Demystifying GPU Microarchitecture through Microbenchmarking

Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos
Department of Electrical and Computer Engineering, University of Toronto
{henry, myrto, alvandim, moshovos}@eecg.utoronto.ca

***Abstract*—Graphics processors (GPU) offer the promise of more than an order of magnitude speedup over conventional processors for certain non-graphics computations. Because the GPU is often presented as a C-like abstraction (e.g., Nvidia's CUDA), little is known about the characteristics of the GPU's architecture beyond what the manufacturer has documented. This work develops a microbechmark suite and measures the CUDA-visible architectural characteristics of the Nvidia GT200 (GTX280) GPU. Various undisclosed characteristics of the processing elements and the memory hierarchies are measured. This analysis exposes undocumented features that impact program performance and correctness. These measurements can be useful for improving performance optimization, analysis, and modeling on this architecture and offer additional insight on the decisions made in developing this GPU.**

I. Introduction

The graphics processor (GPU) as a non-graphics compute processor has a different architecture from traditional sequential processors. For developers and GPU architecture and compiler researchers, it is essential to understand the architecture of a modern GPU design in detail.

The Nvidia G80 and GT200 GPUs are capable of non-graphics computation using the C-like CUDA programming interface. The CUDA Programming Guide provides hints of the GPU performance characteristics in the form of rules [1]. However, these rules are sometimes vague and there is little information about the underlying hardware organization that motivates them.

This work presents a suite of microbenchmarks targeting specific parts of the architecture. The presented measurements focus on two major parts that impact GPU performance: the arithmetic processing cores, and the memory hierarchies that feed instructions and data to these processing cores. A precise understanding of the processing cores and of the caching hierarchies is needed for avoiding deadlocks, for optimizing application performance, and for cycle-accurate GPU performance modeling.

Specifically, in this work:

- We verify performance characteristics listed in the CUDA Programming Guide.
- We explore the detailed functionality of branch divergence and of the barrier synchronization. We find some non-intuitive branching code sequences that lead to deadlock, which an understanding of the internal architecture can avoid.

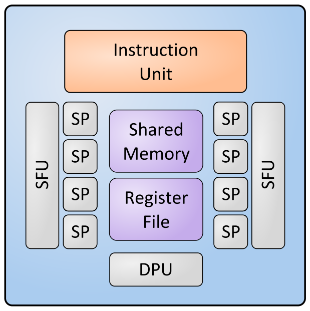


Fig. 1: Streaming Multiprocessor with 8 Scalar Processors Each

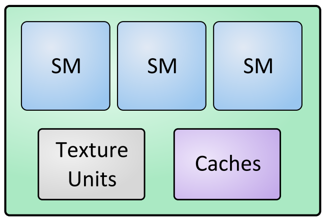


Fig. 2: Thread Processing Cluster with 3 SMs Each

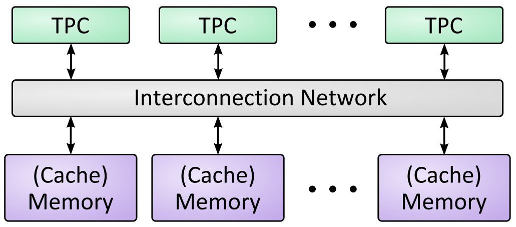


Fig. 3: GPU with TPCs and Memory Banks

- We measure the structure and performance of the memory caching hierarchies, including the Translation Lookaside Buffer (TLB) hierarchy, constant memory, texture memory, and instruction memory caches.
- We discuss our measurement techniques, which we believe will be useful in the analysis and modeling of other GPUs and GPU-like systems, and for improving the fidelity of GPU performance modeling and simulation [2].

The remainder of this paper is organized as follows. Section II reviews the CUDA computation model. Section III describes the measurement methodology, and Section IV presents the measurements. Section V reviews related work and Section VI summarizes our findings.

II. Background: GPU Architecture and Programming Model

A. GPU Architecture

CUDA models the GPU architecture as a multi-core system. It abstracts the thread-level parallelism of the GPU into a hierarchy of threads (*grids* of *blocks* of *warps* of *threads*) [1]. These threads are mapped onto a hierarchy of hardware resources. Blocks of threads are executed within Streaming Multiprocessors (SM, Figure 1). While the programming model uses collections of scalar threads, the SM more closely resembles an eight-wide vector processor operating on 32-wide vectors.

SM Resources	
SPs (Scalar Processor)	8 per SM
SFUs (Special Function Unit)	2 per SM
DPUs (Double Precision Unit)	1 per SM
Registers	16,384 per SM
Shared Memory	16 KB per SM

Caches	
Constant Cache	8 KB per SM
Texture Cache	6-8 KB per SM

GPU Organization	
TPCs (Thread Processing Cluster)	10 total
SMs (Streaming Multiprocessor)	3 per TPC
Shader Clock	1.35 GHz
Memory	8 × 128MB, 64-bit
Memory Latency	400-600 clocks

Programming Model	
Warps	32 threads
Blocks	512 threads max
Registers	128 per thread max
Constant Memory	64 KB total
Kernel Size	2 M PTX insns max

TABLE I: GT200 Parameters according to Nvidia [1], [3]

The basic unit of execution flow in the SM is the *warp*. In the GT200, a warp is a collection of 32 threads and is executed in groups of eight on eight *Scalar Processors* (SP). Nvidia refers to this arrangement as Single-Instruction Multiple-Thread (SIMT), where every thread of a warp executes the same instruction in lockstep, but allows each thread to branch separately. The SM contains arithmetic units, and other resources that are private to blocks and threads, such as per-block *shared memory* and the register file. Groups of SMs belong to Thread Processing Clusters (TPC, Figure 2). TPCs also contain resources (e.g., caches, texture fetch units) that are shared among the SMs, most of which are not visible to the programmer. From CUDA's perspective, the GPU comprises the collection of TPCs, the interconnection network, and the memory system (DRAM memory controllers), as shown in Figure 3. Table I shows the parameters Nvidia discloses for the GT200 [1], [3].

B. CUDA Software Programming Interface

CUDA presents the GPU architecture using a C-like programming language with extensions to abstract the threading model. In the CUDA model, *host* CPU code can launch GPU *kernels* by calling *device* functions that execute on the GPU. Since the GPU uses a different instruction set from the host CPU, the CUDA compilation flow compiles CPU and GPU code using different compilers targeting different instruction sets. The GPU code is first compiled into PTX "assembly", then "assembled" into native code. The compiled CPU and GPU code is then merged into a single "fat" binary [4].

Although PTX is described as being the assembly level representation of GPU code, it is only an intermediate representation, and was not useful for detailed analysis or microbenchmarking. Since the native instruction set is different and compiler optimization is performed on the PTX code, PTX code is not a good representation of the actual machine instructions executed. In most cases, we have found it most productive to write in CUDA C, then verify the generated machine code sequences at the native code level using decuda [5]. The use of decuda was mainly for convenience, as the generated instruction sequences can be verified in the native cubin binary. Decuda is a disassembler for Nvidia's machine-level instructions, derived from analysis of Nvidia's compiler output, as the native instruction set is not publicly documented.

III. Measurement Methodology

A. Microbenchmark Methodology

To explore the GT200 architecture, we create microbenchmarks to expose each characteristic we wish to measure. Our conclusions were drawn from analyzing the execution times of the microbenchmarks. Decuda was used to report code size and location when measuring instruction cache parameters, which agreed with our analysis of the compiled code. We also used decuda to inspect native instruction sequences generated by the CUDA compiler and to analyze code generated to handle branch divergence and reconvergence.

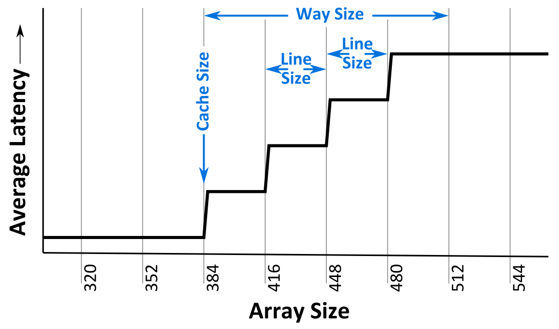
The general structure of a microbenchmark consists of GPU kernel code containing timing code around a code section (typically an unrolled loop running multiple times) that exercises the hardware being measured. A benchmark kernel runs through the entire code twice, disregarding the first iteration to avoid the effects of cold instruction cache misses. In all cases, the kernel code size is small enough to fit into the L1 instruction cache (4 KB, see Section IV-K). Timing measurements are done by reading the clock register (using clock()). The clock values are first stored in registers, then written to global memory at the end of the kernel to avoid slow global memory accesses from interfering with the timing measurements.

When investigating caching hierarchies, we observed that memory requests which traverse the interconnect (e.g., accessing L3 caches and off-chip memory) had latencies that varied depending on which TPC was executing the code. We average our measurements across all 10 TPC placements and report the variation where relevant.

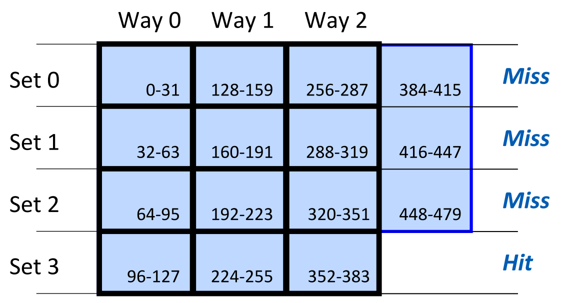
B. Deducing Cache Characteristics from Latency Plots

Most of our cache and TLB parameter measurements use stride accesses to arrays of varying size, with *average* access latency plotted. The basic techniques described in this section are also used to measure CPU cache parameters. We develop variations for instruction caches and shared cache hierarchies.

Figure 4 shows an example of extracting cache size, way size, and line size from an average latency plot. This example assumes an LRU replacement policy, a set-associative cache, and no prefetching. The cache parameters can be deduced from the example plot of Figure 4(a) as follows: As long as the array fits in the cache, the latency remains constant (sizes 384 and below). Once the array size starts exceeding the cache size, latency steps, equal in number to the number of cache sets (four), occur as the sets overflow one by one (sizes 385-512,



(a) Latency Plot for 384-byte, 3-way, 4-set, 32-byte line cache



(b) Array 480 Bytes (15 Lines) in Size

Fig. 4: Three-Way 12-Line Set-Associative Cache and its Latency Plot

cache way size). The increase in array size needed to trigger each step of average latency increase equals the line size (32 bytes). Latency plateaus when all cache sets overflow (size $\geq$16 cache lines). The cache associativity (three) can be found by dividing cache size (384 bytes) by the way size (128 bytes). This calculation does not need the line size nor the number of cache sets. There are other possible ways to compute the four cache parameters, as knowing any three will give the fourth, using $cache\_size = cache\_sets \times line\_size \times associativity$.

Listings 1 and 2 show the structure of our memory microbenchmarks. For each array size and stride, the microbenchmark performs a sequence of dependent reads, with the pre-computed stride access pattern stored in the array, eliminating address computation overhead in the timed inner loop. The stride should be smaller than the cache line size so all steps in the latency plot are observable, but large enough so that transitions between latency steps are not too small to be clearly distinguished.

```
for (i = 0; i < array_size; i++) {
  int t = i + stride;
  if (t >= array_size) t %= stride;
  host_array[i] = (int)device_array + 4*t;
}
cudaMemcpy(device_array, host_array,...);
```

Listing 1: Array Initialization (CPU Code)

```
  int *j = &device_array[0];
  // start timing
  repeat256(j=*(int**)j;) // Macro copy 256 times
  // end timing
```

Listing 2: Sequence of Dependent Reads (GPU Kernel Code)

IV. Tests and Results

This section presents our detailed tests and results. We begin by measuring the latency of the clock() function. We then investigate the SM's various arithmetic pipelines, branch divergence and barrier synchronization. We also explore the memory caching hierarchies both within and surrounding the SMs, as well as memory translation and TLBs.

A. Clock Overhead and Characteristics

All timing measurements use the clock() function, which returns the value of a counter that is incremented every clock cycle [1]. The clock() function translates to a move from the clock register followed by a dependent left-shift by one, suggesting that the counter is incremented at half the shader clock frequency. A clock() followed by a non-dependent operation takes 28 cycles.

The experiment in Figure 5 demonstrates that clock registers are per-TPC. Points in the figure show timestamp values returned by clock() when called at the beginning and end of a block's execution. We see that blocks running on the same TPC share timestamp values, and thus, share clock registers. If clock registers were globally synchronized, the start times of all blocks in a kernel would be approximately the same. Conversely, if the clock registers were per-SM, the start times of blocks within a TPC would not share the same timestamp.

B. Arithmetic Pipelines

Each SM contains three different types of execution units (as shown in Figure 1 and Table I):

- Eight Scalar Processors (SP) that execute single precision floating point and integer arithmetic and logic instructions.
- Two Special Function Units (SFU) that are responsible for executing transcendental and mathematical functions such as reverse square root, sine, cosine, as well as single-precision floating-point multiplication.
- One Double Precision Unit (DPU) that handles computations on 64-bit floating point operands.

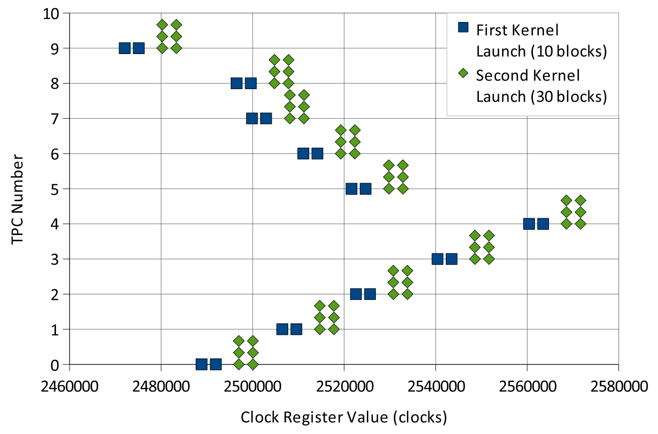


Fig. 5: Timing of two consecutive kernel launches of 10 and 30 blocks. Kernel calls are serialized, showing TPCs have independent clock registers.

	Latency (clocks)	**Throughput (ops/clock)**	**Issue Rate (clocks/warp)**
SP	24	8	4
SFU	28	2	16
DPU	48	1	32

TABLE II: Arithmetic Pipeline Latency and Throughput

Table II shows the latency and throughput for these execution units when all operands are in registers.

To measure the pipeline latency and throughput, we use tests consisting of a chain of dependent operations. For latency tests, we run only one thread. For throughput tests, we run a block of 512 threads (maximum number of threads per block) to ensure full occupancy of the units. Tables III and IV show which execution unit each operation uses, as well as the observed latency and throughput.

Table III shows that single- and double-precision floating-point multiplication and multiply-and-add (*mad*) each map to a single device instruction. However, 32-bit integer multiplication translates to four native instructions, requiring 96 cycles. 32-bit integer *mad* translates to five dependent instructions and takes 120 cycles. The hardware supports only 24-bit integer multiplication via the __mul24() intrinsic.

For 32-bit integer and double-precision operands, division translates to a subroutine call, resulting in high latency and low throughput. However, single-precision floating point division is translated to a short inlined sequence of instructions with much lower latency.

The measured throughput for single-precision floating point multiplication is $\sim$11.2 ops/clock. This is greater than the SP throughput of eight, which suggests that multiplication is issued to both the SP and SFU units. This suggests that each of the two SFUs is capable of doing $\sim$2 multiplications per cycle (4 in total for the 2 SFUs), twice the throughput of other (more complex) instructions that map to the SFU. The throughput for single-precision floating point *mad* is 7.9 ops/clock, suggesting that *mad* operations cannot be executed by the SFUs.

Decuda shows that __sinf(), __cosf(), and __exp2f() intrinsics each translate to a sequence of two dependent instructions operating on a single operand. The Programming Guide states that SFUs execute transcendental operations, however, the latency and throughput measurements for these transcendental instructions do not match those for simpler instructions (e.g., log2f) executed by these units. sqrt() maps to two instructions: a *reciprocal-sqrt* followed by a *reciprocal*.

Figure 6 shows latency and throughput of dependent SP instructions (integer additions), as the number of warps on the SM increases. Below six concurrent warps, the observed latency is 24 cycles. Since all warps observe the same latency, the warp scheduler is fair. Throughput increases linearly while the pipeline is not full, then saturates at eight (number of SP units) operations per clock once the pipeline is full. The Programming Guide states that six warps (192 threads) should be sufficient to hide register read-after-write latencies. However, the scheduler does not manage to fill the pipeline when there are six or seven warps in the SM.

Operation	**Type**	**Execution Unit**	**Latency (clocks)**	**Throughput (ops/clock)**
add, sub, max, min	uint, int	SP	24	7.9
mad	uint, int	SP	120	1.4
mul	uint, int	SP	96	1.7
div	uint	–	608	0.28
div	int	–	684	0.23
rem	uint	–	728	0.24
rem	int	–	784	0.20
and, or, xor, shl, shr	uint	SP	24	7.9

Operation	**Type**	**Execution Unit**	**Latency (clocks)**	**Throughput (ops/clock)**
add, sub, max, min	float	SP	24	7.9
mad	float	SP	24	7.9
mul	float	SP, SFU	24	11.2
div	float	–	137	1.5

Operation	**Type**	**Execution Unit**	**Latency (clocks)**	**Throughput (ops/clock)**
add, sub, max, min	double	DPU	48	1.0
mad	double	DPU	48	1.0
mul	double	DPU	48	1.0
div	double	–	1366	0.063

TABLE III: Latency and Throughput of Arithmetic and Logic Operations

Operation	**Type**	**Execution Unit**	**Latency (clocks)**	**Throughput (ops/clock)**
__umul24()	uint	SP	24	7.9
__mul24()	int	SP	24	7.9
__usad()	uint	SP	24	7.9
__sad()	int	SP	24	7.9
__umulhi()	uint	–	144	1.0
__mulhi()	int	–	180	0.77
__fadd_rn(), __fadd_rz()	float	SP	24	7.9
__fmul_rn(), __fmul_rz()	float	SP, SFU	26	10.4
__fdividef()	float	–	52	1.9
__dadd_rn()	double	DPU	48	1.0
__sinf(), __cosf()	float	SFU?	48	2.0
__tanf()	float	–	98	0.67
__exp2f()	float	SFU?	48	2.0
__expf(), __exp10f()	float	–	72	2.0
__log2f()	float	SFU	28	2.0
__logf(), __log10f()	float	–	52	2.0
__powf()	float	–	75	1.0
rsqrt()	float	SFU	28	2.0
sqrt()	float	SFU	56	2.0

TABLE IV: Latency and Throughput of Mathematical Intrinsics. A "–" in the Execution Unit column denotes an operation that maps to a multi-instruction routine.

C. Control Flow

1) Branch Divergence: All threads of a warp execute a single common instruction at a time. The Programming Guide states that when threads of a warp diverge due to a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path [1]. Our observations are consistent with the expected behavior. Figure 7 shows the measured execution timeline for two concurrent warps in a block whose threads diverge 32 ways. Each thread takes a different path based on its thread ID and performs a sequence of arithmetic operations. The figure shows

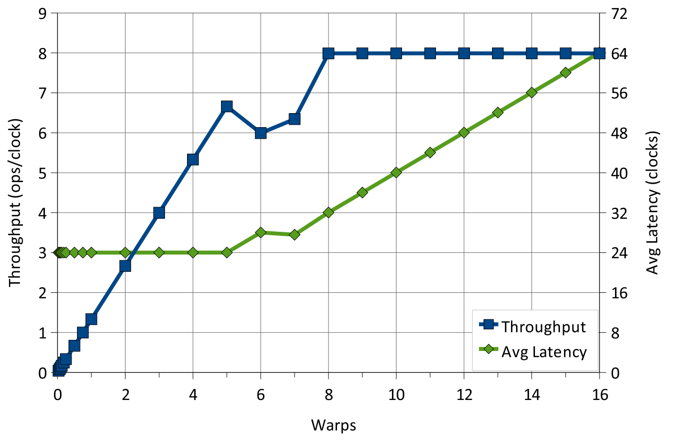


Fig. 6: SP Throughput and Latency. Having six or seven warps does not fully utilize the pipeline.

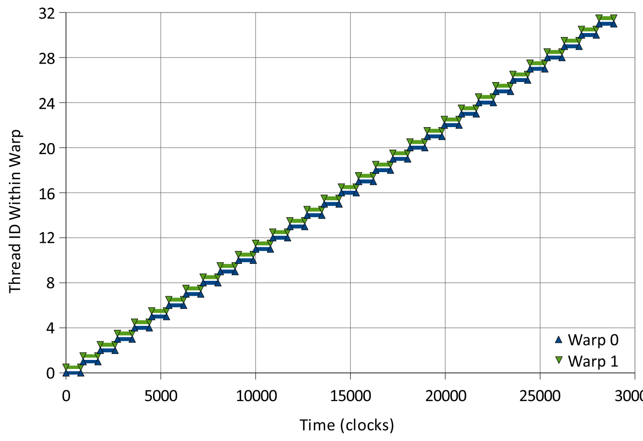


Fig. 7: Execution Timeline of Two 32-Way Divergent Warps. Top series shows timing for Warp 0, bottom for Warp 1.

that within a single warp, each path is executed serially, while the execution of different warps may overlap. Within a warp, threads that take the same path are executed concurrently.

2) Reconvergence: When the execution of diverged paths is complete, the threads converge back to the same execution path. Decuda shows that the compiler inserts one instruction before a potentially-diverging branch, which provides the hardware with the location of the reconvergence point. Decuda also shows that the instruction at the reconvergence point is marked using

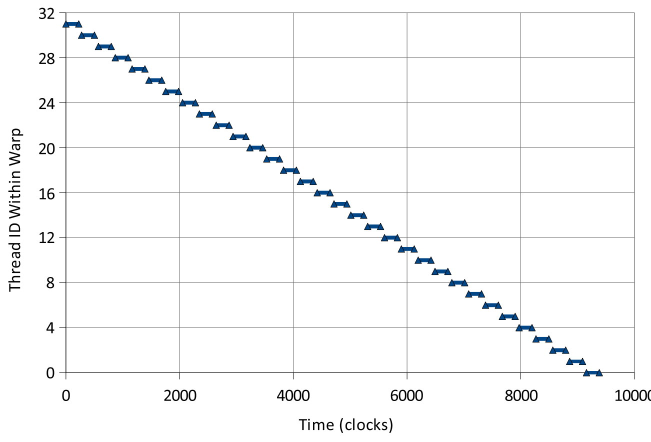


Fig. 8: Execution Timeline of Kernel Shown in Listing 3. Array c contains the increasing sequence {0, 1, ..., 31}

a field in the instruction encoding. We observe that when threads diverge, the execution of each path is serialized up to the reconvergence point. Only when one path reaches the reconvergence point does the other path begin executing.

According to Lindholm et al., a branch synchronization stack is used to manage independent threads that diverge and converge [6]. We use the kernel shown in Listing 3 to confirm this statement. The array `c` contains a permutation of the set of numbers between 0 and 31 specifying the thread execution order. We observe that when a warp reaches a conditional branch, the taken path is always executed first: for each *if* statement the *else* path is the taken path and is executed first, so that the last then-clause (`else if (tid == c[31])`) is always executed first, and the first then-clause (`if (tid == c[0])`) is executed last.

```
if (tid == c[0]) { ... }
else if (tid == c[1]) { ... }
else if (tid == c[2]) { ... }
...
else if (tid == c[31]) { ... }
```

Listing 3: Reconvergence Stack Test

```
int __shared__ sharedvar = 0;
while (sharedvar != tid);
/*** reconvergence point ***/
sharedvar++;
```

Listing 4: Example code that breaks due to SIMT behavior.

Figure 8 shows the execution timeline of this kernel when the array `c` contains the increasing sequence {0, 1, ..., 31}. In this case, thread 31 is the first thread to execute. When the array `c` contains the decreasing sequence {31, 30, ..., 0}, thread 0 is the first to execute, showing that the thread ID does not affect execution order. The observed execution ordering is consistent with the taken path being executed first, and the fall-through path being pushed on a stack. Other tests show that the number of active threads on a path also has no effect on which path is executed first.

3) Effects of Serialization due to SIMT: The Programming Guide states that for correctness, the programmer can ignore the SIMT behavior. In this section, we show an example of code that would work if threads were independent, but deadlocks due to the SIMT behavior. In Listing 4, if threads were independent, the first thread would break out of the while loop and increment `sharedvar`. This would cause each consecutive thread to do the same: fall out of the while loop and increment `sharedvar`, permitting the next thread to execute. In the SIMT model, branch divergence occurs when thread 0 fails the while-loop condition. The compiler marks the reconvergence point just before `sharedvar++`. When thread 0 reaches the reconvergence point, the other (serialized) path is executed. Thread 0 cannot continue and increment `sharedvar` until the rest of the threads also reach the reconvergence point. This causes deadlock as these threads can never reach the reconvergence point.

D. Barrier Synchronization

Synchronization between warps of a single block is done using __syncthreads(), which acts as a barrier. __syncthreads()

is implemented as a single instruction with a latency of 20 clock cycles for a single warp executing a sequence of __syncthreads().

The Programming Guide recommends that __syncthreads() be used in conditional code only if the condition evaluates identically across the entire thread block. The rest of this section investigates the behavior of __syncthreads() when this recommendation is violated. We demonstrate that __syncthreads() operates as a barrier for *warps*, not threads. We show that when threads of a warp are serialized due to branch divergence, any __syncthreads() on one path does not wait for threads from the other path, but only waits for other warps running within the same thread block.

1) __syncthreads() for Threads of a Single Warp: The Programming Guide states that __syncthreads() acts as a barrier for all *threads* in the same block. However, the test in Listing 5 shows that __syncthreads() acts as a barrier for all *warps* in the same block. This kernel is executed for a single warp, in which the first half of the warp produces values in shared memory for the second half to consume.

If __syncthreads() waited for all *threads* in a block, the two __syncthreads() in this example would act as a common barrier, forcing the producer threads (first half of the warp) to write values before the consumer threads (second half of the warp) read them. In addition, since branch divergence serializes execution of divergent warps (See Section IV-C1), a kernel would deadlock whenever __syncthreads() is used within a divergent warp (In this example, one set of 16 threads would wait for the other serialized set of 16 threads to reach its __syncthreads() call). We observe that there is no deadlock, and that the second half of the warp does not read the updated values in the array `shared_array` (The *else* clause executes first, see Section IV-C1), showing that __syncthreads() does not synchronize diverged threads within one warp as the Programming Guide's description might suggest.

```
if (tid < 16)  {
  shared_array[tid] = tid;
  __syncthreads();
}
else  {
  __syncthreads();
  output[tid] =
    shared_array[tid%16];
}
```

Listing 5: Example code that shows __syncthreads() synchronizes at warp granularity

```
// Test run with two warps
count = 0;
if (warp0)  {
  __syncthreads();
  count = 1;
}
else  {
   while (count == 0);
}
```

Listing 6: Example code that deadlocks due to __syncthreads(). Test is run with two warps.

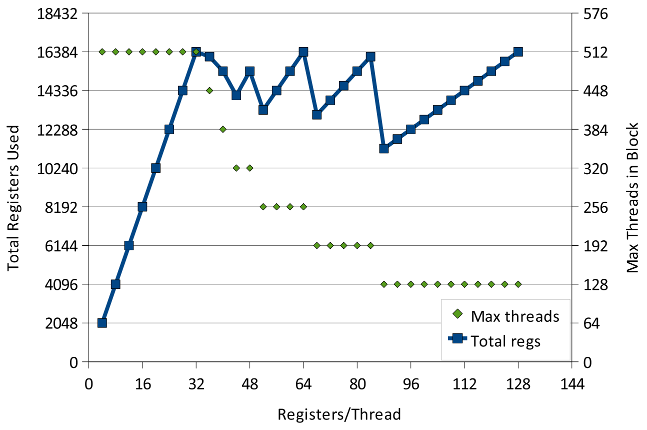


Fig. 9: Total registers used by a block is limited to 16,384 (64 KB). Maximum number of threads in a block is quantized to 64 threads when limited by register file capacity.

```
if (warp0)  {
  // Two-way divergence
  if (tid < 16)
    __syncthreads(); [1]
  else
    __syncthreads(); [2]
}
if (warp1)  {
  __syncthreads();   [3]
  __syncthreads();   [4]
}
```

Listing 7: Example code that produces unintended results due to __syncthreads()

2) __syncthreads() Across Multiple Warps: __syncthreads() is a barrier that waits for all warps to either call __syncthreads() or terminate. If there is a warp that neither calls __syncthreads() nor terminates, __syncthreads() will wait indefinitely, suggesting the lack of a time-out mechanism. Listing 6 shows one example of such a deadlock (with no branch divergence) where the second warp spins waiting for data generated after the __syncthreads() by the first warp.

Listing 7 illustrates the details of the interaction between __syncthreads() and branch divergence. Given that __syncthreads() operates at a warp granularity, one would expect that either the hardware would ignore __syncthreads() inside divergent warps, or that divergent warps participate in barriers in the same way as warps without divergence. We show that the latter is true.

In this example, the second __syncthreads() synchronizes with the third, and the first with the fourth (For warp 0, code block 2 executes before code block 1 because block 2 is the branch's taken path, see Section IV-C1). This confirms that __syncthreads() operates at the granularity of warps and that diverged warps are no exception. Each serialized path executes __syncthreads() separately (Code block 2 does not wait for 1 at the barrier). It waits for all other warps in the block to also execute __syncthreads() or terminate.

E. Register File

We confirm that the register file contains 16,384 32-bit registers (64 KB), as the Programming Guide states [1]. The number of registers used by a thread is rounded up to a multiple of four [4]. Attempting to launch kernels that use more than 128 registers per thread or a total of more than 64 KB of registers in a block results in a failed launch. In Figure 9, below 32 registers per thread, the register file cannot be fully utilized because the maximum number of threads allowed per block is 512. Above 32 registers per thread, the register file capacity limits the number of threads that can run in a block. Figure 9 shows that when limited by register file capacity, the maximum number of threads in a block is quantized to 64 threads. This puts an additional limit on the number of registers that can be used by one kernel, and is most visible when threads use 88 registers each: Only 128 threads can run in a block and only 11,264 (128 threads $\times$ 88) registers can be used, utilizing only 69% of the register file.

The quantizing of threads per block to 64 threads suggests that each thread's registers are distributed to one of 64 logical "banks". Each bank is the same size, so each bank can fit the same number of threads, limiting threads to multiples of 64 when limited by register file capacity. Note that this is different from quantizing the total register use.

Because all eight SPs always execute the same instruction at any given time, a physical implementation of 64 logical banks can share address lines among the SPs and use wider memory arrays instead of 64 real banks. Having the ability to perform four register accesses per SP every clock cycle (four logical banks) provides sufficient bandwidth to execute three-read, one-write operand instructions (e.g., multiply-add) every clock cycle. A thread would access its registers over multiple cycles, since they all reside in a single bank, with accesses for multiple threads occurring simultaneously.

Having eight logical banks per SP could provide extra bandwidth for the "dual-issue" feature using the SFUs (see Section IV-B) and for performing memory operations in parallel with arithmetic.

The Programming Guide alludes to preferring multiples of 64 threads by suggesting that to avoid bank conflicts, "best results" are achieved if the number of threads per block is a multiple of 64. We observe that when limited by register count, the number of threads per block is *limited* to a multiple of 64, while no bank conflicts were observed.

F. Shared Memory

Shared memory is a non-cached per-SM memory space. It is used by threads of a block to cooperate by sharing data with other threads from the same block. The amount of shared memory allowed per block is 16 KB. The kernel's function parameters also occupy shared memory, thus slightly reducing the usable memory size.

We measure the read latency to be 38 cycles using stride accesses as in Listings 1 and 2. Volkov and Demmel reported a similar latency of 36 cycles on the 8800GTX, the predecessor of the GT200 [7]. The Programming Guide states that shared memory latency is comparable to register access latency. Varying the memory footprint and stride of our microbenchmark verified the lack of caching for shared memory.

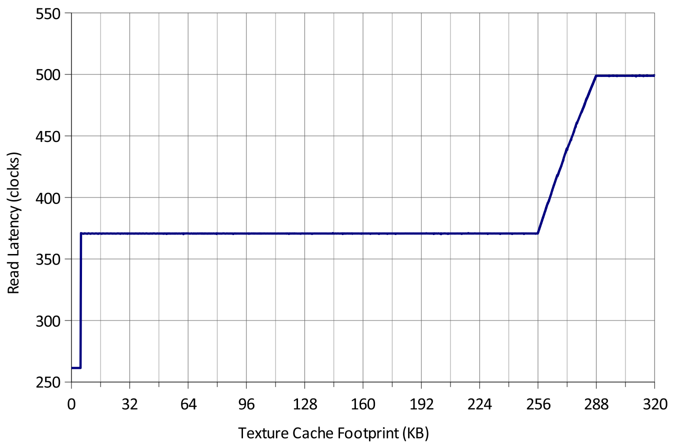


Fig. 10: Texture Memory. 5 KB L1 and 256 KB, 8-way L2 caches. Measured using 64-byte stride.

G. Global Memory

Global memory is accessible by all running threads, even if they belong to different blocks. Global memory accesses are uncached and have a documented latency of 400-600 cycles [1]. Our microbenchmark executes a sequence of pointer-chasing dependent reads to global memory, similar to Listings 1 and 2. In the absence of a TLB miss, we measure a read latency in the range of 436-443 cycles. Section IV-I2 presents more details on the effects of memory translation on global memory access latency. We also investigated the presence of caches. No caching effects were observed.

H. Texture Memory

Texture memory is a cached, read-only, globally-visible memory space. In graphics rendering, textures are often two-dimensional and exhibit two-dimensional locality. CUDA supports one-, two-, and three-dimensional textures. We measure the cache hierarchy of the one-dimensional texture bound to a region of linear memory. Our code performs dependent texture fetches from a texture, similar to Listings 1 and 2. Figure 10 shows the presence of two levels of texture caching using a stride of 64 bytes, showing 5 KB and 256 KB for L1 and L2 cache sizes, respectively.

We expect the memory hierarchy for higher-dimension (2D and 3D) textures not to be significantly different. 2D spatial locality is typically achieved by rearranging texture elements in "tiles" using an address computation, rather than requiring specialized caches [8]–[10].

1) Texture L1 Cache: The texture L1 cache is 5 KB 20-way set-associative with 32 byte cache lines. Figure 11 focuses on the first latency increase at 5 KB and shows results with an eight byte stride. A 256-byte way size for a 5 KB cache implies 20-way set associativity. We see that the L1 hit latency (261 clocks) is more than half that of main memory (499 clocks), consistent with the Programming Guide's statement that texture caches do not reduce fetch latency but do reduce DRAM bandwidth demand.

2) Texture L2 Cache: The texture L2 cache is 256 KB 8-way set associative with 256-byte cache lines. Figure 10 shows a way size of 32 KB for a 256 KB cache, implying 8-way set

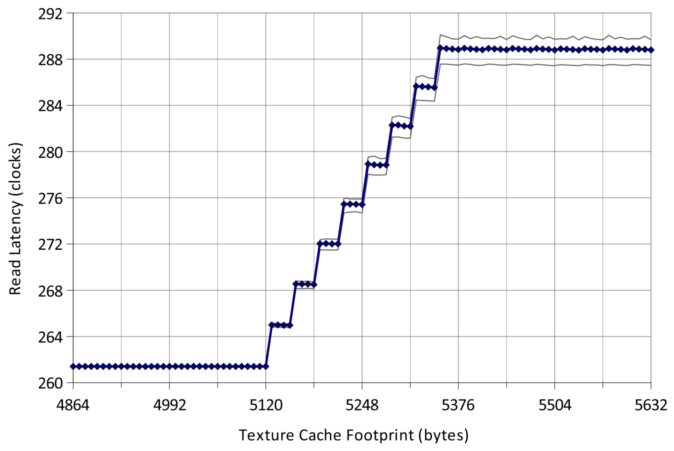


Fig. 11: Texture L1 Cache. 5 KB, 20-way, 32-byte lines. Measured using 8-byte stride. Maximum and minimum average latency over all TPC placements are also shown: L2 has TPC placement-dependent latency.

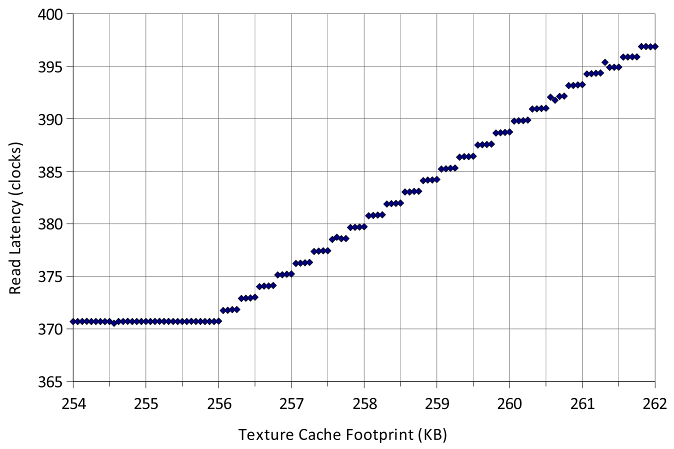


Fig. 12: Texture L2 Cache. 256 KB, 8-way, 256-byte lines. Measured using 64-byte stride.

associativity. Figure 12 zooms in on the previous graph near 256 KB to reveal the presence of latency steps that indicate a 256-byte cache line size. We can also see in Figure 11 that the L2 texture cache has TPC placement-dependent access times, suggesting the L2 texture cache does not reside within the TPC.

I. *Memory Translation*

We investigate the presence of TLBs using stride-accessed dependent reads, similar to Listings 1 and 2. Measuring TLBs parameters is similar to measuring caches, but with increased array sizes and larger strides comparable to the page size. Detailed TLB results for both global and texture memory are presented in Sections IV-I1 and IV-I2 respectively.

1) Global Memory Translation: Figure 13 shows that there are two TLB levels for global memory. The L1 TLB is fully-associative, holding mappings for 8 MB of memory, containing 16 lines with a 512 KB TLB line size. The 32 MB L2 TLB is 8-way set associative, with a 4 KB line size. We use the term *TLB size* to refer to the total size of the pages that can be mapped by the TLB, rather than the raw size of the entries stored in the TLB. For example, an 8 MB TLB describes a TLB that can cache 2 K mappings when the page size is 4 KB. If,

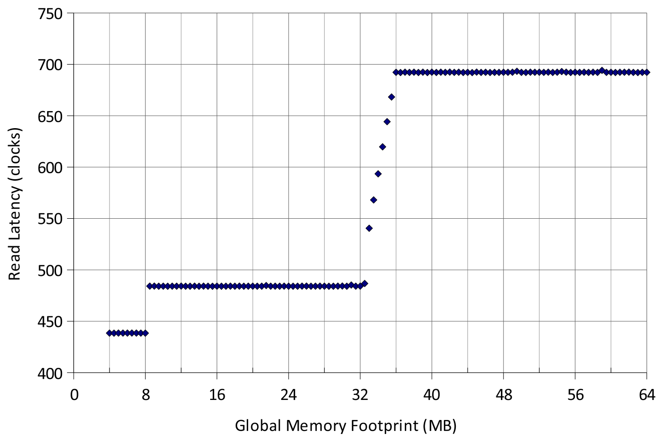


Fig. 13: Global Memory. 8 MB fully-associative L1 and 32 MB 8-way L2 TLBs. Measured using 512 KB stride.

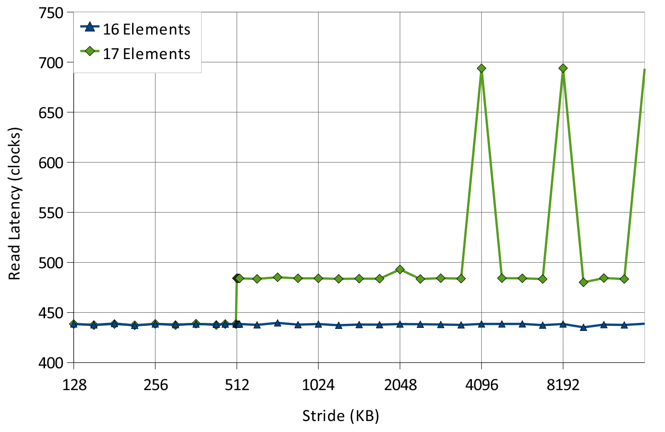


Fig. 14: Global L1 TLB. 16-way fully-associative with 512 KB line size.

in addition, the TLB line size were 512 KB, the TLB would be organized in 16 lines with 128 mappings of consecutive pages per line.

In Figure 13, the first latency plateau at ∼440 cycles indicates an L1 TLB hit (global memory read latency as measured in Section IV-G). The second plateau at ∼487 cycles indicates an L2 TLB hit, while an L2 TLB miss takes ∼698 cycles. We measure the 16-way associativity of the L1 TLB by accessing a fixed number of elements with varying strides. Figure 14 depicts the results when 16 and 17 array elements are accessed. For large strides, where all elements map to same cache set (e.g., 8 MB), accessing 16 elements always experiences L1 TLB hits, while accessing 17 elements will miss the L1 TLB at 512 KB stride and up. We can also see that the L1 TLB has only one cache set, implying it is fully-associative with 512 KB lines. If there were at least two sets, then when the stride is not a power of two and is greater than 512 KB (the size of a cache way), some elements would map to different sets. When 17 elements are accessed, they would not be all mapped to the same set, and there would be some stride for which no L1 misses occur. We never see L1 TLB hits for strides beyond 512 KB (e.g., 608, 724, and 821 KB). However, we can see (at strides beyond 4 MB) that the L2 TLB is not fully-associative.

Figure 13 shows that the size of an L2 TLB way is 4 MB (32

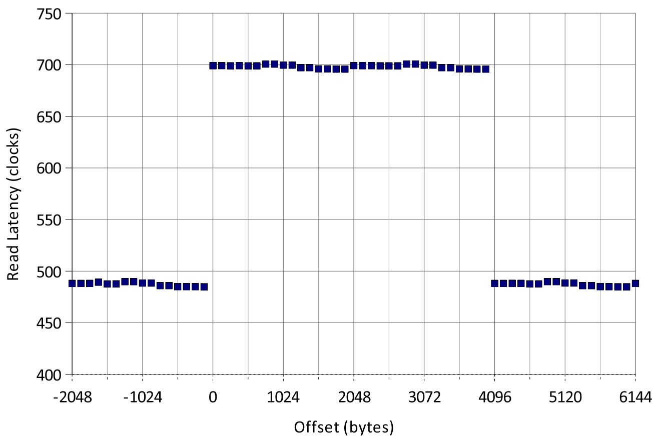


Fig. 15: Global L2 TLB. 4 KB TLB line size.

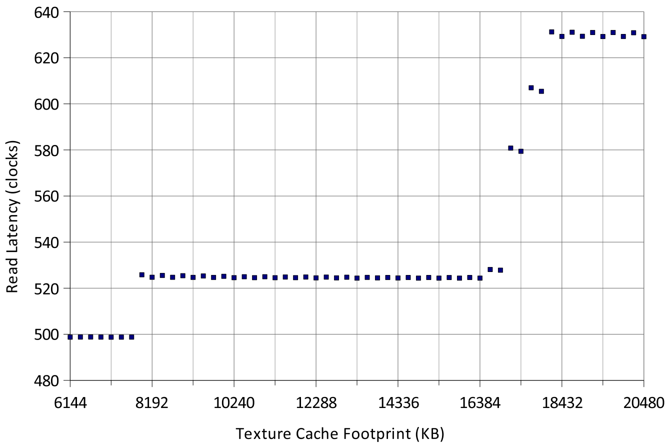


Fig. 16: Texture Memory. 8 MB fully-associative L1 TLB and 16 MB 8-way L2 TLB. 544 clocks L1 and 753 clocks L2 TLB miss. Measured using 256 KB stride.

to 36 MB; see Section III-B). With an L2 TLB size of 32 MB, the associativity of the L2 TLB is eight. Extending the test did not find evidence of multi-level paging.

Although the L1 TLB line size is 512 KB, the L2 TLB line size is a smaller 4 KB. We used a microbenchmark that uses two sets of 10 elements (20 total) with each element separated by a 2 MB stride. The two sets of elements are separated by 2 MB+*offset*. The i^{th} element has address $(i < 10) ? (i \times 2\ MB) : (i \times 2\ MB + \mathit{offset})$. We need to access more than 16 elements to prevent the 16-way L1 TLB from hiding the accesses. Since the size of an L2 TLB way is 4 MB and we use 2 MB strides, our 20 elements map to two L2 sets when *offset* is zero. Figure 15 shows the 4 KB L2 TLB line size. When *offset* is zero, our 20 elements occupy two sets, 10 elements per set, causing conflict misses in the 8-way associative L2 TLB. As *offset* is increased beyond the 4 KB L2 TLB line size, having 5 elements per set no longer causes conflict misses in the L2 TLB.

Although the page size can be less than the 4 KB L2 TLB line size, we believe a 4 KB page size is a reasonable choice. We note that the Intel x86 architecture uses multi-level paging with mainly 4 KB pages, while Intel's family of GPUs uses single-level 4 KB paging [9], [11].

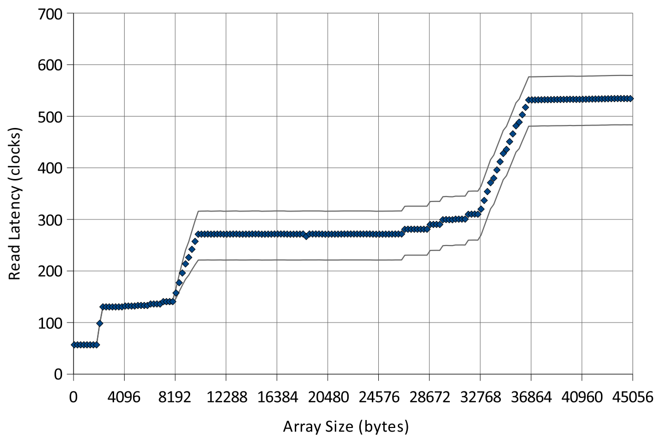


Fig. 17: Constant Memory. 2 KB L1, 8 KB 4-way L2, 32 KB 8-way L3 caches. Measured using 256-byte stride. Maximum and minimum average latency over all TPC placements are also shown: L3 has TPC placement-dependent latency.

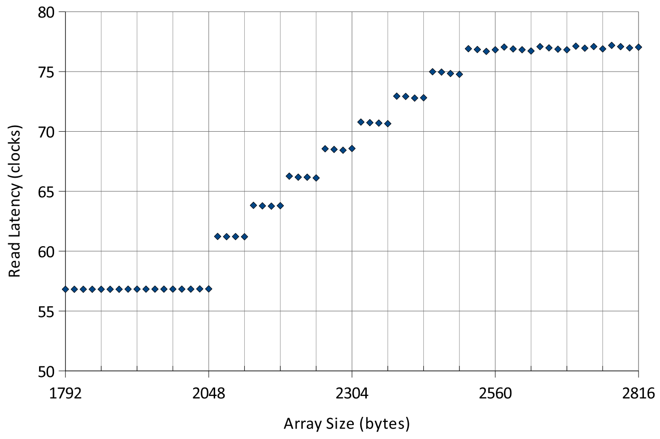


Fig. 18: Constant L1 cache. 2 KB, 4-way, 64-byte lines. Measured using 16-byte stride.

2) Texture Memory Translation: We used the same methodology as in Section IV-I1 to compute the configuration parameters of the texture memory TLBs. The methodology is not repeated here for the sake of brevity. Texture memory contains two levels of TLBs, with 8 MB and 16 MB of mappings, as seen in Figure 16 with 256 KB stride. The L1 TLB is 16-way fully-associative with each line holding translations for 512 KB of memory. The L2 TLB is 8-way set associative with a line size of 4 KB. At 512 KB stride, the virtually-indexed 20-way L1 texture cache hides the features of the L1 texture TLB. The access latencies as measured with 512 KB stride are 497 (TLB hit), 544 (L1 TLB miss), and 753 (L2 TLB miss) clocks.

J. Constant Memory

There are two segments of constant memory: one is user-accessible, while the other is used by compiler-generated constants (e.g., comparisons for branch conditions) [4]. The user-accessible segment is limited to 64 KB.

The plot in Figure 17 shows three levels of caching of sizes 2 KB, 8 KB, and 32 KB. The measured latency includes the latency of two arithmetic instructions (one address computation and one load), so the raw memory access time would be roughly 48 cycles lower (8, 81, 220, and 476 clocks for an L1 hit, L2

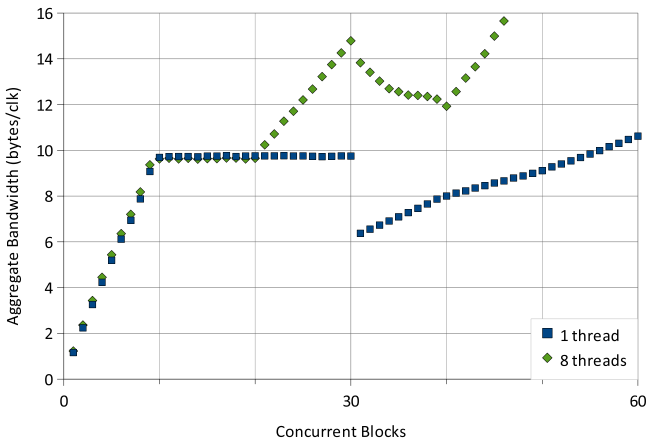


Fig. 19: Constant L3 cache bandwidth. 9.75 bytes per clock.

hit, L3 hit, and L3 miss, respectively). Our microbenchmarks perform dependent constant memory reads, similar to Listings 1 and 2.

1) Constant L1 Cache: A 2 KB L1 constant cache is located in each SM (See Section IV-J4). The L1 has a 64-byte cache line size, and is 4-way set associative with eight sets. The way size is 512 bytes, indicating 4-way set associativity in a 2 KB cache. Figure 18 shows these parameters.

2) Constant L2 Cache: An 8 KB L2 constant cache is located in each TPC and is shared with instruction memory (See Sections IV-J4 and IV-J5). The L2 cache has a 256-byte cache line size and is 4-way set associative with 8 sets. The region near 8,192 bytes in Figure 17 shows these parameters. A 2 KB way size in an 8 KB cache indicates an associativity of four.

3) Constant L3 Cache: We observe a 32 KB L3 constant cache shared among all TPCs. The L3 cache has 256-byte cache lines and is 8-way set associative with 16 sets. We observe cache parameters in the region near 32 KB in Figure 17. The minimum and maximum access latencies for the L3 cache (Figure 17, 8-32 KB region) differ significantly depending on which TPC executes the test code. This suggests that the L3 cache is located on a non-uniform interconnect that connects TPCs to L3 cache and memory. The latency variance does not change with increasing array size, even when main memory is accessed (array size $>$ 32 KB), suggesting that L3 cache is located near the main memory controllers.

We also measure the L3 cache bandwidth. Figure 19 shows the aggregate L3 cache read bandwidth when a varying number of blocks make concurrent L3 cache read requests, with the requests within each thread being independent. The observed aggregate bandwidth of the L3 constant cache is $\sim$9.75 bytes/clock when running between 10 and 20 blocks.

We run two variants of the bandwidth tests: one variant using one thread per block and one using eight to increase constant cache fetch demand within a TPC. Both tests show similar behavior below 20 blocks. This suggests that when running one block, an SM is only capable of fetching $\sim$1.2 bytes/clock even with increased demand within the block (from multiple threads). The measurements are invalid above 20 blocks in the

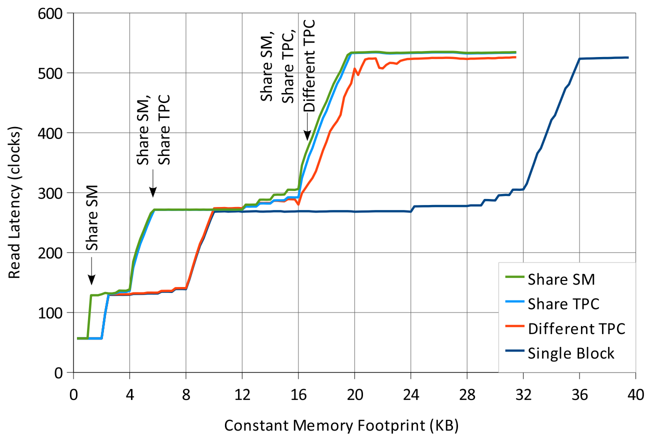


Fig. 20: Constant Memory Sharing. Per-SM L1 cache, per-TPC L2, global L3. Measured using 256-byte stride.

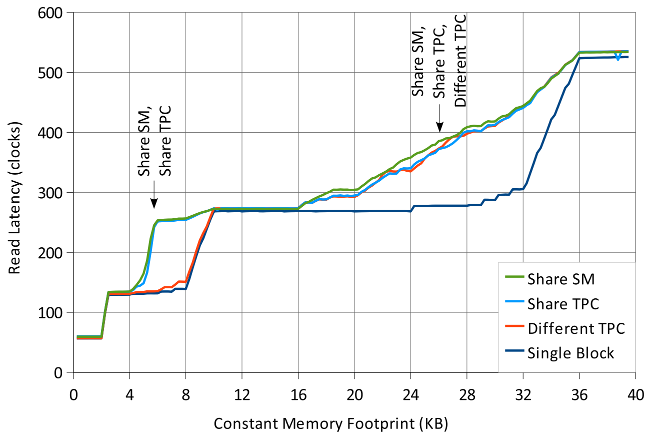


Fig. 21: Constant Memory Instruction Cache Sharing. L2 and L3 caches are shared with instructions. Measured using 256-byte stride.

eight-thread case, as there are not enough unique data sets and the per-TPC L2 cache hides some requests from the L3, causing apparent aggregate bandwidth to increase. Above 30 blocks, some SMs run more than one block causing load imbalance.

4) Cache Sharing: The L1 constant cache is private to each SM, the L2 is shared among SMs on a TPC, and the L3 is global. This was tested by measuring latency using two concurrent blocks with varying placement (same SM, same TPC, two different TPCs). The two blocks will compete for shared caches, causing the observed cache size to be halved. Figure 20 shows the results of this test. In all cases, the observed cache size is halved to 16 KB (L3 is global). With two blocks placed on the same TPC the observed L2 cache size is halved to 4 KB (L2 is per-TPC). Similarly, with two blocks on the same SM, the observed L1 cache size is halved to 2 KB (L1 is per-SM).

5) Cache Sharing with Instruction Memory: It has been suggested that part of the constant cache and instruction cache hierarchies are unified [12], [13]. We find that the L2 and L3 caches are indeed instruction *and* constant caches, while the L1 caches are single-purpose. Similar to Section IV-J4, we measure the interference between instruction fetches and constant cache fetches with varying placements. The result is plotted in Figure 21. The L1 access times are not affected by instruction fetch

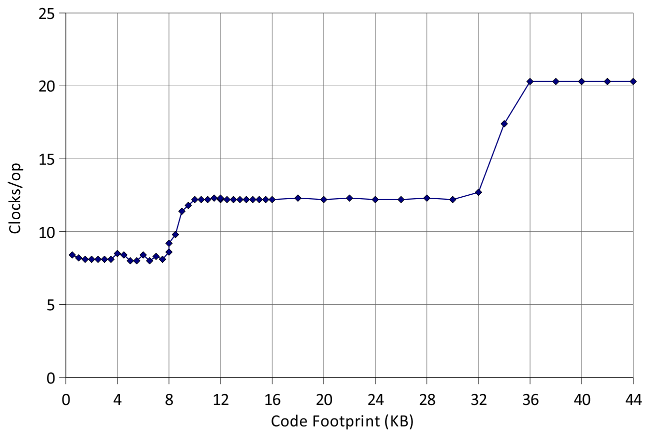


Fig. 22: Instruction Cache Latency. 8 KB 4-way L2, 32 KB 8-way L3. This test fails to detect the 4 KB L1 cache.

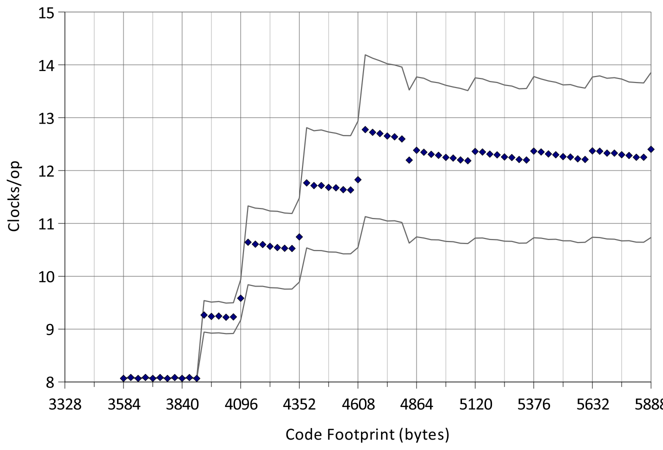


Fig. 23: Instruction L1 cache. 4 KB, 4-way, 256-byte lines. Contention for the L2 cache is added to make L1 misses visible. Maximum and minimum average latency over all TPC placements are also shown: L3 has TPC placement-dependent latency.

demand even when the blocks run on the same SM, so the L1 caches are single-purpose.

K. Instruction Supply

We detect three levels of instruction caching, of sizes 4 KB, 8 KB, and 32 KB, respectively (plotted in Figure 22). The microbenchmark code consists of differently-sized blocks of independent 8-byte arithmetic instructions (abs) to maximize fetch demand. The 8 KB L2 and 32 KB L3 caches are visible in the figure, but the 4 KB L1 is not, probably due to a small amount of instruction prefetching that hides the L2 access latency.

1) L1 Instruction Cache: The 4 KB L1 instruction cache resides in each SM, with 256-byte cache lines and 4-way set associativity.

The L1 cache parameters were measured (Figure 23) by running concurrent blocks of code on the other two SMs on the same TPC to introduce contention for the L2 cache, so L1 misses beginning at 4 KB do not stay hidden as in Figure 22. The 256-byte line size is visible, as well as the presence of 4 cache sets. The L1 instruction cache is per-SM. When other SMs on the same TPC flood their instruction cache hierarchies,

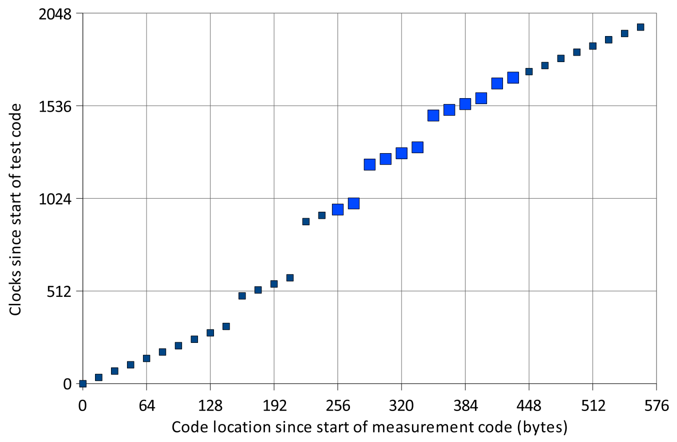


Fig. 24: Instruction fetch size. The SM appears to fetch from the L1 cache in blocks of 64 bytes. The code spans three 256-byte cache lines, with boundaries at 160 and 416 bytes.

the instruction cache size of 4 KB for the SM being observed does not decrease.

2) L2 Instruction Cache: The 8 KB L2 instruction cache is located on the TPC, with 256-byte cache lines and 4-way set associativity. We have shown in Section IV-J5 that the L2 instruction cache is also used for constant memory. We have verified that the L2 instruction cache parameters match those of the L2 constant cache, but omit the results due to space constraints.

3) L3 Instruction Cache: The 32 KB L3 instruction cache is global, with 256-byte cache lines and 8-way set associativity. We have shown in Section IV-J5 that the L3 instruction cache is also used for constant memory, and have verified that the cache parameters for instruction caches match those for constant memory.

4) Instruction Fetch: The SM appears to fetch instructions from the L1 instruction cache 64 bytes at a time (8-16 instructions). Figure 24 shows the execution timeline of our measurement code, consisting of 36 consecutive `clock()` reads (72 instructions), averaged over 10,000 executions of the microbenchmark.

While one warp runs the measurement code, seven "evicting warps" running on the same SM repeatedly evict the region indicated by the large points in the plot, by looping through 24 instructions (192 bytes) that cause conflict misses in the instruction cache. The evicting warps will repeatedly evict a cache line used by the measurement code with high probability, depending on warp scheduling. The cache miss latency caused by an eviction will be observed only at instruction fetch boundaries (160, 224, 288, 352, and 416 bytes in Figure 24).

We see that a whole cache line (spanning the code region 160-416 bytes) is evicted when a conflict occurs and that the effect of a cache miss is only observed across, but not within, blocks of 64 bytes.

V. Related Work

Microbenchmarking has been used extensively in the past to determine the hardware organization of various processor structures. We limit our attention to work targeting GPUs.

Volkov and Demmel benchmarked the 8800GTX GPU, the predecessor of the GT200 [7]. They measured characteristics of the GPU relevant to accelerating dense linear algebra, revealing the structure of the texture caches and one level of TLBs. Although they used the previous generation hardware, their measurements generally agree with ours. We focus on the microarchitecture of the GPU, revealing an additional TLB level and caches, and the organization of the processing cores.

GPUs have also been benchmarked for performance analysis. An example is GPUBench [14], a set of microbenchmarks written in the OpenGL ARB shading language that measures some of the GPU instruction and memory performance characteristics. The higher-level ARB shading language is further abstracted from the hardware than CUDA, making it difficult to infer detailed hardware structures from the results. However, the ARB shading language offers vendor-independence, which CUDA does not.

Currently, specifications of Nvidia's GPUs and CUDA optimization techniques come from the manufacturer [1], [3]. Studies on optimization (e.g., [15]) as well as performance simulators (e.g., [2]) rely on these published specifications. We present more detailed parameters, which we hope will be useful in improving the accuracy of these studies.

VI. Summary and Conclusions

This paper presented our analysis of the Nvidia GT200 GPU and our measurement techniques. Our suite of microbenchmarks revealed architectural details of the processing cores and the memory hierarchies. A GPU is a complex device, and it is impossible that we reverse-engineer every detail. We believe we have investigated an interesting subset of features. Table V summarizes our architectural findings.

Our results validated some of the hardware characteristics presented in the CUDA Programming Guide [1], but also revealed the presence of some undocumented hardware structures such as mechanisms for control flow and caching and TLB hierarchies. In addition, in some cases our findings deviated from the documented characteristics (e.g., texture and constant caches).

We also presented our techniques for our architectural analysis. We believe that these techniques will be useful for the analysis of other GPU-like architectures and validation of GPU-like performance models.

The ultimate goal is to know the hardware better, so that we can harvest its full potential.

Arithmetic Pipeline

	Latency (clocks)	Throughput (ops/clock)
SP	24	8
SFU	28	2 (4 for MUL)
DPU	48	1

Pipeline Control Flow

Branch Divergence	Diverged paths are serialized. Reconvergence is handled via a stack.
Barrier Synchronization	__syncthreads() works at warp granularity. Warps wait at the barrier until all other warps execute __syncthreads() or terminate.

Memories

Register File	16 K 32-bit registers, 64 logical banks per SM
Instruction	L1: 4 KB, 256-byte line, 4-way, per-SM L2: 8 KB, 256-byte line, 4-way, per-TPC L3: 32 KB, 256-byte line, 8-way, global L2 and L3 shared with constant memory
Constant	L1: 2 KB, 64-byte line, 4-way, per-SM, 8 clk L2: 8 KB, 256-byte line, 4-way, per-TPC, 81 clk L3: 32 KB, 256-byte line, 8-way, global, 220 clk L2 and L3 shared with instruction memory
Global	∼436-443 cycles read latency 4 KB translation page size L1 TLB: 16 entries, 128 pages/entry, 16-way L2 TLB: 8192 entries, 1 page/entry, 8-way
Texture	L1: 5 KB, 32-byte line, 20-way, 261 clk L2: 256 KB, 256-byte line, 8-way, 371 clk 4 KB translation page size L1 TLB: 16 entries, 128 pages/entry, 16-way L2 TLB: 4096 entries, 1 page/entry, 8-way
Shared	16 KB, 38 cycles read latency

TABLE V: GT200 Architecture Summary

References

[1] 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.

[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 *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, April 2009, pp. 163–174.

[3] Nvidia, "NVIDIA GeForce GTX 200 GPU Architectural Overview," http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf, May 2008.

[4] ——, "The CUDA Compiler Driver NVCC," http://www.nvidia.com/object/io_1213955090354.html.

[5] W. J. van der Laan, "Decuda," http://wiki.github.com/laanwj/decuda/.

[6] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[7] V. Volkov and J. W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[8] Z. S. Hakura and A. Gupta, "The Design and Analysis of a Cache Architecture for Texture Mapping," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 108–120, 1997.

[9] Intel, *G45: Volume 1a Graphics Core, Intel 965G Express Chipset Family and Intel G35 Express Chipset Graphics Controller Programmer's Reference Manual (PRM)*, January 2009.

[10] AMD, *ATI CTM Guide, Technical Reference Manual*.

[11] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, September 2009.

[12] H. Goto, "Gt200 over view," http://pc.watch.impress.co.jp/docs/2008/0617/kaigai_10.pdf, 2008.

[13] D. Kirk and W. W. Hwu, "ECE 489AL Lectures 8-9: The CUDA Hardware Model," http://courses.ece.illinois.edu/ece498/al/Archive/Spring2007/lectures/lecture8-9-hardware.ppt, 2007.

[14] I. Buck, K. Fatahalian, and M. Houston, "GPUBench," http://graphics.stanford.edu/projects/gpubench/.

[15] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2008, pp. 73–82.