

[3] S. Hong and H. Kim, “An integrated GPU power and performance model,” in *IEEE/ACM ISCA*, 2010.

[4] E. Lindholm, *et al.*, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro,* vol. 28, no. 2, pp. 39-55, 2008.

[5] Advanced Micro Devices, "Heterogeneous Computing -- OpenCL and the ATI Radeon ("Evergreen") architecture," [Online]. Available: http://developer.amd.com

[6] Advanced Micro Devices, 2008. [Online]. Available: http://developer.amd.com

[7] L. Seiler, *et al.*, “Larrabee: a many-core x86 architecture for visual computing,” *ACM Transactions on Graphics,* vol. 27, no. 3, pp. 18:1-18:15, 2008.

[8] H. Wong, *et al.*, "Demystifying GPU microarchitecture through microbenchmarking," in *IEEE ISPASS*, 2010.

[9] NVIDIA, "NVIDIA® CUDA SDK v2.3," [Online]. Available: http://developer.nvidia.com

[10] M. Mantor and M. Houston, "AMD Graphics Core Next," *AMD Fusion Developers Summit*, 2012.

[11] N. Weste and D. Harris, CMOS VLSI Design: A Circuits and Systems Perspective, Addison Wesley, 2010.

[12] M. Gebhart, *et al.*, "Energy-efficient mechanisms for managing thread context in throughput processors," in *IEEE/ACM ISCA*, 2011.

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

[14] L. Huang, *et al.*, “A New Architecture For Multiple-Precision Floating-Point Multiply-Add Fused Unit Design,” in *IEEE ARITH*, 2007.

[15] A. Bakhoda, *et al.*, "Analyzing CUDA workloads using a detailed GPU simulator," in *IEEE ISPASS*, 2009.

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

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

[18] NVIDIA, *NVIDIA's Next Generation CUDA(TM) compute Architecture: Fermi,* 2009.

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

[20] M. Gebhart, S. Keckler and W. Dally, “A compile-time managed multi-level register file hierarchy,” in *IEEE/ACM MICRO*, 2011.

[21] I. Kim and M. Lipasti, "Macro-op scheduling: relaxing scheduling loop constraints," in *IEEE/ACM MICRO*, 2003.

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

[23] R. Hameed, *et al.*, “Understanding sources of inefficiency in general-purpose chips,” in *IEEE/ACM ISCA*, 2010.

[24] O. Ergin, *et al.*, "Register Packing: Exploiting Narrow-Width Operands for Reducing Register File Pressure," in *IEEE/ACM MICRO*, 2004.

[25] D. Chang, *et al.* "ERCBench: An Open-Source Benchmark Suite for Embedded and Reconfigurable Computing," in *IEEE FPLA*, 2010.

[26] S. Che, *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE IISWC*, 2009.