2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor

Mark Gebhart[1,2] Stephen W. Keckler[1,2] Brucek Khailany[1] Ronny Krashinsky[1] William J. Dally[1,3]

[1]NVIDIA [2]The University of Texas at Austin [3]Stanford University

{mgebhart, skeckler, bkhailany, rkrashinsky, bdally}@nvidia.com

Abstract

Modern throughput processors such as GPUs employ thousands of threads to drive high-bandwidth, long-latency memory systems. These threads require substantial on-chip storage for registers, cache, and scratchpad memory. Existing designs hard-partition this local storage, fixing the capacities of these structures at design time. We evaluate modern GPU workloads and find that they have widely varying capacity needs across these different functions. Therefore, we propose a unified local memory which can dynamically change the partitioning among registers, cache, and scratchpad on a per-application basis. The tuning that this flexibility enables improves both performance and energy consumption, and broadens the scope of applications that can be efficiently executed on GPUs. Compared to a hard-partitioned design, we show that unified local memory provides a performance benefit as high as 71% along with an energy reduction up to 33%.

1. Introduction

Modern GPUs have emerged as an attractive platform for high performance computing. Oriented to throughput processing, GPUs are highly parallel with hundreds of cores and extremely high-bandwidth external memory systems. GPUs employ thousands of chip-resident threads to drive these parallel resources. With so many threads, register files are the largest on-chip memory resource in current GPUs. However, GPUs also provide both scratchpad memories and caches. These local resources provide low latency and high bandwidth access, as well as flexible scatter/gather addressing. In contrast to register files, scratchpad and cache memories allow threads to share data on chip, avoiding costly round trips through DRAM.

Although GPU architectures have traditionally focused primarily on throughput and latency hiding, data locality and reuse are becoming increasingly important with power-limited technology scaling. The energy spent communicating data within a chip rivals the energy spent on actual computation, and an off-chip memory transfer consumes orders of magnitude greater energy than an on-chip access. These trends have made on-chip local memories one of the most crucial resources for high performance throughput processing. As a result, in addition to their large and growing register files, future GPUs will likely benefit from even larger primary cache and scratchpad memories. However, these resources can not all grow arbitrarily large, as GPUs continue to be area-limited even as they become power limited.

Unfortunately a one-size-fits-all approach to sizing register file, scratchpad, and cache memories has proven difficult. To maximize performance, programmers carefully tune their applications to fit a given design, and many of these optimizations must be repeated for each new processor. Even after careful optimization, different programs stress the GPU resources in different ways. This situation is exacerbated as more applications are mapped to GPUs, especially irregular ones with diverse memory requirements.

In this work, we evaluate unified local memory with flexible partitioning of capacity across the register file, scratchpad (shared memory in NVIDIA terminology), and cache. When resources are unified, aggregate capacities can be allocated differently according to each application's needs. This design may at first seem fanciful, as register files have typically had very different requirements than other local memories, particularly in the context of CPUs. However in GPUs, register files are already large, highly banked, and built out of dense SRAM arrays, not unlike typical cache and scratchpad memories. Still, a remaining challenge for unification is that even GPU register files are very bandwidth constrained. For that reason, we build on prior work that employs a two-level warp scheduler and a software-controlled register file hierarchy [8, 9]. These techniques reduce accesses to the main register file by 60%, mitigating the potential overheads of moving to a unified design with shared bandwidth.

Unified memory potentially introduces several overheads. For applications that are already tuned for a fixed partitioning, the main overhead is greater bank access energy for the larger unified structure. Another potential drawback is that with more sharing, unified memory can lead to more bank conflicts. Our analysis shows that even for benchmarks that do not benefit from the unified memory design, the performance and energy overhead is less than 1%.

The unified memory design provides performance gains ranging from 4–71% for benchmarks that benefit from increasing the amount of one type of storage. In addition, DRAM accesses are reduced by up to 32% by making better use of on-chip storage. The combination of improved performance and fewer DRAM accesses reduces energy by up to 33%.

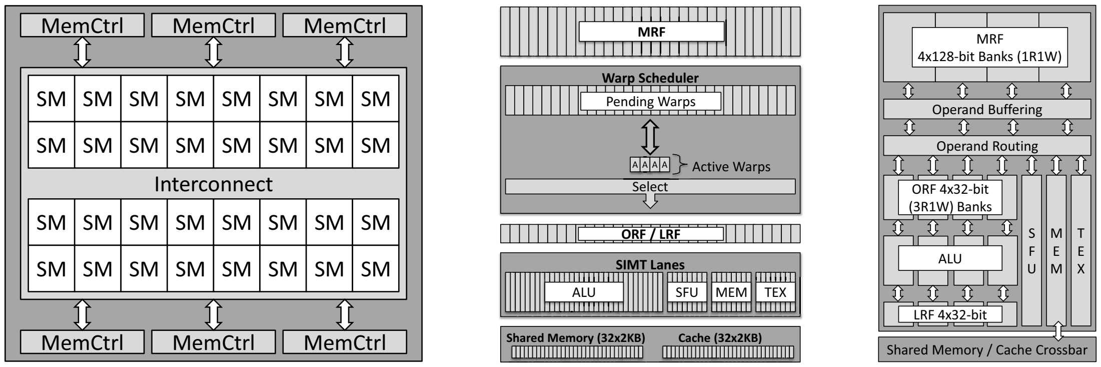
The rest of this paper is organized as follows. Section 2 describes our baseline GPU model. Section 3 characterizes the sensitivity to register file, shared memory, and cache capacity of modern GPU workloads. Section 4 proposes our unified memory microarchitecture. Sections 5 and 6 discuss our methodology and results. Sections 7 and 8 describe related work and conclusions.

2. Background

While GPUs are increasingly being used for compute applications, most design decisions were made to provide high graphics performance. Graphics applications have inherent parallelism, with memory access patterns that are hard to capture in a typical CPU L1 cache [7]. To tolerate DRAM latency and provide high performance, GPUs employ massive multithreading. Additionally, programmers can explicitly manage data movement into and out of low-latency on-chip scratchpad memory called *shared memory*.

Figure 1 shows the design of our baseline GPU, which is loosely modeled after NVIDIA's Fermi architecture. The figure represents a generic design point similar to those discussed in the literature [2, 16, 25], but is not intended to correspond directly to any existing

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 (c) 4-Wide SIMT lane detail

Figure 1: Baseline GPU architecture.

industrial product. The GPU consists of 32 streaming multiprocessors (SMs) and 6 high-bandwidth DRAM channels for a total of 256 bytes/cycle of DRAM bandwidth. Figure 1b shows an SM containing 32 SIMT (single-instruction, multiple thread) lanes that each execute up to one thread instruction per cycle. A group of 32 threads form an execution unit called a *warp*. The SIMT model executes all threads in a warp together using a common physical program counter. While the hardware supports control-flow divergence of threads within a warp, the SM operates most efficiently when all threads execute along a common control-flow path. Warps are grouped into larger units called co-operative thread arrays (CTAs) by the programmer. Threads in the same CTA execute on the same SM and can communicate through shared memory. A program may consist of one or more *kernels*, each consisting of one or more CTAs.

2.1. Baseline SM Architecture

In this work, we focus on the design of the SM shown in Figures 1b and 1c. The SM has up to 1024 resident threads, and a 32-entry, single-issue, in-order warp scheduler selects one warp per cycle to issue an instruction. Each SM provides 64KB of local scratchpad storage known as shared memory, 64KB of cache, and a 256KB register file. While these are large capacity structures compared to a uniprocessor, the SM provides on average only 256 bytes of registers, 64 bytes of data cache, and 64 bytes of shared memory per thread. Figure 1c provides a 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.

We leverage prior work which introduced a two-level warp scheduler and a software controlled register file hierarchy [8, 9]. The two-level warp scheduler divides the 32 warps present on an SM into an active set and an inactive set. Only warps in the active set are allowed to issue instructions, and warps are moved to the inactive set when they encounter a dependence on a long-latency operation. The software controlled register file hierarchy introduces two additional levels beyond the main register file (MRF): the operand register file (ORF) with 4 entries per thread, and a last result file (LRF) with a single entry per thread. Only active warps can allocate values in the ORF and LRF. When an active warp is descheduled, all of its live values must be in the MRF. The compiler controls all data movement between the MRF, ORF, and LRF. The result of these prior techniques is a reduction in the number of accesses to the MRF of 60%, without a performance loss, resulting in a significant savings in register file energy and MRF bandwidth.

Each MRF bank is 16 bytes wide with 4 bytes allocated to the same-named architectural register for threads in each of the 4 SIMT lanes in the cluster. Each bank has a capacity of 8KB, providing a total of 256KB of register file capacity per SM. Registers are interleaved across the register file banks to minimize bank conflicts. Instructions that access multiple values from the same bank incur a cycle of delay for each access beyond the first. 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. Stalls due to bank conflicts are rare and can be minimized with compiler techniques [27].

Each SM contains 64KB of cache and 64KB of shared memory. Each of these structures is composed of 32 2KB banks, and each bank supports one 4-byte read and one 4-byte write per cycle. The cache uses 128-byte cache lines which span all 32 banks, and only supports aligned accesses with 1 tag lookup per cycle. Shared memory supports scatter/gather reads and writes, subject to the limitation of one access per bank per cycle. Avoiding shared memory bank conflicts is a common optimization employed by programmers. The cache and shared memory banks are connected to the memory access units in the SM clusters through a crossbar.

2.2. Unified Cache and Shared Memory

Fermi has a unified cache and shared memory, providing programmers a limited choice of either a 16KB cache and a 48KB shared memory or a 48KB cache and a 16KB shared memory [16]. The memory configuration is controlled through a CUDA library function. Section 6.3 shows that a limited form of flexibility across shared memory and cache, like that found in Fermi, has benefits. However, a more flexible solution across all three types of storage (register file, cache, and shared memory) further improves both performance and energy consumption.

3. Application Characterization

In this section, we characterize modern GPU applications based on their usage of registers, shared memory, and cache. We begin with a large number of benchmarks and show that modern workloads fall into several different categories. Next, we explain in detail why some applications benefit from larger capacity in a given type of storage. Finally, we study the performance sensitivity of applications to storage capacity.

Table 1: Workload characteristics.

Workload	Registers per thread (no spills)	Registers per Thread (normalized dynamic instructions)					RF Size full occupancy no spills (KB)	Shared Memory (bytes / thread)	Cache Size (normalized DRAM Accesses)		
		18	24	32	40	64			0	64KB	256KB
Shared Memory Limited											
Needle [3]	18	1.02	1	1	1	1	72	264.1	0.85	1	1
sto [2]	33	1.18	1.08	1	1	1	132	127	3.95	1	1
lu [3]	20	1	1	1	1	1	80	96	1.94	1.46	1
Cache Limited											
GPU-mummer [3]	21	1.04	1	1	1	1	84	0	1.48	1.01	1
BFS [3]	9	1	1	1	1	1	36	0	1.46	1.13	1
Backprop [3]	17	1.02	1	1	1	1	68	2.125	1.56	1	1
MatrixMul [15]	17	1.04	1	1	1	1	68	8	4.77	1	1
Nbody [15]	23	1	1	1	1	1	92	0	3.52	1	1
VectorAdd [15]	9	1	1	1	1	1	36	0	3.88	1	1
srad [3]	18	1	1	1	1	1	72	24	1.22	1.20	1
Register Limited											
DGEMM [11]	57	1.42	1.23	1.01	1	1	228	66.5	1	1	1
PCR [26]	33	1.39	1.18	1.03	1	1	132	20	2.88	1.29	1
BicubicTexture [15]	33	1.18	1.10	1.05	1	1	132	0	1	1	1
hwt [3]	35	1.04	1.04	1.04	1	1	140	23	1	1	1
ray [2]	42	1.18	1.11	1.08	1.05	1	168	0	1.02	1.07	1
Balanced / Minimal Capacity Requirements											
Hotspot [3]	22	1.21	1	1	1	1	88	12	1.44	1	1
RecursiveGaussian [15]	23	1.02	1	1	1	1	92	2.125	1.04	1.03	1
Sad [17]	31	1.01	1	1	1	1	124	0	1.01	1.01	1
ScalarProd [15]	18	1.01	1	1	1	1	72	16	1	1	1
SGEMV [11]	14	1	1	1	1	1	56	4	1.01	1.01	1
SobolQRNG [15]	12	1	1	1	1	1	48	2	1	1	1
aes [2]	28	1.30	1.18	1	1	1	112	24	1	1	1
Dct8x8 [15]	26	1.16	1.10	1	1	1	104	0	1	1	1
DwtHaar1D [15]	14	1	1	1	1	1	56	8	1	1	1
lps [2]	15	1	1	1	1	1	60	19	1.48	1	1
nn [2]	13	1	1	1	1	1	52	0	20.81	1.07	1

3.1. Workload Characterization

We characterize these applications along three axes:

- Register usage: Two parameters are related to register file capacity: registers per thread and number of threads. Each thread is allocated registers for thread private values, with the same number of registers allocated for every thread in a kernel. Modern GPUs support a very large number of registers per thread. However, using more registers per thread results in fewer threads per SM, as the register file is shared across the SM. The compiler inserts spill and fill code when there are not enough registers available. We use the number of dynamic instructions executed as a metric to measure the overhead of register spills.
- Shared memory usage: Shared memory tradeoffs are controlled by the programmer, with each kernel specifying the total shared memory required per CTA along with the number of threads per CTA. The physical shared memory capacity available in an SM then dictates the maximum number of CTAs that can be mapped, if the register file capacity does not become a bottleneck first. While the programmer can often adjust shared memory requirements by changing an application's blocking pattern, we evaluate existing benchmarks that have fixed shared memory requirements per thread. Section 6.5 discusses tuning the shared memory requirements to exploit the unified design.
- Cacheable memory usage: The amount of spatial and temporal locality varies from application to application. Streaming applications mainly have spatial locality, but often have some degree of access redundancy which can be filtered by a small cache. Applications with cache blocking or a large number of register spills have higher temporal locality. The cache is a very scarce resource, and our baseline configuration has only 64 bytes on a per-thread basis. We use the number of DRAM accesses as a metric for the cache's effectiveness.

Table 1 presents an analysis of a range of CUDA applications according to the above criteria. Columns 2–8 show the per-thread register requirements, with column 2 showing the number of registers per thread required to eliminate spills. Columns 3–7 show the increase in dynamic instructions due to spill and fill code with different numbers of registers per thread. All of our surveyed benchmarks experience no spills when 64 registers per thread are available. Hand tuned programs tend to use more registers per thread than compiled programs as the programmer can block data into the register file for higher performance. `DGEMM, PCR,` and `BicubicTexture` all experience a large number of spills with a small number of registers per thread. Column 8 shows the register file capacity required to achieve full occupancy without experiencing register spills. The capacity required ranges from 36KB to 228KB. Column 9 shows the number of bytes of shared memory required per thread. Many applications need less than 20 bytes per thread, particularly when developed to fit the small shared memory capacities of early GPUs. `Needle` on the other hand, requires a large amount of shared memory. Columns 10–12 show the number of DRAM accesses for different capacity primary data caches. In general, as the cache capacity is increased, DRAM ac-

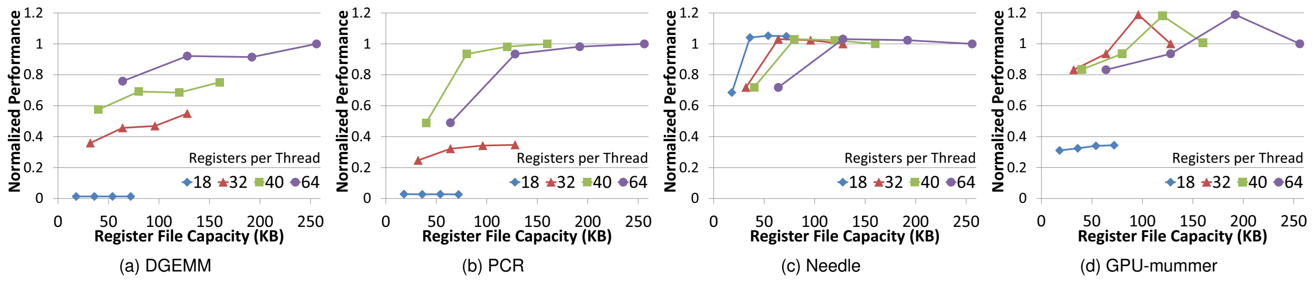


Figure 2: Performance as a function of register file capacity (with 64KB cache and unbounded shared memory), normalized to 64 registers per thread and 1,024 threads per SM.

cesses decrease. This decrease in DRAM traffic is due to the cache's ability to filter traffic and amplify bandwidth. The DRAM bandwidth demand can actually go up when using a cache, particularly when the cache line size exceeds the minimum DRAM fetch size. For example, `Needle` fetches unneeded data because only a fraction of the cache line is used after fetch.

Table 1 demonstrates that different applications place different stresses on the register file, shared memory, and cache structures. Many of the benchmarks fall into the balanced / minimal capacity requirements category as they were developed to fit the design of existing GPUs. As new emerging applications are ported to GPUs and applications are optimized to take advantage of our unified design, we expect to see more diversity in the memory requirements.

3.2. Application Case Studies

To provide greater insight into the advantages of the unified design, we discuss in detail the benchmarks which see a significant performance benefit from higher capacity in a given type of storage.

`Needle` implements the Needleman-Wunsch algorithm for DNA sequence alignment using dynamic programming [3]. The algorithm constructs a large (2048 by 2048 entry) matrix where each entry depends on its north, west, and north-west neighbor. The problem is broken into subblocks to make use of shared memory. The size of the subblock is a key parameter for this algorithm. Larger subblocks improve performance, but increase the shared memory requirements quadratically. Section 6.5 discusses the choice of blocking factor in more detail.

`LU` performs LU decomposition to solve a set of linear equations [3]. The kernel requires a moderate amount of registers but a high capacity shared memory. A large cache can exploit the reuse patterns as values in the input matrix are accessed repeatedly.

`GPU-mummer` implements DNA sequence alignment using graph traversal [3]. The algorithm consist of many parallel graph traversals across a large reference suffix tree. Each thread processes a single independent query. This workload does not use shared memory, as the working set size depends on the input. If the reference suffix tree is cached, a large performance gain is possible.

`BFS` is a breadth-first search of a graph with one million nodes [3]. It does not make use of shared memory and uses a small number of registers per thread. The application benefits from caching as the node and edge list is accessed repeatedly.

`SRAD` is an image processing application that relies on partial differential equations [3]. It uses a moderate number of registers and shared memory per thread, but benefits greatly from a large primary cache. Each output element is computed based on its four neighbors, allowing the cache to filter DRAM accesses.

`DGEMM` is an optimized double precision matrix multiplication kernel from the MAGMA library [11]. Two temporary matrices in shared memory capture subblock temporal locality. There is little performance benefit from caching. Each thread requires 57 registers per thread to eliminate spills, requiring a large register file.

`PCR` is a parallel cyclic reduction kernel that solves a tridiagonal linear system [26]. The algorithm uses shared memory to store temporary data and streams a large dataset from global memory. The large amount of communication between steps of the algorithm requires high bandwidth access to shared memory.

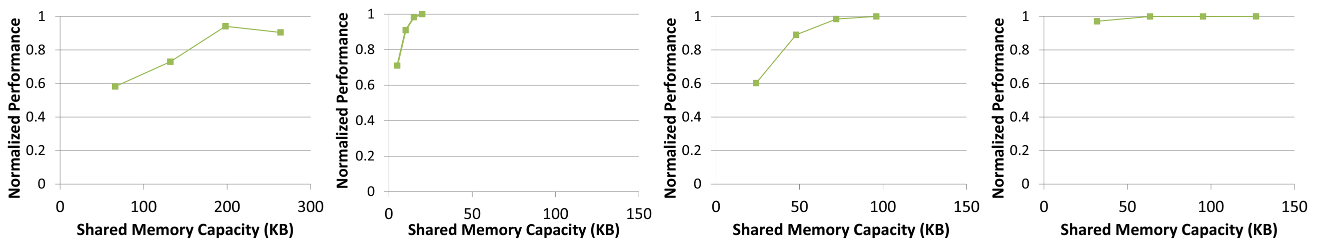
`RAY` performs ray-tracing with each thread responsible for rendering a single pixel; several levels of reflections and shadows are modeled. The kernel does not use shared memory but does require a large number of registers. A larger data cache is able to capture the environment, reducing the number of DRAM accesses.

3.3. Performance Sensitivity Study

Finally, we explore the performance sensitivity to the capacity of the register file, shared memory, and cache. We present limit studies which highlight the diverse memory requirements of modern workloads and the performance gains that can be achieved with larger storage structures. The details of our evaluation methodology are in Section 5. Because of the large number of benchmarks that we characterize in Table 1, we only present results for a subset of benchmarks which exhibit unique behaviors across the three different types of on-chip storage.

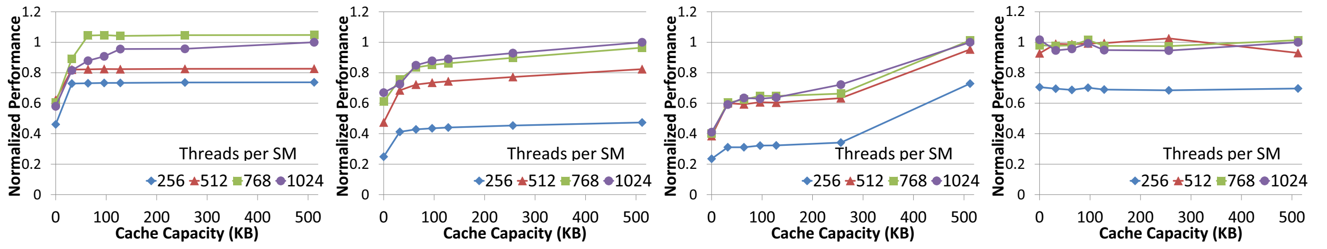
3.3.1. Register File Capacity: Register file capacity is a function of both the number of registers allocated to each thread and the number of concurrent threads. Performance is penalized when the number of registers per thread is small, which results in a large number of spills and fills. Likewise, applications that must tolerate DRAM accesses experience performance degradations when the number of concurrent threads is small.

Figure 2 illustrates the relationship between performance and register file capacity for four different types of applications. Each line in the graph shows performance with a different number of registers per thread. The performance penalty of spills can be seen by comparing the four lines. The points on a given line show performance for 256, 512, 768, and 1024 threads per SM. `DGEMM` requires both a large number of registers per thread and a large number of threads to maximize performance. These types of applications that require both a large number of registers per thread and a large number of concurrent threads stress the capacity of the register files found on current GPUs. `PCR` experiences a large number of spills with 18 or 32 register per thread and is less sensitive to thread count than `DGEMM`. There is no advantage to using more registers per thread than is necessary to eliminate spills. `Needle` is an example of an application that eliminates



(a) Needle (b) PCR (c) LU (d) STO

Figure 3: Performance versus shared memory capacity (with 64 registers per thread and 64KB of cache), normalized to 1,024 threads per SM. Note the wider x-axis on Needle.



(a) GPU-mummer (b) BFS (c) PCR (d) Needle

Figure 4: Performance as a function of cache capacity (with 64 registers per thread and unbounded shared memory), normalized to 512KB cache and 1,024 threads per SM.

spills even with as few as 18 registers per thread. Further, increasing thread count beyond 512 threads does not increase performance. DRAM latency tolerance is not important for this application, as it operates mainly out of shared memory. The spikes in performance in Figure 2d result from the interaction between the cache size and thread count. Changing the thread count can change performance due to interactions with the thread scheduler, especially when the larger number of threads are not needed to tolerate DRAM latency.

3.3.2. Shared Memory Capacity: Figure 3 shows the tradeoff in performance and shared memory capacity. Each point along a line shows an increasing number of threads per SM ranging from 256 to 1,024 in increments of 256. To isolate the effects of shared memory, these experiments use a large register file, eliminating register spills, and a 64KB cache. The application with the largest shared memory needs is `Needle`, which requires over 200KB. We discuss alternate blocking factors that can be used for `Needle` in Section 6.5. The shared memory usage of `PCR` is typical of today's applications. There is a large performance gain from maximizing thread count and even with the maximum number of threads per SM only 20KB of shared memory is required. `LU` is an example of an application that requires more shared memory than is present on today's GPUs and maximizing thread count improves performance. `STO` is an example where the application operates primarily out of shared memory, reducing the importance of running a large number of threads to tolerate DRAM latency. A small number of threads can still achieve high performance and minimizes the shared memory requirements.

3.3.3. Cache Capacity: Figure 4 shows the performance sensitivity to cache capacity. In these graphs, each line shows a different number of threads per SM (ranging from 256 to 1,024), and each point along a line shows performance with a different cache capacity. To isolate the effects of cache capacity, the register file is sized to eliminate spills and shared memory is unbounded. Running more threads per SM helps to tolerate latency from main memory access, but also reduces the amount of cache available on a per-thread basis. `BFS` and `PCR` benefit from having a large cache. In particular, `PCR` sees a large performance benefit moving from a 256KB to 512KB cache. `GPU-mummer` sees a performance benefit from caching, but it has a small working set for the input datasets we used. We expect a greater improvement with larger datasets. `Needle` is an example of an application that sees little performance benefit from caching as it operates mostly out of shared memory.

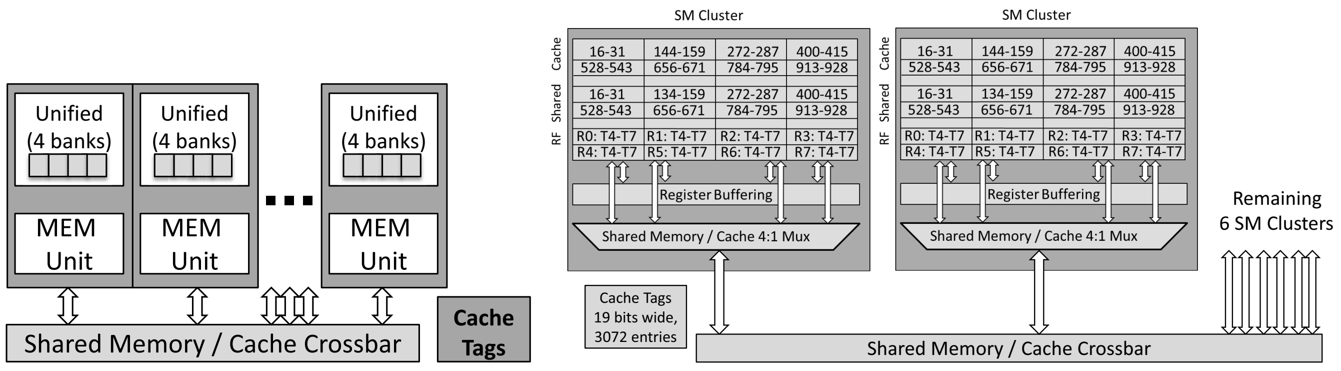
4. Microarchitecture

The characterization in Section 3 shows that modern GPU workloads have diverse local storage requirements and a single resource is often most critical to performance of a given application. We propose a *unified memory* architecture that aggregates these three types of storage and allows for a flexible allocation on a *per-kernel basis.*

4.1. Overview

Figures 5a and 6 compare the microarchitectures of the baseline design and our proposed unified architecture. The baseline design is structured as discussed in Section 2.1. In the unified design, all data storage is moved into the SM clusters. Effectively, the unified design merges together the 32 MRF banks, 32 shared memory banks, and 32 cache banks. Although we evaluate a range of capacities, the number of unified banks is always 32 per SM, to keep bandwidth constant. Each unified bank supports 1 read and 1 write per cycle, as do the banks in the baseline design. Also similar to the baseline design, the SM clusters in the unified design are connected by a crossbar to transfer data between the memory access units and other SM clusters.

As with the partitioned design, the cache tags are stored outside the SM clusters and 1 tag lookup can be processed per cycle. A 384KB unified design requires up to 7.125KB of tag storage compared with the baseline 64KB cache requiring 1.125KB. This overhead can be reduced by limited the maximum cache size in the unified design.


Unified (4 banks)
MEM Unit
Shared Memory / Cache Crossbar
Cache Tags
SM Cluster
Cache
Shared
RF
16-31 528-543
144-159 656-671
272-287 784-795
400-415 913-928
134-159 656-671
R0: T4-T7 R1: T4-T7 R2: T4-T7 R3: T4-T7
R4: T4-T7 R5: T4-T7 R6: T4-T7 R7: T4-T7
Register Buffering
Shared Memory / Cache 4:1 Mux
Cache Tags 19 bits wide, 3072 entries
Remaining 6 SM Clusters

(a) Unified SM architecture, 3 of 8 SM clusters shown.

(b) Detailed address mapping: RF: thread ID / register ID, Shared Memory/Cache: address (bytes).

Figure 5: Proposed unified memory microarchitecture.

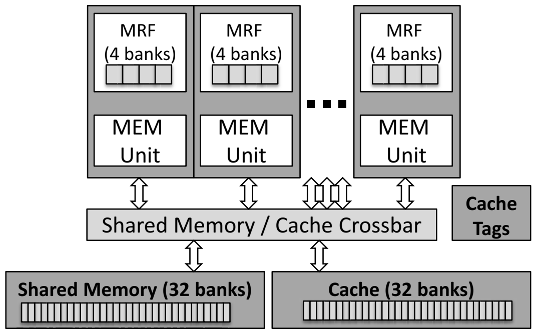


Figure 6: Baseline SM architecture, 3 of 8 SM clusters shown.

4.2. Unified Memory Bank Design

Each unified memory bank is 16 bytes wide with byte-enable support. Figure 5b shows how registers, cache, and shared memory are mapped across the banks. This figure shows 2 of the 8 SM clusters found in one SM. Threads from a single warp are mapped evenly across the 8 SM clusters, with 4 threads executing on each SM cluster. As all of a thread's register file entries are located in the same SM cluster where it executes, register file values are not communicated between SM clusters. The unified memory architecture does not change the register file bank mapping, bank widths, or register muxing in any way. The cache line size is 128 bytes in both the partitioned and unified designs. As shown in Figure 5b, the cache line is address-partitioned across 8 of the unified banks, 1 from each of the SM clusters. The shared memory address space is mapped across the banks in a similar manner. The unified memory design uses a smaller number of larger memory banks. The banks are sized such that the increase in bank size does not result in additional cycles required for bank access. The bank access is not on the processor's critical path, allowing for a larger memory bank.

When accessing cache or shared memory, only a single bank is used from each of the 8 clusters. This single bank routes its 16 bytes of data onto the crossbar, providing a peak shared memory or cache bandwidth of 128 bytes per cycle, identical to the baseline partitioned design. Compared with the partitioned design, the unified design adds one level of additional muxing for shared memory and cache accesses across SM clusters. Section 5 describes how we account for the extra wiring energy to access this multiplexor. This 4 to 1 mux is used to select which bank should access the crossbar and is only traversed for remote memory traffic, not for register file accesses. However, this single bank per cluster design is more restrictive than the partitioned design. To be bank-conflict free, a warp's shared memory accesses must coalesce to 8 banks rather than 32. A more aggressive design allows multiple banks in a single cluster to be accessed to increase the scatter / gather bandwidth. This enhanced design increases the complexity of the data muxing in a cluster, but still only allows 16 bytes per cluster to enter the crossbar. We simulated this design and found that it had an average performance improvement of 0.5%, compared to the simpler design. Our results in Section 6 assume the simpler design.

4.3. Arbitration Conflicts

In our baseline design, bank conflicts only occur within a single type of storage. With the unified design, accesses to the register file and cache or shared memory can conflict with each other. We refer to these conflicts as arbitration conflicts. One of the key enablers of the unified design is the software controlled register file hierarchy, which fetches most operands from the ORF or LRF and greatly reduces the required bandwidth to the MRF [9]. We model all conflicts and give priority to register access before cache or shared memory, but find that the performance impact of conflicts is small. Memory instructions fetch a small number of register operands and these operands often come from the LRF or ORF rather than the MRF, minimizing the number of arbitration conflicts. Our design uses a write through cache, eliminating bank accesses for evicting dirty data. The large number of threads can also tolerate some additional latency from conflicts without harming performance.

4.4. Managing Partitioning

Modern GPU workloads typically contain several different kernels, each of which may have different memory requirements. Before each kernel launch, the system can reconfigure the memory banks to change the memory partitioning. Because the register file and shared memory are not persistent across CTA boundaries, the only state that must be considered when repartitioning is the cache. As we use a write-through cache, the cache does not contain dirty data to evict. In the applications that we evaluated, the memory requirements across kernels were similar. Therefore, the results in Section 6 reflect choosing a single memory partitioning at the start of each benchmark and not reconfiguring the partitioning for each kernel.

4.5. Allocation Decisions

The unified memory architecture requires the programming system and hardware to determine the capacity of the register file, shared memory, and cache. We use the following automated algorithm to calculate the storage partitioning evaluated in Section 6:

- **Register File:** The compiler calculates how many registers per thread are required to avoid spills (Table 1, column 2).
- **Shared Memory:** The programmer specifies the amount of shared memory required per thread when constructing each kernel in the same manner as today's partitioned designs.
- **Thread count:** The hardware scheduler takes as input the number of registers per thread to avoid spills, the number of bytes of shared memory, and the overall capacity of the unified memory. The scheduler calculates the maximum number of threads by dividing the unified memory capacity by the per-thread register and shared memory requirements. Some applications see higher performance with fewer than the maximum number of threads, due to interactions with the thread scheduler and memory system. This phenomenon occurs both for the partitioned and unified design. Techniques like autotuning [24] can be used to automatically optimize thread count.
- **Cache:** Any remaining storage is allocated to the primary data cache.

5. Methodology

When possible, we used the default input sets and arguments distributed with the benchmarks described in Table 1, but we scaled down some of the workloads to make the simulation time tractable.

5.1. Simulation

We used Ocelot, a PTX dynamic compilation framework, to create execution and address traces [6]. We built a custom SM simulator that takes these traces as input and measures performance. We simulate execution using the SM parameters shown in Table 2. Our SM simulator runs the traces to completion, correctly modeling synchronization between threads in a CTA. We model execution for the full application running on a single SM and allocate 8 bytes per cycle of DRAM bandwidth making the simplifying assumption that the global DRAM bandwidth is evenly shared among all 32 SMs. Because the applications run each kernel many times across a large number of threads, modeling a single SM, rather than the full chip, simplifies simulation without sacrificing accuracy.

5.2. Energy Model

We assume a 32nm technology node for our energy evaluation using the parameters listed in Table 3 and focus on the following elements which are affected by our unified design:

- Bank Access Energy: Compared with the baseline partitioned design, the unified design uses a smaller number of larger banks, resulting in more energy per access to the main register file, shared memory, and cache. Table 4 shows dynamic read and write energy for SRAM banks of various sizes. These numbers are scaled using a combination of CACTI [13] and prior work that used synthesis for memory structures [8]. While the unified design increases bank access energy, especially for shared memory and cache accesses, Section 6 shows that this increase is small in comparison to total system energy.

Table 2: Simulation parameters.

Parameter	Value
SM Execution Width	32-wide SIMT
SM Execution Model	In-order
SM Register File Capacity	256 KB
SM MRF Bank Capacity	8 KB
SM Shared Memory Capacity	64 KB
SM Shared Memory Bandwidth	128 bytes/cycle
SM Cache Associativity	4-way
SM DRAM Bandwidth	8 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 3: Energy parameters.

Parameter	Value
Technology node	32 nm
Frequency	1 GHz
Voltage	0.9 V
Wire capacitance	300 fF / mm
Wire energy	1.9 pJ / mm
Dynamic power per SM	1.9 W
Leakage power per SM	0.9 W
Leakage power per KB of SRAM	2.37 mW
DRAM energy	40 pJ / bit

- Wiring Energy: In the baseline design, the cache and shared memory banks are 4 bytes wide. In the unified design, the banks are 16 bytes wide. To simplify the crossbar, we stripe cache lines across banks in different SM clusters as described in Section 4. However, we still incur an overhead of a 4:1 multiplexer. Furthermore, for an equal capacity design, the area of an SM cluster will increase as cache and shared memory storage is moved into the clusters. This increase in area will increase the overhead of the crossbar that connects the clusters. As we have not implemented a detailed physical design, we model these overheads as 10% additional energy relative to the bank access energy for cache and shared memory reads and writes. We also account for an increase in cache tag lookup energy with this factor.
- SRAM Leakage: Each of the unified and partitioned designs use different amounts of SRAM for the main register file, shared memory, and cache. We use an estimate of 2.37 mW per KB of SRAM capacity from prior work to calculate leakage for each design [10].
- DRAM Energy: Our architecture reduces DRAM accesses by making better use of on-chip memory. We model each DRAM access as consuming 40 pJ/bit [22].

We use a high-level GPU power model to account for the core dynamic and leakage energy. We assume a modern GPU in 32nm process technology that consumes 130 watts and contains 32 SMs. The SMs consume 70% of the chip-wide energy, with the remaining 30% consumed by the memory system. Assuming that leakage is one third of the chip-wide power, each SM consumes 1.9 watts of dynamic power and 0.9 watts of leakage power. Except for bank access and DRAM energy, we assume that dynamic power for the SM is constant across the various configurations. We use the performance of the baseline 256/64/64 configuration to calculate SM dynamic power for each benchmark. We adjust leakage for each configuration using the SRAM leakage data of 2.37mW per KB of capacity. On the

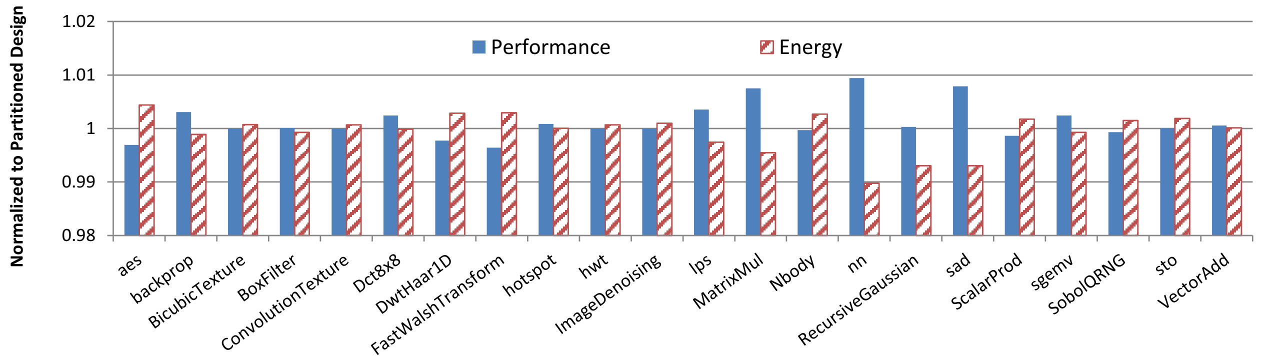


Figure 7: Performance (higher is better) and energy (lower is better) of unified design (384KB) normalized to an equal-capacity partitioned design for applications that do not benefit from unified storage (note the narrow range of the y-axis).

Table 4: Energy for 16-byte SRAM bank access (32nm) for unified and partitioned designs.

Structure	Bank Size	Read (pJ)	Write (pJ)
Partitioned			
256KB RF	8 KB	9.8	11.8
64KB Shared Memory	2 KB	3.9	5.1
64KB Cache	2 KB	3.9	5.1
Unified			
384KB Unified	12 KB	12.1	14.9

baseline design with 384KB of SRAM storage, 0.7 watts of leakage is from the core and 0.2 watts is from the SRAM. The SM and SRAM leakage energy is calculated for each design point based on performance. Design points with higher performance experience less leakage, since faster completion of the workload results in less time for transistors to leak.

6. Results

In this section, we evaluate the overheads and advantages of the unified memory design. We divide the benchmarks that we characterized in Section 3 into two sets: those that see no benefit from the unified design (Section 6.1) and those that benefit (Section 6.2).

6.1. Applications With No Benefit From Unified Memory

First, we evaluate the set of benchmarks that do not benefit from the unified design. These benchmarks are not able to make use of the additional capacity provided by the unified design to improve performance. However, the unified design does not harm performance or energy. Many of these benchmarks were tuned for the small capacity structures present on early GPUs and may benefit from the unified design if they were tuned for larger capacity structures.

Figure 7 shows the performance and energy improvements of a 384KB unified design normalized to an equal-capacity partitioned design. Each SM in this baseline partitioned design contains a 256KB register file, a 64KB shared memory, and a 64KB primary data cache as described in Section 2. The unified design only slightly changes performance and energy for these benchmarks, with the largest changes less than 1%. The slight changes in performance and energy are mainly due to (1) changes in bank conflicts resulting from changing the bank width from 4 bytes in the partitioned case to 16 bytes in the unified case, and (2) from bank conflicts associated with combining the register file with shared memory and the cache. These results show that the performance degradation due to an increase in bank conflicts is negligible.

One of the potential overheads of the unified design is an increase in memory bank conflicts, as each memory bank supports only one read and one write operation per cycle. Bank conflicts are due to accesses from the same instruction or different instructions mapped to the same bank. The inter-instruction conflicts depend on the exact scheduling policy and instruction pipeline used. To get an estimate of the potential increase in bank conflicts from unified memory, we rely on a simplified model where we only track conflicts within a single warp instruction. For each warp instruction, we count the bank accesses across the 32 threads in the warp. We then impose a performance penalty of 1 cycle for each access beyond the first for the bank that was accessed the most by that warp instruction. For example, if one bank was accessed three times and another bank was accessed twice the instruction would be delayed by 2 cycles. This model is likely pessimistic, as accesses from a single warp instruction can actually span different clock cycles due to the pipeline design. In the partitioned design, bank conflicts occur (1) in the register file when an instruction tries to read multiple registers mapped to the same bank, and (2) in the cache and shared memory when threads in the same warp access values that are mapped to the same bank. In the unified design, additional arbitration conflicts occur when an instruction tries to read or write a value from the cache or shared memory that is mapped into the same bank as its register operands.

Table 5 quantifies the potential increase in bank conflicts by showing how many accesses each warp instruction makes to the same memory bank. In both designs, the vast majority of warp instructions make one or fewer accesses to each memory bank. The unified design experiences a small increase (0.6 percentage points) in the number of warp instructions that access a bank multiple times. However, Figure 7 shows this increase in accesses leads to a negligible performance change. The key enabler that allows the unification of on-chip memory without excessive numbers of arbitration conflicts is the register file hierarchy, which dramatically reduces the number of accesses to the main register file [9].

Relative to the partitioned architecture, the unified memory design slightly increases bank access energy due to its smaller number of banks, each with higher capacity. However, bank access energy

Table 5: Breakdown of warp instructions based on the maximum number of accesses to a single bank for the unified and partitioned design, averaged across Figure 7 benchmarks.

	Maximum accesses to a single bank per instruction				
	<= 1	2	3	4	>4
Partitioned	97.0%	2.7%	0.09%	0.14%	0.03%
Unified	96.4%	3.4%	0.01%	0.02%	0.21%

makes up a small component of overall system energy. Figure 7 shows that the overall changes in energy are negligible. The largest increase in energy is 0.9% for `nn` and on average the energy of the unified design is 0.06% lower than that of the partitioned design. Much of the energy spent in the register file system, cache, and shared memory is for control and wiring rather than actual bank access. Additionally, the register file hierarchy reduces the number of accesses to the main register file, minimizing register file bank access energy for both the partitioned and unified designs. These results show that even though these benchmarks do not benefit from the unified design the overhead from our proposed design is negligible.

6.2. Applications That Benefit From Unified Memory

Next, we evaluate benchmarks that see significant improvements from the unified memory architecture. We have made no source code modifications to these benchmarks to tune them for the unified memory architecture. As the analysis in Section 3 shows, modern applications have a variety of requirements in on-chip storage needs and the unified memory architecture is able to adapt on a per-application basis with the most efficient partitioning of on-chip storage.

As described in Section 4.5, the allocation decisions are managed automatically by the compiler and hardware. Figure 8 shows how the 384KB of unified memory was configured for each of these benchmarks. The amount of storage devoted to the register file ranges from 36KB on `bfs` to 228KB on `dgemm`. One of the applications, `needle`, devotes 264KB to shared memory to allow a larger number of concurrent threads to execute. The remaining applications that make use of shared memory devote less than 100KB of their on-chip storage to it. The unified memory design allows larger primary caches, as any remaining storage not used for the register file or shared memory serves as cache.

Figure 9 shows the performance, energy, and DRAM traffic improvements for eight benchmarks that see significant improvements. The performance improvements range from 4.2% to 70.8% with an average performance improvement of 16.2%. These performance improvements are the result of a combination of having a larger register file, shared memory, or cache. In many cases, the larger capacity register file or shared memory allows more concurrent threads to run, which allows the SM to better tolerate DRAM latency.

All of the benchmarks, except for `DGEMM`, see a reduction in DRAM traffic ranging from 1% to 32%. The reduction in DRAM accesses is primarily the result of having higher capacity caches. As DRAM bandwidth is and will continue to be a precious resource, minimizing off-chip traffic is vital to improving efficiency. The performance improvements along with the reduction in DRAM accesses lead to a reduction in chip-wide energy. The energy savings range from 2.8% to 33% across these eight applications. These savings are significant for today's power limited devices.

6.3. Comparison to Limited Unified Memory

As discussed in Section 2.2, Fermi has a limited form of unified memory. The programmer can choose between either 16KB of shared memory and 48KB of cache or 48KB of shared memory and 16KB of cache per SM. Our unified design allows all three types of storage found in the SM to be unified. We evaluate an equal-capacity Fermi-like design which has a total of 384KB of storage divided into a 256KB register file and either 96KB of shared memory and 32KB of cache or 32KB of shared memory and 96KB of cache. Figure 10 shows the improvement in performance, energy, and DRAM accesses compared to the partitioned design.

Table 6: Performance and energy of three unified memory capacities normalized to the the baseline partitioned design.

	Performance (higher is better)			Energy (lower is better)		
Benchmark	128KB	256KB	384KB	128KB	256KB	384KB
bfs	1.03	1.08	1.12	0.91	0.89	0.88
dgemm	0.77	1.01	1.08	1.13	0.95	0.94
lu	0.96	1.07	1.07	1.00	0.91	0.89
GPU-mummer	0.96	1.04	1.04	0.97	0.95	0.97
pcr	0.77	1.04	1.06	1.33	0.92	0.93
ray	0.94	1.03	1.13	1.01	0.95	0.89
srad	1.00	1.08	1.09	0.94	0.86	0.89
needle	1.29	1.75	1.71	0.76	0.64	0.67
Average	0.97	1.14	1.16	0.98	0.87	0.87
Figure 7 Benchmarks (Average)	**0.99**	**1.00**	**1.00**	**0.93**	**0.96**	**1.00**

The Fermi-like design is able to improve performance for all of the benchmarks between 1%–20%. However, comparing Figures 9 and 10 shows that the unified design achieves higher performance for all but one benchmark. The Fermi-like design actually achieves higher performance on `GPU-mummer` because this benchmark is extremely sensitive to cache size and thread scheduling. The smaller capacity cache provided by the Fermi-like design results in slightly different cache behaviors that interact differently with the thread scheduler. Overall, the gains in energy-efficiency and DRAM traffic reduction are higher for the unified architecture than the Fermi-like limited flexibility design.

6.4. Capacity Sensitivity

Next, we explore the sensitivity of performance and energy to the capacity of the unified memory. Larger unified memory designs improve performance at the cost of increased SRAM leakage. Table 6 shows performance and energy for a range of different unified memory capacities. Performance is generally maximized with 384KB of unified memory. `Needle` sees slightly higher performance with 256KB due to the choice of thread count and the resulting scheduling decisions made by the thread scheduler. The performance of the benchmarks from Figure 7 is flat across the range of capacities as they do not see a speedup from the larger capacity designs.

As the capacity of unified memory is increased, SRAM leakage increases. However, a larger capacity design can also reduce overall leakage (higher performance reduces runtime) and DRAM energy. The benchmarks that do not benefit from unified memory see the lowest energy with the 128KB design, which minimizes SRAM leakage. The benchmarks that benefit from the unified design generally see the lowest energy with either the 256KB or 384KB design. The tradeoff between performance, area, and leakage must be carefully considered when deciding how much storage should be allocated per SM. Compared with the partitioned or limited flexibility designs, the unified architecture gives designers more freedom as each memory structure must be provisioned for the maximum requirements of any workload. By dynamically partitioning storage, the unified design allows the amount of storage to be set based on the aggregate storage requirements of workloads.

6.5. Tuning Applications for Unified Architecture

Many applications are tuned to fit into the storage requirements of either the register file, shared memory, or cache. A partitioned design

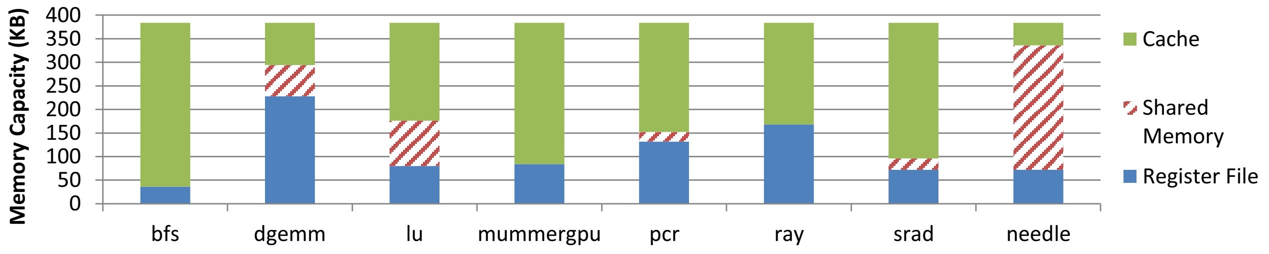


Figure 8: Partitioning of 384KB unified memory.

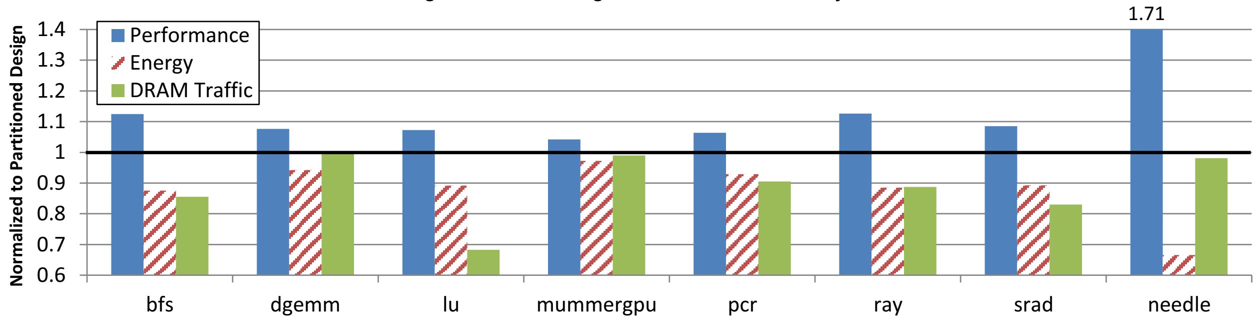


Figure 9: Performance (higher is better), energy (lower is better), and DRAM traffic (lower is better) of unified design (384KB) normalized to an equal-capacity partitioned design for applications that benefit from unified storage.

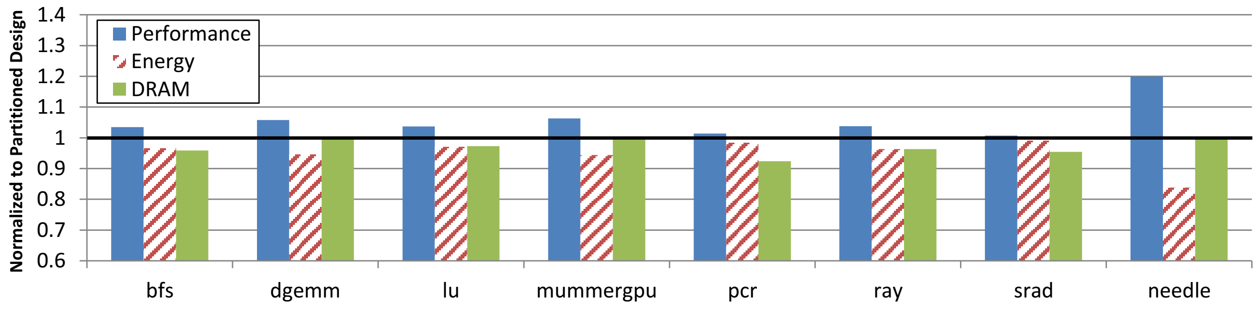


Figure 10: Performance (higher is better), energy (lower is better), and DRAM traffic (lower is better) of Fermi-like limited flexibility design (384KB) normalized to an equal-capacity partitioned design.

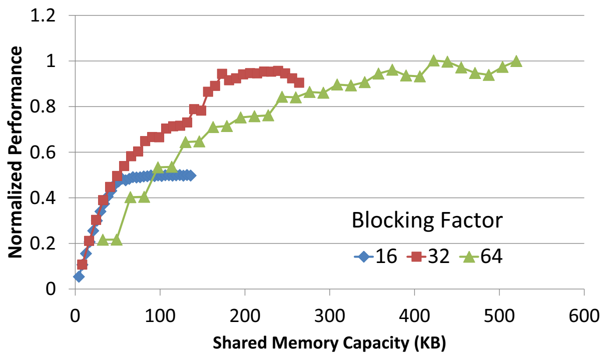


Figure 11: Performance of various blocking factors and shared memory requirements for needle.

forces applications to be tuned across the narrow capacity range of each structure. The unified architecture presents an opportunity to tune applications across the entire range of performance and unified memory capacity points. As a case study, Figure 11 shows performance as a function of shared memory capacity for three different shared memory blocking factors on `Needle`. Performance is normalized to the maximum shared memory capacity tested of 520KB which is required with a blocking factor of 64 and 1024 threads per SM. As the blocking factor is increased, the amount of shared memory required per thread increases. Each point along the lines represents increasing the number of concurrent threads from 32 to 1024 in increments of 32. When the amount of shared memory available is small, as found on prior generation GPUs, the blocking factor of 16 was used. The results discussed so far in this paper have used a blocking factor of 32, as this is the most efficient operating point when 64KB of shared memory is available. When more than 300KB of shared memory is available, a blocking factor of 64 provides slightly better performance and requires fewer concurrent threads than a blocking factor of 32. The unified design allows programmers the option of optimizing their applications over wider ranges of performance points and potentially utilizing more efficient algorithms.

7. Related Work

Several projects have considered reconfigurable memories that serve as either cache or scratchpad for designs other than GPUs, including Smart Memories [12], TRIPS [19], and the TI TMS320C62xx DSP [21]. Ranganathan et al. proposed a reconfigurable cache that could be divided into several partitions with each partition performing a different task [18]. Their work mainly focused on the cache design required for reconfigurability and provided a case study for using a cache partition for instruction reuse in a media processing application. The ALP project proposed to reconfigure part of the L1 data cache to serve as a vector register file when performing vector processing [20]. Cook et al. proposed mechanisms for flexible partitioning between cache and shared memory in a multi-core CPU [4]. Albonesi proposed selective cache ways which allows a subset of the ways in a set associative cache to be disabled to save power [1].

Volkov identified applications that achieve better performance by using fewer threads as this allows more registers to be allocated to each thread [23]. Recent work uses cyclic reduction as a case study on the tradeoffs between allocating values to the register file versus shared memory along with balancing the number of registers per thread and the number of threads per SM [5]. Murthy et al. developed a model for optimal loop unrolling for GPGPU programs that considers the increase in register pressure versus the potential improvements from unrolling [14]. Our flexible storage system can relax the programming burden associated with the fixed capacity storage structures and accommodate diverse workloads.

8. Conclusion

Modern applications have varying requirements in register file, cache, and shared memory capacity. Traditional GPUs require the programmer to carefully tune their applications to account for the size of each of these structures. In this work, we propose a unified on-chip storage for the register file, cache, and shared memory. This flexible structure can adjust the storage partitioning on a per application basis, providing a performance improvement as high as 71% along with an energy reduction up to 33%. The overhead of the flexibility is small, with a minimal increase in bank conflicts and a small increase in bank access energy. These overheads are negligible in terms of system performance and energy, even for benchmarks that do not benefit from the unified design. We explore the sensitivity to unified memory capacity and find that many benchmarks achieve energy savings with smaller capacity unified memory. Future systems could exploit this fact by disabling unneeded memory. Our unified equal capacity design provides meaningful energy efficiency improvements for a significant number of today's benchmarks, which are tuned for partitioned designs. By making the processor's storage more flexible, we broaden the scope of applications that GPUs can efficiently execute. Future applications or application tuning can further improve efficiency by taking advantage of this new flexibility.

Acknowledgments

We thank the anonymous reviewers, Trevor Mudge, 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.

References

[1] D. H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation," in *International Symposium on Microarchitecture*, November 1999, pp. 248–259.

[2] 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*, April 2009, pp. 163–174.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *International Symposium on Workload Characterization*, October 2009, pp. 44–54.

[4] H. Cook, K. Asanovic, and D. A. Patterson, "Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-131, September 2009.

[5] A. Davidson and J. D. Owens, "Register Packing for Cyclic Reduction: A Case Study," in *Workshop on General Purpose Processing on Graphics Processing Units*, March 2011, pp. 1–6.

[6] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogeneous Systems," in *International Conference on Parallel Architectures and Compilation Techniques*, September 2010, pp. 353 – 364.

[7] K. Fatahalian and M. Houston, "A Closer Look at GPUs," *Communications of the ACM*, vol. 51, no. 10, pp. 50–57, October 2008.

[8] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *International Symposium on Computer Architecture*, June 2011, pp. 235–246.

[9] M. Gebhart, S. W. Keckler, and W. J. Dally, "A Compile-Time Managed Multi-Level Register File Hierarchy," in *International Symposium on Microarchitecture*, December 2011, pp. 465–476.

[10] X. Guo, E. Ipek, and T. Soyata, "Resistive Computation: Avoiding the Power Wall with Low-Leakage STT-MRAM Based Computing," in *International Symposium on Computer Architecture*, June 2010, pp. 371–382.

[11] "MAGMA: Matrix Algebra for GPU and Multicore Architectures," http://icl.eecs.utk.edu/magma.

[12] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *International Symposium on Computer Architecture*, June 2000, pp. 161–171.

[13] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep., April 2009.

[14] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal Loop Unrolling for GPGPU Programs," in *International Symposium on Parallel and Distributed Processing*, April 2010, pp. 1–11.

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

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

[17] "Parboil Benchmark Suite," http://impact.crhc.illinois.edu/parboil.php.

[18] P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable Caches and their Application to Media Processing," in *International Symposium on Computer Architecture*, June 2000, pp. 214–224.

[19] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," in *International Symposium on Computer Architecture*, June 2003, pp. 422–433.

[20] R. Sasanka, M.-L. Li, S. V. Adve, Y.-K. Chen, and E. Debes, "ALP: Efficient Support for All Levels of Parallelism for Complex Media Applications," *ACM Transactions on Architecture and Code Optimization*, vol. 4, no. 1, March 2007.

[21] Texas Instruments, "TMS320C6202/C6211 Peripherals Addendum to the TMS320C6201/C6701 Peripherals Reference Guide (SPRU290)," Tech. Rep., August 1998.

[22] T. Vogelsang, "Understanding the Energy Consumption of Dynamic Random Access Memories," in *International Symposium on Microarchitecture*, December 2010, pp. 363–374.

[23] V. Volkov, "Better Performance at Lower Occupancy," in *GPU Technology Conference*, September 2010.

[24] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," in *International Conference for High Performance Computing*, 1998, pp. 1–27.

[25] 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*, March 2010, pp. 235–246.

[26] Y. Zhang, J. Cohen, and J. D. Owens, "Fast Tridiagonal Solvers on the GPU," in *Symposium on Principles and Practice of Parallel Programming*, January 2010, pp. 127–136.

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