Appears in the Proceedings of the 38^{th} International Symposium on Computer Architecture [‡]

Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors

Mark Gebhart[1†] Daniel R. Johnson[2†] David Tarjan[3]
mgebhart@cs.utexas.edu djohns53@illinois.edu dtarjan@nvidia.com
Stephen W. Keckler[1,3] William J. Dally[3,4] Erik Lindholm[3] Kevin Skadron[5]
skeckler@nvidia.com bdally@nvidia.com erikl@nvidia.com skadron@cs.virginia.edu

[1]The University of Texas at Austin
Austin, TX

[2]University of Illinois at Urbana-Champaign
Urbana, IL

[3]NVIDIA
Santa Clara, CA

[4]Stanford University
Stanford, CA

[5]University of Virginia
Charlottesville, VA

ABSTRACT

Modern graphics processing units (GPUs) use a large number of hardware threads to hide both function unit and memory access latency. Extreme multithreading requires a complicated thread scheduler as well as a large register file, which is expensive to access both in terms of energy and latency. We present two complementary techniques for reducing energy on massively-threaded processors such as GPUs. First, we examine register file caching to replace accesses to the large main register file with accesses to a smaller structure containing the immediate register working set of active threads. Second, we investigate a two-level thread scheduler that maintains a small set of active threads to hide ALU and local memory access latency and a larger set of pending threads to hide main memory latency. Combined with register file caching, a two-level thread scheduler provides a further reduction in energy by limiting the allocation of temporary register cache resources to only the currently active subset of threads. We show that on average, across a variety of real world graphics and compute workloads, a 6-entry per-thread register file cache reduces the number of reads and writes to the main register file by 50% and 59% respectively. We further show that the active thread count can be reduced by a factor of 4 with minimal impact on performance, resulting in a 36% reduction of register file energy.

Categories and Subject Descriptors: C.1.4 [Computer Systems Organization]: Processor Architectures – *Parallel Architectures*

General Terms: Experimentation, Measurement

ISCA'11, June 4–8, 2011, San Jose, California, USA.

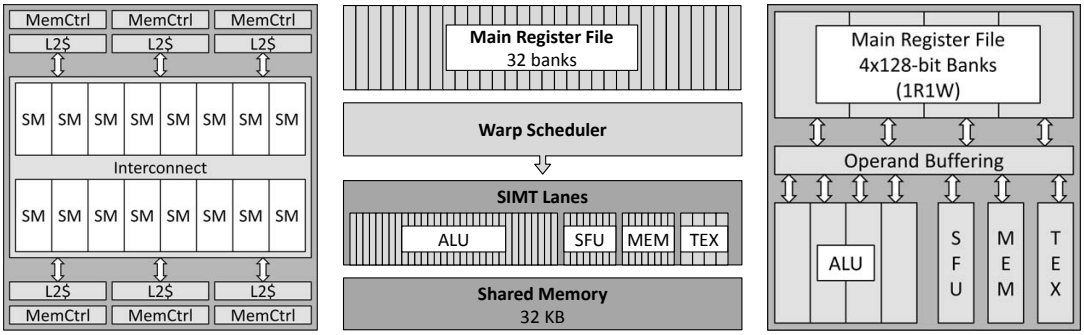
1. INTRODUCTION

Graphics processing units (GPUs) such as those produced by NVIDIA [28] and AMD [5] are massively parallel programmable processors originally designed to exploit the concurrency inherent in graphics workloads. Modern GPUs contain hundreds of arithmetic units, tens of thousands of hardware threads, and can achieve peak performance in excess of a teraFLOP. As single-thread performance improvements have slowed, GPUs have become more attractive targets for non-graphics workloads that demand high computational performance.

Unlike CPUs that generally target single-thread performance, GPUs aim for high throughput by employing extreme multithreading [13]. For example, NVIDIA's recent Fermi design has a capacity of over 20,000 threads interleaved across 512 processing units [28]. Just holding the register context of these threads requires substantial on-chip storage – 2 MB in total for the maximally configured Fermi chip. Further, extreme multithreaded architectures require a thread scheduler that can select a thread to execute each cycle from a large hardware-resident pool. Accessing large register files and scheduling among a large number of threads consumes precious energy that could otherwise be spent performing useful computation. As existing and future integrated systems become power limited, energy efficiency (even at a cost to area) is critical to system performance.

In this work, we investigate two techniques to improve the energy efficiency of the core datapath. *Register file caching* adds a small storage structure to capture the working set of registers, reducing the number of accesses to the larger main register file. While previous work has mainly examined register file caching to reduce latency for large CPU register files [11], this paper examines architectures and policies for register file caching aimed at reducing energy in throughput-oriented multithreaded architectures. Our analysis of both graphics and compute workloads on GPUs indicates sub-

[†]*First authors Gebhart and Johnson have made equal contributions to this work and are listed alphabetically.* *This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.*



(a) Full chip (b) Streaming multiprocessor (SM) (c) 4-Wide SIMT lane detail

Figure 1: Contemporary GPU architecture.

stantial register locality. A small, 6-entry per-thread register file cache reduces the number of reads and writes to the main register file by 50% and 43% respectively. Static liveness information can be used to elide writing dead values back from the register file cache to the main register file, resulting in a total reduction of write traffic to the main register file of 59%.

Multi-level scheduling partitions threads into two classes: (1) *active* threads that are issuing instructions or waiting on relatively short latency operations, and (2) *pending* threads that are waiting on long memory latencies. The cycle-by-cycle instruction scheduler need only consider the smaller number of active threads, enabling a simpler and more energy-efficient scheduler. Our results show that a factor of 4 fewer threads can be active without suffering a performance penalty. The combination of register file caching and multi-level scheduling enables a register file cache that is 21× smaller than the main register file, while capturing over half of all register accesses. This approach reduces the energy required for the register file by 36% compared to the baseline architecture without register file caching.

The remainder of this paper is organized as follows. Section 2 provides background on the design of contemporary GPUs and characterizes register value reuse in compute and graphics workloads. Section 3 describes our proposed microarchitectural enhancements, including both register file caching and scheduling techniques. Section 4 describes our evaluation methodology. Section 5 presents performance and power results. Sections 6 and 7 discuss related work, future work, and conclusions.

2. BACKGROUND

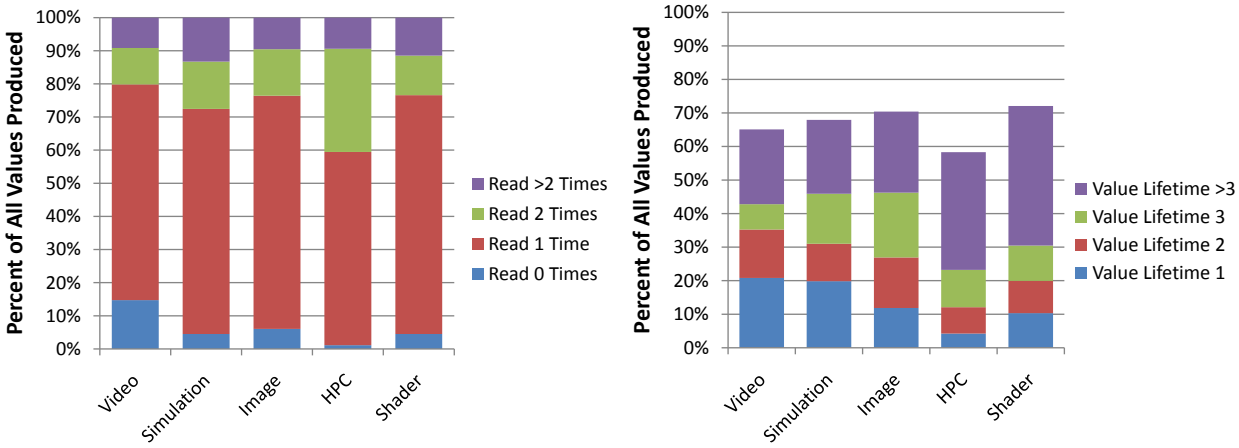
While GPUs are becoming increasingly popular targets for computationally-intensive non-graphics workloads, their design is primarily influenced by triangle-based raster graphics. Graphics workloads have a large amount of inherent parallelism that can be easily exploited by a parallel machine. Texture memory accesses are common operations in graphics workloads and tend to be fine-grained and difficult to prefetch. Graphics workloads have large, long-term (inter-frame) working sets that are not amenable to caching; therefore, texture cache units focus on conserving bandwidth rather than reducing latency [13]. Because texture accesses are macroscopically unpredictable, and frequent, GPUs rely on massive multithreading to keep arithmetic units utilized.

Figure 1 illustrates the architecture of a contemporary GPU, similar in nature to NVIDIA's Fermi design. The figure represents a generic design point similar to those discussed in the literature [6, 28, 35], but is not intended to correspond directly to any existing industrial product. The GPU consists of 16 streaming multiprocessors, 6 high-bandwidth DRAM channels, and an on-chip level-2 cache. A streaming multiprocessor (SM), shown in Figure 1(b), contains 32 SIMT (single-instruction, multiple thread) lanes that can collectively issue up to 32 instructions per cycle, one from each of 32 threads. Threads are organized into groups called *warps,* which execute together using a common physical program counter. While each thread has its own logical program counter and the hardware supports control-flow divergence of threads within a warp, the streaming multiprocessor executes most efficiently when all threads execute along a common control-flow path.

Fermi supports 48 active warps for a total of 1,536 active threads per SM. To accommodate this large set of threads, GPUs provide vast on-chip register file resources. Fermi provides 128KB of register file storage per streaming multiprocessor, allowing an average of 21 registers per thread at full scheduler occupancy. The total register file capacity across the chip is 2MB, substantially exceeding the size of the L2 cache. GPUs rely on heavily banked register files in order to provide high bandwidth with simple register file designs [27, 35]. Despite aggressive banking, these large register file resources not only consume area and static power, but result in high per-access energy due to their size and physical distance from execution units. Prior work examining a previous generation NVIDIA GTX280 GPU (which has 64 KB of register file storage per SM), estimates that nearly 10% of total GPU power is consumed by the register file [16]. Our own estimates show that the access and wire energy required to read an instruction's operands is twice that of actually performing a fused multiply-add [15]. Because power-supply voltage scaling has effectively come to an end [18], driving down per-instruction energy overheads will be the primary way to improve future processor performance.

2.1 Baseline SM Architecture

In this work, we focus on the design of the SM (Figures 1(b) and 1(c)). For our baseline, we model a contemporary GPU streaming multiprocessor with 32 SIMT lanes. Our baseline architecture supports a 32-entry warp scheduler, for a maximum of 1024 threads per SM, with a warp issuing a single instruction to each of the 32 lanes per cycle. We model single-issue, in-order pipelines for each lane. Each SM provides 32KB of local scratch storage known as *shared*



(a) Number of reads per register value.

(b) Lifetime of values that are read only once.

Figure 2: Value usage characterization.

memory. Figure 1(c) provides a more detailed microarchitectural illustration of a cluster of 4 SIMT lanes. A cluster is composed of 4 ALUs, 4 register banks, a special function unit (SFU), a memory unit (MEM), and a texture unit (TEX) shared between two clusters. Eight clusters form a complete 32-wide SM.

A single-precision fused multiply-add requires three register inputs and one register output per thread for a total register file bandwidth of 96 32-bit reads and 32 32-bit writes per cycle per SM. The SM achieves this bandwidth by subdividing the register file into multiple dual-ported banks (1 read and 1 write per cycle). Each entry in the SM's main register file (MRF) is 128 bits wide, with 32 bits allocated to the same-named register for threads in each of the 4 SIMT lanes in the cluster. Each bank contains 256 128-bit registers for a total of 4KB. The MRF consists of 32 banks for a total of 128KB per SM, allowing for an average of 32 registers per thread, more than Fermi's 21 per thread. The trend over the last several generations of GPUs has been to provision more registers per thread, and our traces make use of this larger register set.

The 128-bit registers are interleaved across the register file banks to increase the likelihood that all of the operands for an instruction can be fetched simultaneously. Instructions that require more than one register operand from the same bank perform their reads over multiple cycles, eliminating the possibility of a stall due to a bank conflict for a single instruction. Bank conflicts from instructions in different warps may occur when registers map to the same bank. Our MRF design is over-provisioned in bandwidth to reduce the effect of these rare conflicts. Bank conflicts can also be reduced significantly via the compiler [37]. The operand buffering between the MRF and the execution units represents interconnect and pipeline storage for operands that may be fetched from the MRF on different cycles.

2.2 GPU Value Usage Characterization

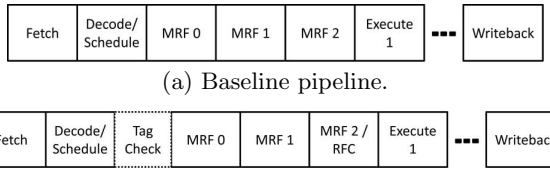
Prior work in the context of CPUs has shown that a large fraction of register values are consumed a small number of times, often within a few instructions of being produced [14]. Our analysis of GPU workloads indicates that the same trend holds. Figure 2(a) shows the number of times a value written to a register is read for a set of real-world graphics and compute workloads. Up to 70% of values are read only once, and only 10% of values are read more than 2 times. HPC workloads show the highest level of register value reuse with 40% of values being read more than once. Graphics workloads, labeled *Shader*, show reuse characteristics similar to the remaining compute traces. Figure 2(b) shows the lifetime of all dynamic values that are read only once. Lifetime is defined as the number of instructions between the producer and consumer (inclusive) in a thread. A value that is consumed directly after being produced has a lifetime of 1. Up to 40% of all dynamic values are read only once and are read within 3 instructions of being produced. In general, the HPC traces exhibit longer lifetimes than the other compute traces, due in part to hand-scheduled optimizations in several HPC codes where producers are hoisted significantly above consumers for improved memory level parallelism. Graphics traces also exhibit a larger proportion of values with longer lifetimes due to texture instructions, which the compiler hoists to improve performance. These value usage characteristics motivate the deployment of a register file cache to capture short-lived values and dramatically reduce accesses to the main register file.

3. ENERGY-EFFICIENT MULTI-THREADED MICROARCHITECTURES

This section details our microarchitectural extensions to the GPU streaming multiprocessor (SM) to improve energy efficiency, including register file caching and a simplified thread scheduler.

3.1 Register File Cache

Section 2.2 shows that up to 40% of all dynamic register values are read only once and within 3 instructions of being produced. Because these values have such short lifetimes, writing them into the main register file wastes energy. We propose a *register file cache* (RFC) to capture these short-lived values. The RFC filters requests to the main register file (MRF) and provides several benefits: (1) reduced MRF energy by reducing MRF accesses; (2) reduced operand delivery energy, since the RFC can be physically closer to the ALUs than the MRF; and (3) reduced MRF bandwidth requirements, allowing for a more energy-efficient MRF. The majority of this paper focuses on (1) while we discuss (2) and (3) in Section 5.4.



(a) Baseline pipeline.

(b) Pipeline with register file cache.

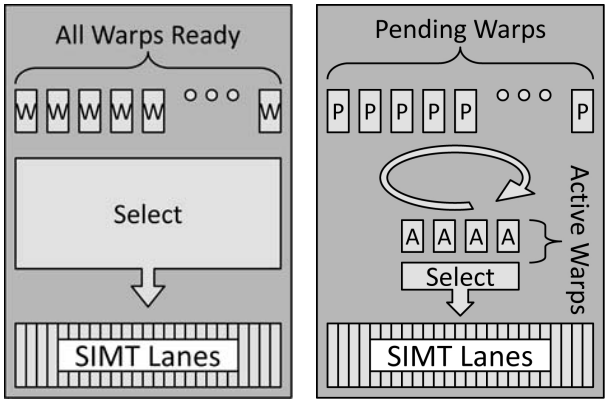
Figure 3: GPU pipelines.

Figure 3 highlights the pipeline modifications required to implement register file caching. Figure 3(a) shows a baseline pipeline with stages for fetch, decode/schedule, register file access (3 cycles to account for fetching and delivering multiple operands), one or more execute stages, and writeback. The pipeline with register file caching adds a stage to check the register file cache tags to determine if the operands are in the RFC. Operands not found in the RFC are fetched over multiple cycles from the MRF as before. Operands in the cache are fetched during the last stage allocated to MRF access. The RFC is multiported and all operands present in the RFC are read in a single cycle. We do not exploit the potential for reducing pipeline depth when all operands can be found in the RFC, as this optimization has a small effect on existing throughput-oriented architectures and workloads. The tag-check stage does not affect back-to-back instruction latencies, but adds a cycle to the branch resolution path. Our results show that this additional stage does not reduce performance noticeably, as branches do not dominate in the traces we evaluate.

RFC Allocation: Our baseline RFC design allocates the result of every operation into the RFC. We explored an extension that additionally allocates RFC entries for an instruction's source operands. We found that this policy results in 5% fewer MRF reads with a large RFC, but also pollutes the RFC, resulting in 10-20% more MRF writes. Such a policy requires additional RFC write ports, an expense not justified by our results.

RFC Replacement: Prior work on register file caches in the context of CPUs has used either LRU replacement [11] or a combination of FIFO and LRU replacement[36] to determine which value to evict when writing a new value into the RFC. While our baseline RFC design uses a FIFO replacement policy, our results show that using LRU replacement results in only an additional 1-2% reduction in MRF accesses. Compared to prior work on CPU register file caching, our RFC can only accommodate very few entries per thread due to the large thread count of a throughput processor, reducing the effectiveness of LRU replacement.

RFC Eviction: While the default policy writes all values evicted from the RFC to the MRF, many of these values will not actually be read again. In order to elide writebacks of dead values, we consider a combined hardware/software RFC design. We extend our hardware-only RFC design with compile-time generated static liveness information, which indicates the last instruction that will read a particular register instance. This information is passed to the hardware by an additional bit in the instruction encoding. Registers that have been read for the last time are marked dead in the RFC and their values need not be written back to the MRF. This optimization is conservative and never destroys valid data that could be used in the future. Due to uncertain control flow in the application, some values that are actually dead will be unnecessarily written back to the MRF.



(a) Single-level (b) Two-level

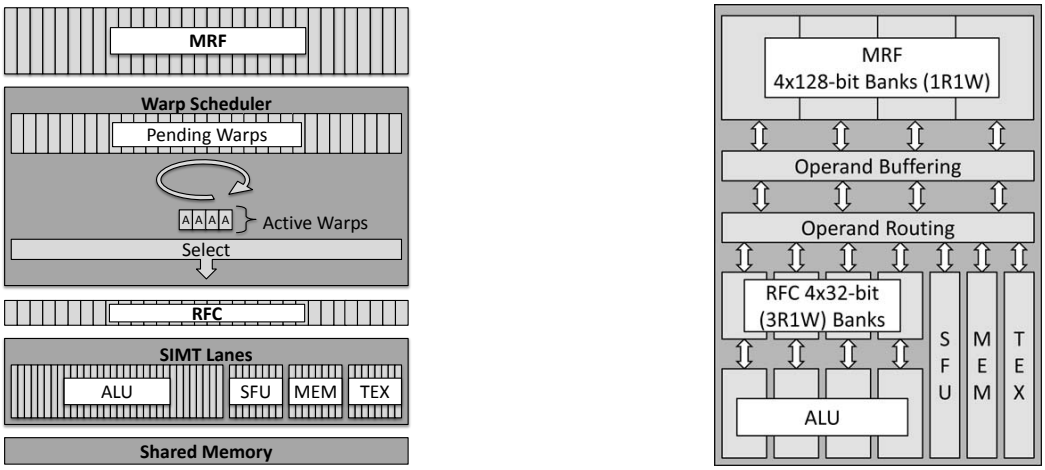
Figure 4: Warp schedulers.

Our results show that 6 RFC entries per thread captures most of the register locality and keeps the storage overhead per thread moderate. However, with 6 (4 byte) entries per thread, 32 threads per warp, and 32 warps per SM, the RFC would require 24 KB per SM. While this proposed baseline RFC is 5 times smaller than the MRF, its large size limits the potential energy savings.

3.2 Two-Level Warp Scheduler

To reduce the storage requirements of the RFC, we introduce a *two-level warp scheduler*. The warp scheduler, shown in Figure 4(a), is responsible for keeping the SIMT cores supplied with work in the face of both pipeline and memory latencies. To hide long latencies, GPUs allocate a large number of hardware thread contexts for each set of SIMT cores. This large set of concurrently executing warps in turn increases scheduler complexity, thus increasing area and power requirements. Significant state must be maintained for each warp in the scheduler, including buffered instructions for each warp. In addition, performing scheduling among such a large set of candidate warps necessitates complex selection logic and policies. The scheduler attempts to hide two distinct sources of latency in the system: (1) long, often unpredictable latencies, such as loads from DRAM or texture operations; and (2) shorter, often fixed or bounded latencies due to ALU operations, branch resolution, or accesses to the SM's local shared memory. A large pool of available warps is required to tolerate latencies in the first group, but a much smaller pool of warps is sufficient to tolerate common short latencies. The latency of arithmetic operations and shared memory accesses along with the amount of per-thread ILP influences the number of threads required to saturate the hardware. Reducing the set of warps available for selection on a given cycle can reduce both the complexity and energy overhead of the scheduler. One important consequence of reducing the number of concurrently active threads is that it reduces the immediate-term working set of registers.

We propose a *two-level warp scheduler* that partitions warps into an *active* set eligible for execution and an inactive *pending* set. The smaller set of active warps hides common short latencies, while the larger pool of pending warps is maintained to hide long latency operations and provide fast thread switching. Figure 4 illustrates a traditional single-level warp scheduler and our proposed two-level warp scheduler. All hardware-resident warps have entries in the outer level of the scheduler and are allocated MRF entries.



(a) High level SM architecture (b) Detailed microarchitecture

Figure 5: Modified GPU microarchitecture. (a) High level SM architecture: MRF with 32 128-bit wide banks, multiported RFC (3R/1W per lane). (b) Detailed SM microarchitecture: 4-lane cluster replicated 8 times to form 32 wide machine.

The outer scheduler contains a large set of entries where *pending* warps may wait on long latency operations to complete, with the number of pending entries required primarily influenced by the memory latency to be hidden. The inner level contains a much smaller set of *active* warps available for selection each cycle and is sized such that it can cover shorter latencies due to ALU operations, branch resolution, shared memory accesses, or cache hits. When a warp encounters a stall-inducing event, that warp can be removed from the active set but left pending in the outer scheduler. Introducing a second level to the scheduler presents a variety of new scheduling considerations for selection and replacement of warps from the active set.

Scheduling: For a two-level scheduler, we consider two common scheduling techniques: round-robin and greedy. For round-robin, we select a new ready warp from the active warp pool each cycle using a rotating priority. For greedy, we continue to issue instructions from a single active warp for as long as possible, without stalling, before selecting another ready warp. Our single-level scheduler has the same options, but all 32 warps remain selectable at all times. We evaluate the effectiveness of these policies in Section 5.

Replacement: A two-level scheduler must consider when to remove warps from the active set. Only warps which are ready or will be ready soon should be kept in the active set; otherwise, they should be replaced with ready warps to avoid stalls. Replacement can be done preemptively or reactively, and depending on the size of the active set and the latencies of key operations, different policies will be appropriate. We choose to suspend active warps when they consume a value produced by a long latency operation. Instructions marked by the compiler as sourcing an operand produced by a long-latency operation induce the warp to be suspended to the outer scheduler. We consider texture operations and global (cached) memory accesses as long-latency. This preemptive policy speculates that the value will not be ready immediately, a reasonable assumption on contemporary GPUs for both texture requests and loads that may access DRAM. Alternatively, a warp can be suspended after the number of cycles it is stalled exceeds some threshold; however, because long memory and texture latencies are common, we find this strategy reduces the effective size of the active warp set and sacrifices opportunities to execute instructions. For stalls on shorter latency computational operations or accesses to shared memory (local scratchpad), warps retain their active scheduler slot. For different design points, longer computational operations or shared memory accesses could be triggers for eviction from the active set.

3.3 Combined Architecture

While register file caching and two-level scheduling are each beneficial in isolation, combining them substantially increases the opportunity for energy savings. Figure 5(a) shows our proposed architecture that takes advantage of register file caching to reduce accesses to the MRF while employing a two-level warp scheduler to reduce the required size of an effective RFC. Figure 5(b) shows the detailed SM microarchitecture which places private RFC banks adjacent to each ALU. Instructions targeting the private ALUs are most common, so co-locating RFC banks with each ALU provides the greatest opportunity for energy reduction. Operands needed by the SFU, MEM, or TEX units are transmitted from the RFC banks using the operand routing switch.

To reduce the size of the RFC, entries are only allocated to *active* warps. Completed instructions write their results to the RFC according to the policies discussed in Section 3.1. When a warp encounters a dependence on a long latency operation, the two-level scheduler suspends the warp and evicts dirty RFC entries back to the MRF. To reduce writeback energy and avoid polluting the RFC, we augment the allocation policy described in Section 3.1 to bypass the results of long latency operations around the RFC, directly to the MRF. Allocating entries only for active warps and flushing the RFC when a warp is swapped out increases the number of MRF accesses but dramatically decreases the storage requirements of the RFC. Our results show that combining register file caching with two-level scheduling produces an RFC that (1) is 21 times smaller than the MRF, (2) eliminates more than half of the reads and writes to the MRF, (3) has negligible impact on performance, and (4) reduces register file energy by 36%.

Category	Examples	Traces	Avg. Dynamic Warp Insts.	Avg. Threads
Video Processing	H264 Encoder, Video Enhancement	19	60 million	99K
Simulation	Molecular Dynamics, Computational Graphics, Path Finding	11	691 million	415K
Image Processing	Image Blur, JPEG	7	49 million	329K
HPC	DGEMM, SGEMM, FFT	18	44 million	129K
Shader	12 Modern Video Games	155	5 million	13K

Table 1: Trace characteristics.

RFC Entries per Thread	Active Warps					
	4		6		8	
	μ^2	R/W (pJ)	μ^2	R/W (pJ)	μ^2	R/W (pJ)
4	5100	1.2/3.8	7400	1.2/4.4	9600	1.9/6.1
6	7400	1.2/4.4	10800	1.7/5.4	14300	2.2/6.7
8	9600	1.9/6.1	14300	2.2/6.7	18800	3.4/10.9

Table 2: RFC area and read/write energy for 128-bit accesses.

Parameter	Value
MRF Read/Write Energy	8/11 pJ
MRF Bank Area	38000 μ^2
MRF Distance to ALUs	1 mm
Wire capacitance	300 fF/mm
Voltage	0.9 Volts
Wire Energy (32 bits)	1.9 pJ/mm

Table 3: Modeling parameters.

Parameter	Value
Execution Model	In-order
Execution Width	32 wide SIMT
Register File Capacity	128 KB
Register Bank Capacity	4 KB
Shared Memory Capacity	32 KB
Shared Memory Bandwidth	32 bytes / cycle
SM External Memory Bandwidth	32 bytes / cycle
ALU Latency	8 cycles
Special Function Latency	20 cycles
Shared Memory Latency	20 cycles
Texture Instruction Latency	400 cycles
DRAM Latency	400 cycles

Table 4: Simulation parameters.

4. METHODOLOGY

As described in Section 2.1, we model a contemporary GPU SIMT processor, similar in structure to the NVIDIA Fermi streaming multiprocessor (SM). Table 4 summarizes the simulation parameters used for our SM design. Standard integer ALU and single-precision floating-point operations have a latency of 8-cycles and operate with full throughput across all lanes. While contemporary NVIDIA GPUs have longer pipeline latencies for standard operations [35], 8 cycles is a reasonable assumption based on AMD's GPUs [4]. As with modern GPUs, various shared units operate with a throughput of less than the full SM SIMT width. Our texture unit has a throughput of four texture (`TEX`) instructions per cycle. Special operations, such as transcendental functions, operate with an aggregate throughput of 8 operations per cycle.

Due to the memory access characteristics and programming style of the workloads we investigate, we find that system throughput is relatively insensitive to cache hit rates and typical DRAM access latency. Codes make heavy use of shared memory or texture for memory accesses, using most DRAM accesses to populate the local scratchpad memory. Combined with the large available hardware thread count, the relatively meager caches provided by modern GPUs only minimally alter performance results, especially for shader workloads. We find the performance difference between no caches and perfect caches to be less than 10% for our workloads, so we model the memory system as bandwidth constrained with a fixed latency of 400 cycles.

4.1 Workloads

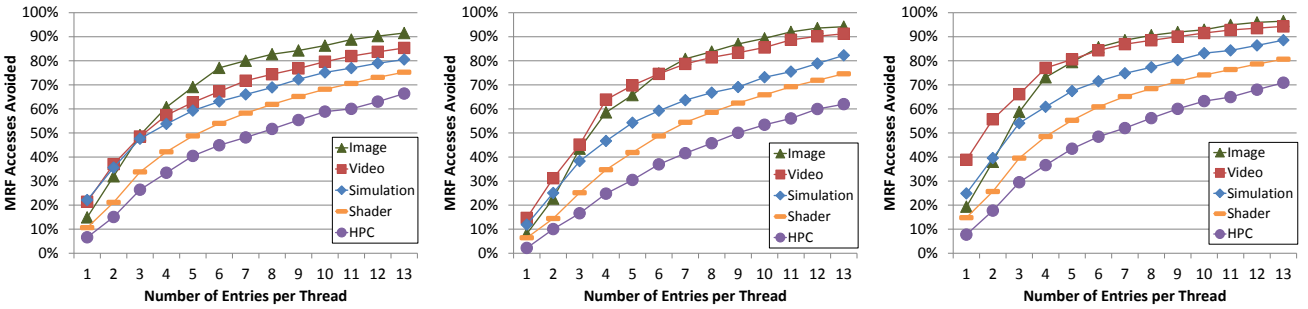
We evaluate 210 real world instruction traces, described in Table 1, taken from a variety of sources. The traces are encoded in NVIDIA's native ISA. Due to the large number of traces we evaluate, we present the majority of our results as category averages. 55 of the traces come from compute workloads, including high-performance and scientific computing, image and video processing, and simulation. The remaining 155 traces represent important shaders from 12 popular games published in the last 5 years. Shaders are short programs that perform programmable rendering operations, usually on a per-pixel or per-vertex basis, and operate across very large datasets with millions of threads per frame.

4.2 Simulation Methodology

We employ a custom trace-based simulator that models the SM pipeline and memory system described in Sections 3 and 4. When evaluating register file caching, we simulate all threads in each trace. For two-level scheduling, we simulate execution time on a single SM for a subset of the total threads available for each workload, selected in proportion to occurrence in the overall workload. This strategy reduces simulation time while still accurately representing the behavior of the trace.

4.3 Energy Model

We model the energy requirements of several 3-read port, 1-write port RFC configurations using synthesized flip-flop arrays. We use Synopsys Design Compiler with both clock-gating and power optimizations enabled and commercial 40 nm high-performance standard cell libraries with a clock target of 1GHz at 0.9V. We estimate access energy by performing several thousand reads and writes of uniform random data across all ports. Table 2 shows the RFC read and write energy for four 32-bit values, equivalent to one 128-bit MRF



(a) Read accesses avoided (b) Write accesses avoided (c) Write accesses avoided (SW liveness)

Figure 6: Reduction of MRF accesses by baseline register file cache.

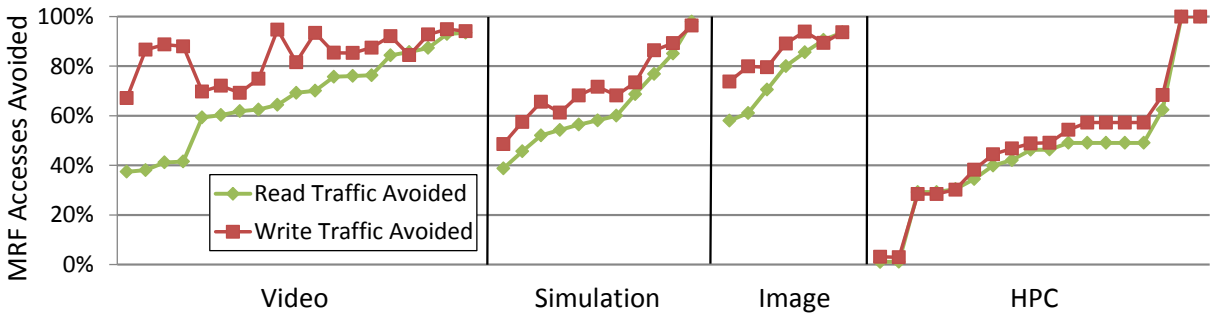


Figure 7: Per-trace reduction in MRF accesses with a 6 entry RFC per thread (one point per trace).

entry. We model the main register file (MRF) as a collection of 32 4KB, 128-bit wide dual-ported (1 read, 1 write) SRAM banks. SRAMs are generated using a commercial memory compiler and are characterized similarly to the RFC for read and write energy at 1GHz.

We model wire energy based on the methodology of [20] using the parameters listed in Table 3, resulting in energy consumption of 1.9pJ per mm for a 32-bit word. From a Fermi die photo, we estimate the area of a single SM to be 16 mm^2 and assume that operands must travel 1 mm from a MRF bank to the ALUs. Each RFC bank is private to a SIMT lane, greatly reducing distance from the RFC banks to the ALUs. The tags for the RFC are located close to the scheduler to minimize the energy spent accessing them. Section 5.4 evaluates the impact of wire energy. Overall, we found our energy measurements to be consistent with previous studies [8] and CACTI [24] after accounting for differences in design space and process technology.

5. EVALUATION

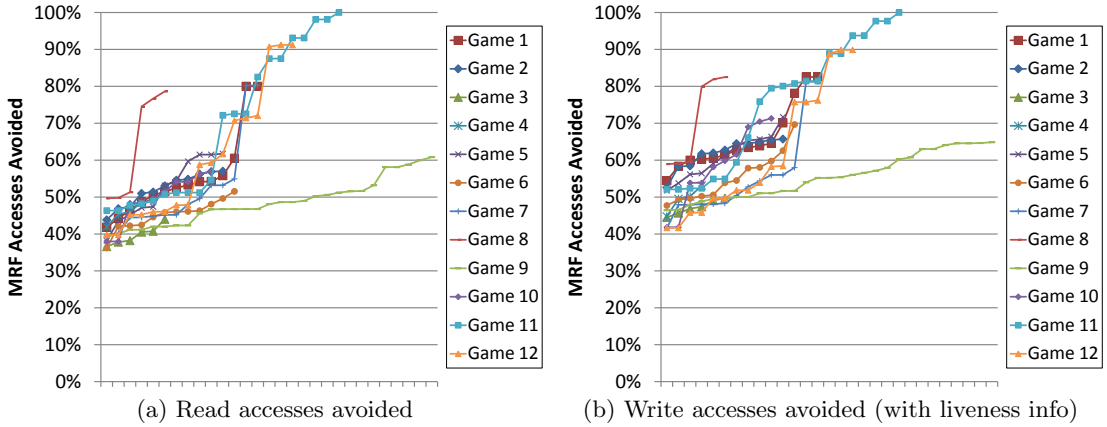
This section demonstrates the effectiveness of register file caching and two-level scheduling on GPU compute and graphics workloads. We first evaluate the effectiveness of each technique individually and then show how the combination reduces overall register file energy. As power consumption characteristics are specific to particular technology and implementation choices, we first present our results in a technology-independent metric (fraction of MRF reads and writes avoided), and then present energy estimates for our chosen design points.

5.1 Baseline Register File Cache

Figures 6(a) and 6(b) show the percentage of MRF read and write traffic that can be avoided by the addition of the baseline RFC described in Section 3.1. Even a single-entry RFC reduces MRF reads and writes, with the knee of the curve at about 6 entries for each per-thread RFC. At 6 RFC entries, this simple mechanism filters 45-75% of MRF reads and 35-75% of MRF writes. RFC effectiveness is lowest on HPC traces, where register values are reused more frequently and have longer average lifetimes, a result of hand scheduling.

As discussed in Section 2.2, many register values are only read a single time. Figure 6(c) shows the percentage of MRF writes avoided when static liveness information is used to identify the last consumer of a register value and avoid writing the value back to the MRF on eviction from the RFC. Read traffic does not change, as liveness information is used only to avoid writing back dead values. With 6 RFC entries per thread, the use of liveness information increases the fraction of MRF accesses avoided by 10–15%. We present the remaining write traffic results assuming static liveness information is used to avoid dead value writebacks.

Figure 7 plots the reduction in MRF traffic with a 6-entry RFC for each individual compute trace. For these graphs, each point on the x-axis represents a different trace from one of the sets of compute applications. The traces are sorted on the x-axis by the amount of read traffic avoided. The lines connecting the points serve only to clarify the two categories and do not imply a parameterized data series. The effectiveness of the RFC is a function of both the inherent data reuse in the algorithms and the compiler generated schedule in the trace. Some optimizations such as hoisting improve performance at the expense of reducing the effectiveness of the RFC. All of the traces, except for a few hand-scheduled HPC codes, were scheduled by a production NVIDIA compiler that does not optimize for our proposed register file



(a) Read accesses avoided (b) Write accesses avoided (with liveness info)

Figure 8: Graphics per-trace reduction in MRF accesses with a 6 entry RFC per thread (one point per trace).

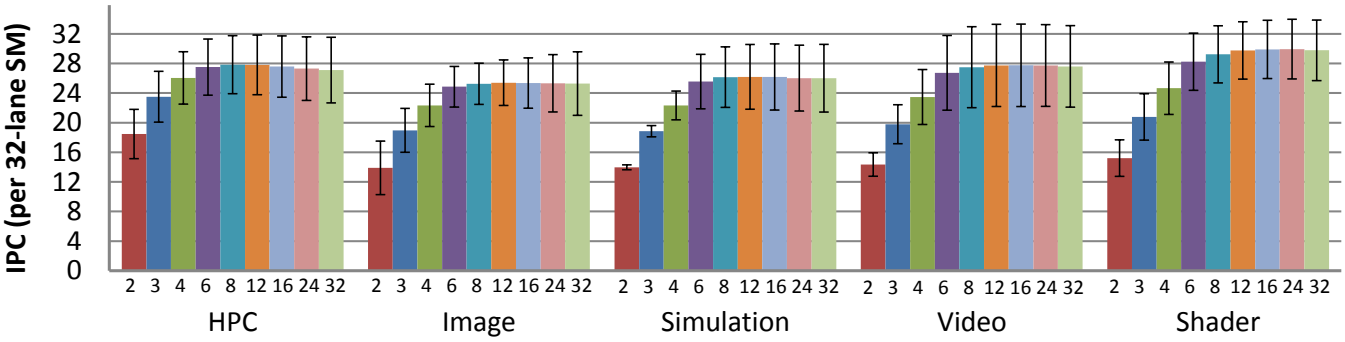


Figure 9: Average IPC with +/-1 standard deviation for a range of active warps.

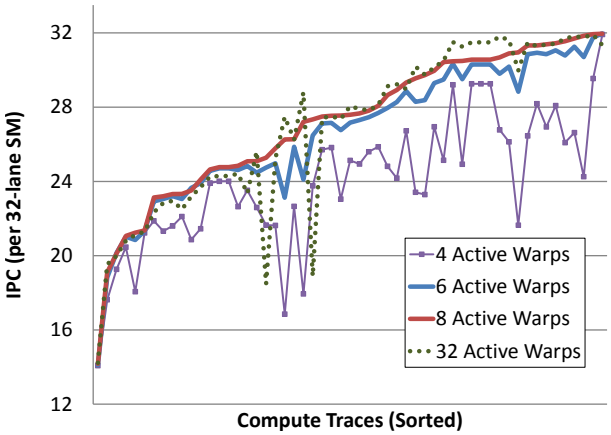
cache. To provide insight into shader behavior, Figure 8 shows results for a 6-entry per thread cache for individual shader traces grouped by games and sorted on the x-axis by MRF accesses avoided. Due to the large number of traces, individual datapoints are hard to observe, but the graphs demonstrate variability both within and across each game. Across all shaders, a minimum of 35% of reads and 40% of writes are avoided, illustrating the general effectiveness of this technique for these workloads. While the remainder of our results are presented as averages, the same general trends appear for other RFC configurations.

5.2 Two-Level Warp Scheduler

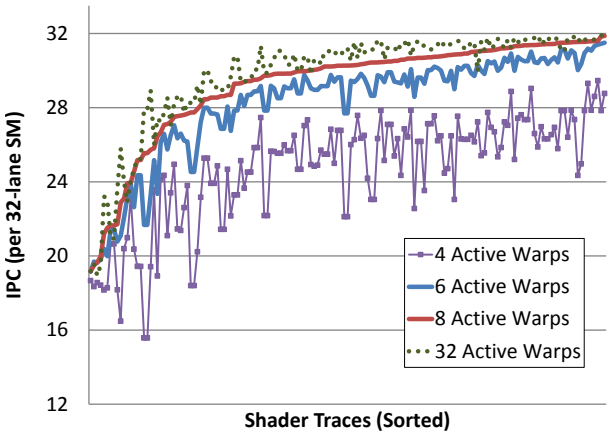
Next, we consider the performance of our two-level warp scheduler rather than the typical, more complex, single level scheduler. Figure 9 shows SM instructions per clock (IPC) for a scheduler with 32 total warps and a range of active warps, denoted below each bar. Along with the arithmetic mean, the graph shows standard deviation across traces for each scheduler size. The scheduler uses a greedy policy in the inner level, issuing from a single warp until it can no longer issue without a stall, and uses a round-robin policy when replacing active warps with ready pending warps from the outer level. The single-level scheduler (all 32 warps active) issues in the same greedy fashion as the inner level. A two-level scheduler with 8 active warps achieves nearly identical performance to a scheduler with all 32 warps active, while scheduling 6 active warps experiences a 1% performance loss on compute and a 5% loss on graphics shaders.

Figure 10 shows a breakdown of both compute and shader traces for a few key active scheduler sizes. The figure shows an all-warps-active scheduler along with three smaller active scheduler sizes. A system with 8 active warps achieves nearly the same performance as a single-level warp scheduler, whereas performance begins to deteriorate significantly with fewer than 6 active warps. The effectiveness of 6 to 8 active warps can be attributed in part to our pipeline parameters; an 8-cycle pipeline latency is completely hidden with 8 warps, while a modest amount of ILP allows 6 to perform nearly as well. Some traces actually see higher performance with fewer active warps when compared with a fully active warp scheduler; selecting among a smaller set of warps until a long latency stall occurs helps to spread out long latency memory or texture operations in time.

For selection among active warps, we compared round-robin and greedy policies. Round-robin performs worse as the active thread pool size is increased beyond a certain size. This effect occurs because a fine-grained round-robin interleaving tends to expose long-latency operations across multiple warps in a short window of time, leading to many simultaneously stalled warps. For the SPMD code common to GPUs, round-robin scheduling of active warps leads to consuming easily extracted parallel math operations without overlapping memory accesses across warps. On the other hand, issuing greedily often allows a stall-inducing long latency operation (memory or texture) in one warp to be discovered before switching to a new warp, overlapping the latency with other computation.

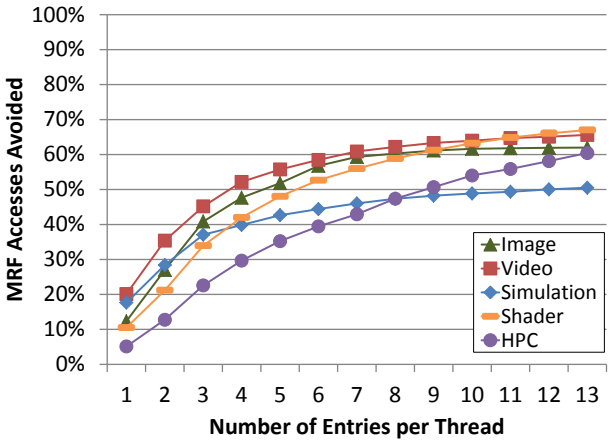


(a) 2-Level scheduling: compute

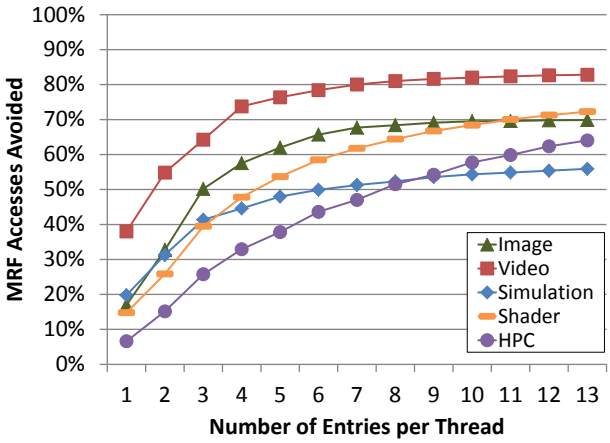


(b) 2-Level scheduling: shader

Figure 10: IPC for various active warp set sizes (one point per trace).



(a) MRF read accesses avoided



(b) MRF write accesses avoided (SW liveness)

Figure 11: Effectiveness of RFC when combined with two-level scheduler.

5.3 Combined Architecture

Combining register file caching with two-level scheduling produces an effective combination for reducing accesses to a large register file structure. A two-level scheduler dramatically reduces the size of the RFC by allocating entries only to active warps, while still maintaining performance comparable to a single-level scheduler. A consequence of two-level scheduling is that when a warp is deactivated, its entries in the RFC must be flushed to the MRF so that RFC resources can be reallocated to a newly activated warp.

Figure 11 shows the effectiveness of the RFC in combination with a two-level scheduler as a function of RFC entries per thread. Compared to the results shown in Figure 6, flushing the RFC entries for suspended warps increases the number of MRF accesses by roughly 10%. This reduction in RFC effectiveness is more than justified by the substantial (4-6×) reduction in RFC capacity requirements when allocating only for active warps.

We explore extending our baseline design by using static liveness information to bypass values that will not be read before the warp is deactivated around the RFC, directly to the MRF. Additionally, we use static liveness information to augment the FIFO RFC replacement policy to first evict RFC values that will not be read before the next long latency operation. These optimizations provide a modest 1-2% reduction in MRF accesses. However, bypassing these values around the RFC saves energy by avoiding RFC accesses and reduces the number of RFC writebacks to the MRF by 30%.

5.4 Energy Savings

MRF Traffic Reduction: Figure 12 shows the energy consumed in register file accesses for a range of RFC configurations, normalized to the baseline design without an RFC. Each bar is annotated with the amount of storage required for the RFC bank per SIMT lane. The RFC architectures include two-level scheduling with 4, 6, or 8 active warps and 4, 6, or 8 RFC entries per thread. The energy for the RFC configurations is based on the results in Section 5.3 and include accesses to the RFC, the MRF, and the RFC/MRF accesses required to flush RFC entries on a thread swap. The addition of a register file cache substantially reduces energy consumption, 20% to 40% for a variety of design points. Generally, an RFC with 6 entries per thread provides the most energy savings for our SM design point. A design with 6 RFC entries per thread and 8 active warps reduces the amount of energy spent on register file accesses by 25% without a performance penalty. This design requires 192 bytes of storage for the RFC bank per SIMT lane for a

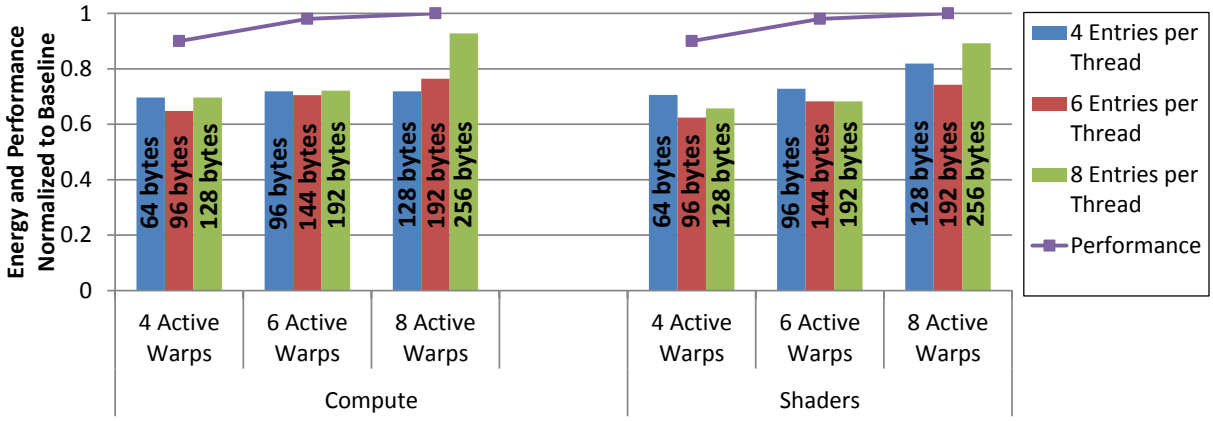


Figure 12: Energy savings due to MRF traffic reduction, bars show register file access energy consumed relative to baseline without RFC (lower is better), lines show performance (higher is better).

total of 6,144 bytes per SM. Additional energy can be saved at the expense of performance for some workloads. In the 8-active, 6-entry configuration, if every access was to the RFC and the MRF was never accessed, the maximum idealized energy savings would be 58%.

Wire Energy: Our proposed design presents several opportunities to reduce the energy expended moving data between the MRF and the ALUs. MRF banks are shared across 4 SIMT lanes, forcing operands to be distributed a greater distance to reach the ALUs compared with the per-lane RFC banks. The operand buffering used to enable multi-cycle operand collection represents a large piece of logic and interconnect that takes significant energy to traverse. Operands from the RFC are fetched in a single cycle, avoiding multi-cycle operand buffering. Further, the RFC banks can be located much closer to the ALUs, reducing wiring energy for ALU operations, while not adversely affecting the wiring energy to the shared units. To evaluate the potential energy savings from operand delivery we focus on the expected reduction in wire distance using the energy parameters in Table 3. While Figure 12 assumes zero wiring overhead for either the MRF or RFC, the bottom three rows of Table 5 show the normalized register file energy savings when accounting for both the reduction in MRF accesses and various wiring distances between the RFC and ALUs, with 8 active warps and 6 RFC entries per thread. Assuming the MRF is 1mm from the ALUs, locating the RFCs 0.2mm from the ALUs boosts energy savings to 35% for the compute traces and 37% for the shader traces. We expect additional savings to come from the reduction in multi-cycle operand buffering and multiplexing of values coming from the MRF, an effect not quantified here.

5.5 Discussion

To put our energy savings in context, we present a high-level GPU power model. A modern GPU chip consumes roughly 130 Watts [16]. If we assume that 1/3 of this power is spent on leakage, the chip consumes 87 Watts of dynamic power. We assume that 30% of dynamic power is consumed by the memory system and 70% of dynamic power is consumed by the SMs [23]. This gives a dynamic power of 3.8 Watts per SM for a chip with 16 SMs. The register file conservatively consumes 15% of the SM power [16], about 0.6 Watts per SM. The register file system power is split between bank access power, wire transfer power, and operand routing and buffering power. Our detailed evaluation shows that our technique saves 36% of register file access and wire energy. A detailed evaluation of operand buffering and routing power is beyond the scope of this paper. For the purpose of this high-level model, we assume that our technique saves 36% of the full register file system power, about 0.2 Watts per SM, for a chip wide savings of 3.3 Watts. This represents 5.4% of SM power and 3.8% of chip-wide dynamic power. While 3.3 Watts may appear to be a small portion of overall chip power, today's GPUs are power limited and improvements in energy efficiency can be directly translated to performance increases. Making future high-performance integrated systems more energy-efficient will come from a series of architectural improvements rather than from a single magic bullet. While each improvement may have a small effect in isolation, collectively they can be significant.

In addition to the energy savings from simplifying the frequently traversed datapaths from operand storage to ALUs, the RFC and two-level scheduler provide two additional opportunities for energy savings. First, an RFC with 6 entries per thread reduces the average MRF bandwidth required per instruction by half. We expect that this effect can be leveraged to build a more energy-efficient MRF with less aggregate bandwidth, without sacrificing performance. Second, the two-level scheduler greatly simplifies scheduling logic relative to a complex all-warps-active scheduler, an energy-intensive component of the SM that must make time-critical selections among a large number of warps. By reducing the number of active warps, a two-level scheduler also reduces storage requirements for buffered instructions and scheduling state. Additionally, reduced ALU latencies can decrease the average short-term latency that the inner scheduler must hide, allowing fewer active warps to maintain throughput, further reducing RFC overhead. ALU latency of 4 cycles increases IPC for a 4-active scheduler by 5%, with smaller increases for 6 and 8. Finally, prior work on a previous generation GPU has shown that instruction fetch, decode, and scheduling consumes up to 20% of chip-level power [16], making the scheduler an attractive target for energy reduction.

		Normalized Register File Energy	
MRF to ALU (mm)	RFC to ALU (mm)	Compute	Shaders
0	0	0.76	0.74
1	1	0.87	0.86
1	0.2	0.65	0.63
1	0	0.63	0.60

Table 5: Combined access and wire energy savings for 8-active scheduler, 6-entry RFC.

6. RELATED WORK

Previous work on ILP-oriented superscalar schedulers has proposed holding instructions dependent on long-latency operations in separate waiting instruction buffers [22] for tolerating cache misses and using segmented [29] or hierarchical [10] windows. Cyclone proposed a two-level scheduling queue for superscalar architectures to tolerate events of different latencies [12].

A variety of multithreaded architectures have been proposed in the literature [34]. The Tera MTA [2] provided large numbers of hardware thread contexts, similar to GPUs, for latency hiding. Multithreaded machines such as Sun's Niagara [21] select among smaller numbers of threads to hide latencies and avoid stalls. AMD GPUs [3, 5] multiplex a small number of very wide (64 thread) wavefronts issued over several cycles. Tune proposed balanced multithreading, an extension to SMT where additional virtual thread contexts are presented to software to leverage memory-level parallelism, while fewer hardware thread contexts simplify the SMT pipeline implementation [33]. Intel's Larabee [30] proposed software mechanisms, similar to traditional software context switching, for suspending threads expected to become idle due to texture requests. MIT's Alewife [1] used coarse-grained multithreading, performing thread context switches only when a thread relied upon a remote memory access. Mechanisms for efficient context switching have been proposed which recognize that only a subset of values are live across context switches [25].

Prior work has found that a large number of register values are only used once and within a short period of time from when they are produced [14]. Swensen and Patt show that a 2-level register file hierarchy can provided nearly all of the performance benefit of a large register file on scientific codes [32]. Prior work has examined using register file caches in the context of CPUs [11, 36, 19, 26, 9, 7], much of which was focused on reducing the latency of register file accesses. Rather than reduce latency, we aim to reduce the energy spent in the register file system. Shioya et al. designed a register cache for a limited-thread CPU that aims to simplify the design of the MRF to save area and energy rather than reducing latency [31]. Other work in the context of CPUs considers register file caches with tens of entries per thread [11]. Since GPUs have a large number of threads, the register file cache must have a limited number of entries per thread to remain small. Zeng and Ghose propose a register file cache that saves 38% of the register file access energy in a CPU by reducing the number of ports required in the main register file [36]. Each thread on a GPU is executed in-order, removing several of the challenges faced by register file caching on a CPU, including preserving register values for precise exceptions [17] and the interaction between register renaming and register file caching. The ELM project considered a software controlled operand register file, private to an ALU, with a small number of entries to reduce energy on an embedded processor [8]. In ELM, entries were not shared among a large number of threads and could be persistent for extended periods of time. Past work has also relied on software providing information to the hardware to increase the effectiveness of the register file cache [19].

AMD GPUs use clause temporary registers to hold short lived values during a short sequence of instructions that does not contain a change in control-flow [5]. The software explicitly controls allocation and eviction from these registers and values are not preserved across clause boundaries. Additionally, instructions can explicitly use the result of the previous instruction [3]. This technique can eliminate register accesses for values that are consumed only once when the consuming instruction immediately follows the producing instruction.

7. CONCLUSION

Modern GPU designs are constrained by various requirements, chief among which is high performance on throughput-oriented workloads such as traditional graphics applications. This requirement leads to design points which achieve performance through massive multithreading, necessitating very large register files. We demonstrate that a combination of two techniques, register file caching and two-level scheduling, can provide power savings while maintaining performance for massively threaded GPU designs. Two-level scheduling reduces the hardware required for warp scheduling and enables smaller, more efficient register file caches. Coupled with two-level scheduling, register file caching reduces traffic to the main register file by 40-80% with 6 RFC entries per thread. This reduction in traffic along with more efficient operand delivery reduces energy consumed by the register file by 36%, corresponding to a savings of 3.3 Watts of total chip power, without affecting throughput.

Several opportunities exist to extend this work to further improve energy efficiency. Rather than rely on a hardware register cache, allocation and eviction could be handled entirely by software, eliminating the need for RFC tag checks. The effectiveness of the RFC could be improved by applying code transformations that were aware of the RFC. Software control in the form of hints could also be applied to provide more flexible policies for activating or deactivating warps in our two-level scheduler. As GPUs have emerged as an important high performance computational platform, improving energy efficiency through all possible means including static analysis will be key to future increases in performance.

Acknowledgments

We thank the anonymous reviewers and the members of the NVIDIA Architecture Research Group for their comments. This research was funded in part by DARPA contract HR0011-10-9-0008 and NSF grant CCF-0936700.

8. REFERENCES

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *International Symposium on Computer Architecture*, pages 104–114, June 1990.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[3] AMD. R600-Family Instruction Set Architecture. `http://developer.amd.com/gpu_assets/R600_Instruction_Set_Architecture.pdf`, January 2009.

[4] AMD. ATI Stream Computing OpenCL Programming Guide. `http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf`, August 2010.

[5] AMD. HD 6900 Series Instruction Set Architecture. `http://developer.amd.com/gpu/amdappsdk/assets/AMD_HD_6900_Series_Instruction_Set_Architecture.pdf`, February 2011.

[6] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *International Symposium on Performance Analysis of Systems and Software*, pages 163–174, April 2009.

[7] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Reducing the Complexity of the Register File in Dynamic Susperscalar Processors. In *International Symposium on Microarchitecture*, pages 237–248, December 2001.

[8] J. Balfour, R. Harting, and W. Dally. Operand Registers and Explicit Operand Forwarding. *IEEE Computer Archiecture Letters*, 8(2):60–63, July 2009.

[9] E. Borch, E. Tune, S. Manne, and J. Emer. Loose Loops Sink Chips. In *International Symposium on High Performance Computer Architecture*, pages 299–310, February 2002.

[10] E. Brekelbaum, J. Rupley, C. Wilkerson, and B. Black. Hierarchical Scheduling Windows. In *International Symposium on Microarchitecture*, pages 27–36, November 2002.

[11] J. Cruz, A. Gonzlez, M. Valero, and N. P. Topham. Multiple-banked Register File Architectures. In *International Symposium on Computer Architecture*, pages 316–325, June 2000.

[12] D. Ernst, A. Hamel, and T. Austin. Cyclone: A Broadcast-free Dynamic Instrution Scheduler with Selective Replay. In *International Symposium on Computer Architecture*, pages 253–263, June 2003.

[13] K. Fatahalian and M. Houston. A Closer Look at GPUs. *Communications of the ACM*, 51(10):50–57, October 2008.

[14] M. Franklin and G. S. Sohi. Register Traffic Analysis for Streamlining Inter-operation Communication in Fine-grain Parallel Processors. In *International Symposium on Microarchitecture*, pages 236–245, November 1992.

[15] S. Galal and M. Horowitz. Energy-Efficient Floating Point Unit Design. *IEEE Transactions on Computers*, 99(PrePrint), 2010.

[16] S. Hong and H. Kim. An Integrated GPU Power and Performance Model. In *International Symposium on Computer Architecture*, pages 280–289, June 2010.

[17] Z. Hu and M. Martonosi. Reducing Register File Power Consumption by Exploiting Value Lifetime Characteristics. In *Workshop on Complexity-Effective Design*, June 2000.

[18] International Technology Roadmap for Semiconductors. `http://itrs.net/links/2009ITRS/Home2009.htm`, 2009.

[19] T. M. Jones, M. F. P. O'Boyle, J. Abella, A. González, and O. Ergin. Energy-efficient Register Caching with Compiler Assistance. *ACM Transactions on Architecture and Code Optimization*, 6(4):1–23, October 2009.

[20] P. Kogge et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical Report TR-2008-13, University of Notre Dame, September 2008.

[21] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, March 2005.

[22] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In *International Symposium on Computer Architecture*, pages 59–70, May 2002.

[23] A. S. Leon, B. Langley, and J. L. Shin. The UltraSPARC T1 Processor: CMT Reliability. In *IEEE Custom Integrated Circuits Conference*, pages 555 –562, September 2007.

[24] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical report, HP Laboratories, April 2009.

[25] P. R. Nuth and W. J. Dally. A Mechanism for Efficient Context Switching. In *International Conference on Computer Design on VLSI in Computer & Processors*, pages 301–304, October 1991.

[26] P. R. Nuth and W. J. Dally. The Named-State Register File: Implementation and Performance. In *International Symposium on High Performance Computer Architecture*, pages 4–13, January 1995.

[27] NVIDIA. Compute Unified Device Architecture Programming Guide Version 2.0. `http://devloper.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`, June 2008.

[28] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. `http://nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[29] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A Scalable Instruction Queue Design Using Dependence Chains. In *International Symposium on Computer Architecture*, pages 318–329, May 2002.

[30] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-core x86 Architecture for Visual Computing. In *International Conference and Exhibition on Computer Graphics and Interactive Techniques*, pages 1–15, August 2008.

[31] R. Shioya, K. Horio, M. Goshima, and S. Sakai. Register Cache System not for Latency Reduction Purpose. In *International Symposium on Microarchitecture*, pages 301–312, December 2010.

[32] J. A. Swensen and Y. N. Patt. Hierarchical Registers for Scientific Computers. In *International Conference on Supercomputing*, pages 346–354, September 1988.

[33] E. Tune, R. Kumar, D. M. Tullsen, and B. Calder. Balanced Multithreading: Increasing Throughput via a Low Cost Multithreading Hierarchy. In *International Symposium on Microarchitecture*, pages 183–194, December 2004.

[34] T. Ungerer, B. Robič, and J. Šilc. A Survey of Processors with Explicit Multithreading. *ACM Computing Surveys*, 35(1):29–63, March 2003.

[35] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *International Symposium on Performance Analysis of Systems and Software*, pages 235–246, March 2010.

[36] H. Zeng and K. Ghose. Register File Caching for Energy Efficiency. In *International Symposium on Low Power Electronics and Design*, pages 244–249, October 2006.

[37] X. Zhuang and S. Pande. Resolving Register Bank Conflicts for a Network Processor. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 269–278, September 2003.