Micro-benchmarking the GT200 GPU

Misel-Myrto Papadopoulou Maryam Sadooghi-Alvandi Henry Wong

***Abstract*—Graphics processors (GPU) are interesting for non-graphics parallel computation because of the potential for more than an order of magnitude of speedup over CPUs. Because the GPU is often presented as a C-like abstraction like Nvidia's CUDA, little is known about the hardware architecture of the GPU beyond the high-level descriptions documented by the manufacturer.**

We develop a suite of micro-benchmarks to measure the CUDA-visible architectural characteristics of the Nvidia GT200 (GTX280) GPU. We measure properties of the arithmetic pipelines, the stack-based handling of branch divergence, and the warp-granularity operation of the barrier synchronization instruction. We confirm that global memory is uncached with ∼441 clock cycles of latency, and measure parameters of the three levels of instruction and constant caches and three levels of TLBs.

We succeed in revealing more detail about the GT200 architecture than previously disclosed.

I. Introduction

The graphics processor (GPU) as a non-graphics compute processor has a very different architecture from traditional sequential processors. For both developers of the platform as well as for GPU architecture and compiler researchers, it is essential to understand the architecture of a current design in detail. For the developer, knowing the underlying hardware architecture is essential for tuning a program to run at maximum performance. Compilers must likewise know the properties of the hardware in order to properly optimize code. In architecture research, it is important to understand the trade-offs that were made (and that a trade-off *was* made) in order to accurately model and improve current architectures.

The Nvidia G80 and GT200 GPUs are capable of non-graphics computation using Nvidia's C-like CUDA programming interface. Nvidia provides hints of the GPU performance characteristics in the CUDA Programming Guide [1] in the form of rules, but the rules are vague, and there is little indication of how the underlying hardware is organized and how it motivates the rules.

To learn more about the characteristics of the Nvidia GPU, we constructed a suite of micro-benchmarks targeting specific parts of the architecture. We verified some of the performance characteristics discussed in the Programming Guide, and revealed architectural details beyond what was revealed in the manuals.

We make the following contributions:

- We verify performance characteristics listed in the CUDA Programmer's Reference.
- We explored the detailed functionality of divergent branches and of the barrier synchronization instruction.

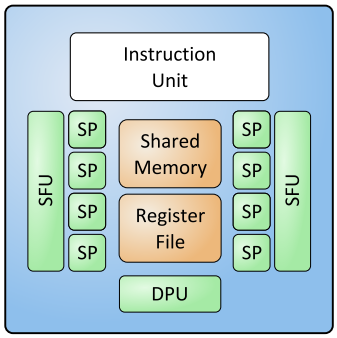

Fig. 1. Streaming Multiprocessor with 8 Scalar Processors Each

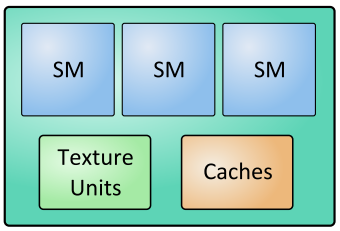

Fig. 2. Thread Processing Cluster with 3 SMs Each

- We measure the structure of the memory caching hierarchy, including the TLB hierarchy, constant memory caches, and instruction memory caches.

We begin with background on the CUDA computation model in Section II. We then describe the architecture of the GT200 GPU in Section III and present our detailed measurements in Section IV. We review related work in Section V.

II. Background

A. GPU Architecture

CUDA models the GPU architecture as a parallel multicore system. It abstracts the thread-level parallelism of the GPU into a hierarchy of threads ("thread", "warp", Cooperative

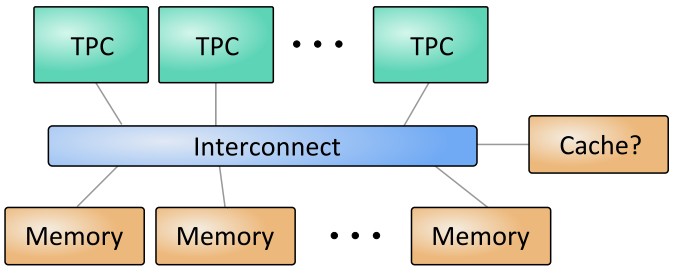

Fig. 3. GPU with TPCs and Memory Banks

SM Resources	
SPs	8 per SM
SFUs	2 per SM
DPUs	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	
SMs	3 per TPC
TPCs	10 total
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	2M PTX insns max

TABLE I
GT200 PARAMETERS ACCORDING TO NVIDIA [1], [2]

Thread Array (CTA) or "block", "grid"). While the programming model uses collections of scalar threads, the hardware more closely resembles an 8-wide processor operating on 32-wide vectors, but allows each channel to branch separately.

These threads are mapped onto a hierarchy of hardware resources. Blocks of threads are executed within Streaming Multiprocessors (SM, Figure 1). The basic unit of execution flow in the SM is the *warp*, a collection of 32 threads. Nvidia refers to this arrangement as Single-Instruction Multiple-Thread (SIMT), where every thread of a warp executes the same instruction in lockstep. 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 contain resources (e.g. caches) that are shared between several SMs, most of which are not visible to the programmer.

The GPU (Figure 3) consists of the collection of TPCs, the interconnection network, and the memory system (DRAM memory controllers).

The parameters Nvidia discloses for the GT200 GPU we used are shown in Table I.

B. Compilation Flow

CUDA presents the GPU architecture as a C-like programming language with extensions to abstract the GPU threading model. In the CUDA model, host CPU code can launch GPU routines by calling *device* functions that execute on the GPU.

Since the GPU uses an instruction set different from the host CPU, the CUDA program uses a compile flow that separates the code into CPU and GPU code, compiles them using different compilers targeting different instruction sets, and then merges the compiled code into a single "fat" binary [3].

A summary of the relevant portions of the compilation flow is as follows:

- C-like CUDA GPU functions separated from Host C functions (**cudafe**)
- GPU code compiled into PTX intermediate representation (**nvopencc**)
- PTX code compiled into native GPU "cubin" binary (**ptxas**)
- Host C code and cubin merged, then compiled into host executable (host C compiler, e.g. gcc)
- Potential for further modification of the GPU cubin by GPU driver at load or run time

Although PTX is described as being the "assembly" level representation of GPU code, we have found that PTX is not useful for detailed optimization or micro-benchmarking. The native instruction set differs too much from PTX, and much optimization is done to the PTX code, so PTX code is not a sufficiently precise 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 cubin level.

We have not observed any modification of the cubin code before execution, but we have no guarantees that such modifications cannot be done. Conceptually, the GPU driver could make changes to the cubin code to emulate unsupported functionality or to work around hardware flaws triggered by specific code sequences.

III. MEASUREMENT METHODOLOGY

To explore the GT200 architecture, we create specially-crafted micro-benchmarks to expose the hardware bottleneck related to each of the characteristics we wish to measure. Our conclusions will be drawn from analyzing the execution times of the micro-benchmarks rather than direct probing of the hardware.

We used decuda [4] to assist in understanding the code sequences generated by the Cuda compiler. Decuda is a disassembler for Nvidia's machine-level instructions, derived from analysis of Nvidia's compiler output, as the instruction set is not publicly documented. Decuda also served as a convenient tool for aligning blocks of code.

```
__global__ void kmulu_dep512(unsigned *ts, int *out, ...)
{
  ...

  for (int i=0;i<2;i++)
  {
    start_time = clock(); // Only measure last iteration
    repeat256(t *= t2; t2 *= t;)  // 512 multiplications
    stop_time = clock();
  }

  ...
}
```

Listing 1. Sample Multiplication Benchmark

The general structure of a micro-benchmark consists of GPU kernel code containing timing code (using clock()) around a section of code that exercises the hardware being measured. The kernel is run enough times so appropriate

averages can be taken. Listing 1 shows a sample kernel used to measure the latency of multiplication operations.

Typically, a benchmark runs through the entire code at least twice, with the first iteration not measured, to avoid the effect of cold instruction cache misses. Benchmarks also write some combination of the internal variables to global memory at the end of the benchmark to prevent the compiler from optimizing away the arithmetic inside the loop. Timing code is done by reading clock() into a register that is later moved to global memory, avoiding global memory writes from interfering with the test.

The remainder of this section describes measurement methodology specific to certain structures in the architecture. Detailed descriptions of the tests and results are in Section IV.

A. Arithmetic Pipeline

Arithmetic performance is measured by timing a block of code that exercises typically the one instruction we are measuring. We ensure that the kernel code fits within the instruction cache. We make each instruction depend on the result of the previous instruction to measure pipeline latency. We use multiple warps of the same code to measure the aggregate throughput.

B. Control Flow and Barrier Synchronization

Control flow behavior is measured by constructing micro-benchmarks of varying branching behavior, including branch divergence. The order of execution of code blocks of interest is observed using the clock() function. Other behaviors were derived from whether the GPU deadlocked on the given kernel code. A similar approach was used to investigate the syncthreads barrier synchronization instruction.

C. Interconnect

We observed non-uniform access latencies in the interconnect when an access leaves a TPC. This was observed in the global memory access time and the L3 instruction and constant cache access time. The access time varied by nearly 100 cycles, dependent only on TPC placement. To factor out the effect of the interconnect, many of the tests report average performance over all 10 placements of the executing block on TPCs.

D. Global Memory, Caches

Memory characteristics were measured using pointer-chasing code. The kernel core measured the runtime of code which repeatedly read a value in memory to obtain the location of the next memory read. Due to the absence of data prefetching, our tests were done on arrays of varying sizes using linear accesses of varying strides. Most TLB measurements used the same technique with the stride set to the memory translation page size (512 KB).

The instruction cache tests did not have the ability to vary the access stride. Branch latency was too high to be useful for measuring stride or random accesses of the instruction memory. Measuring instruction caches required execution of a block of independent instructions and varying the instruction cache footprint.

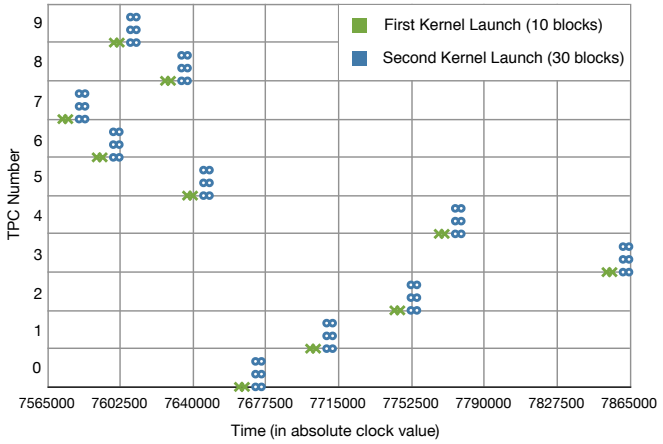

Fig. 4. Timing of two consecutive kernel launches for 10 and 30 blocks.

IV. Tests and Results

This section presents our detailed results and our tests and measurements. We begin by measuring our measuring tool, the clock() function. We then explore the SM's various arithmetic pipelines, branch divergence and barrier synchronization. We also explore the caching hierarchies both within and surrounding the SMs.

A. Clock

The clock() function returns the value of a counter that is incremented every clock cycle [1]. We use the clock() function for all of our timing measurements. Sampling the counter before and after a block of code for each thread provides a measure of the number of cycles taken by the device to completely execute the thread.

A clock() followed by a non-dependent operation takes 28 cycles. This is because the clock() function translates to a move from the %clock register followed by a left-shift of 1 (according to decuda) as follows:

```
mov $r1, %clock
shl $r1, $r1, 1
```

The pipeline latency is 24 cycles (See Section IV-B). Since these two instructions are dependent, it takes 24 cycles to execute the `mov`, plus 4 cycles to issue the `shl` instruction. The left-shift suggests that the counter is incremented at half the shader clock frequency.

%clock registers are per-TPC. Figure 4 shows the recorded timestamps for two consecutive kernel launches. The first kernel call runs 10 blocks on 10 different TPCs, while the second kernel call runs one block on each of the three SMs on all 10 TPCs. The two consecutive markers show the starting and ending times of each block execution on each SM. It can be seen from the second kernel call that the three blocks executing on the same TPC have the same timestamp value, while the timestamps vary across different TPCs.

	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

B. Arithmetic Pipelines

Each SM contains 3 different types of execution units (as shown in Figure 1):

- 8 SP units that execute single precision floating point and 32-bit integer ALU instructions.
- 2 SFU units that are responsible for executing transcendental functions and mathematical functions such as reverse square root, sine, cosine, etc.
- 1 DPU unit that handles computations on 64-bit floating point operands.

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

To measure the pipeline latency and throughput of each type of execution unit, we use a test consisting of a chain of dependent instructions that map to that unit, similar to the one shown in Listing 1 for multiplication. For the latency tests, we run a single block of only 1 thread. For the throughput tests, we run a block of 512 threads (maximum number of threads per block) to ensure full occupancy of the units. Our benchmark suite contains a kernel to test the latency and throughput of a number of the documented arithmetic, logic, and mathematical operations. Tables III and IV show which execution unit each of these operations map to, as well as the observed latency and throughput.

Table III shows the latency and throughput of various arithmetic instructions. Single-precision and double-precision multiplication and multiply-and-add (*mad*) map to a single device instruction, while 32-bit integer multiplication and *mad* translate to multiple native instructions. Integer multiplication translates to a chain of 4 dependent instructions, hence takes 4 times the pipeline latency (96 cycles). Similarly, integer *mad* translates to 5 dependent instructions and takes 120 cycles. The hardware only supports 24-bit integer multiplication via the __mul24() intrinsic. For 32-bit integer and double operands, division translates to a call to a subroutine that emulates this operation, resulting in high latency and low throughput. However, single-precision floating point division is translated to an inlined sequence of instructions, and has a lower latency and higher throughput.

Table III also shows that the throughput for single-precision floating point multiplication is $\sim$11.2 ops/clock, which means that multiplication can be issued to both the SP and SFU units. This suggests that each SFU unit is capable of doing $\sim$2 multiplications per cycle, twice the throughput of other (more complex) instructions that map to this unit. The table also shows that the throughput for single-precision floating point *mad* is 7.9 ops/clock, suggesting that *mad* operations cannot be executed by the SFUs.

Instruction	Type	Execution Unit	Latency	Throughput
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
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
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 INSTRUCTIONS

Instruction	Type	Execution Unit	Latency	Throughput
__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

__mul24() and __umul24() provide 24-bit signed and unsigned integer multiplication that map to a single native instruction, as shown in Table IV. The __log2f() and rsqrt() functions also map to single device instructions handled by

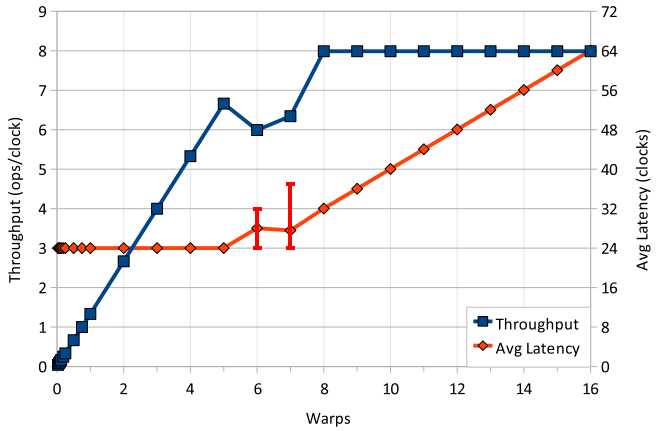

Fig. 5. SP Throughput and Latency

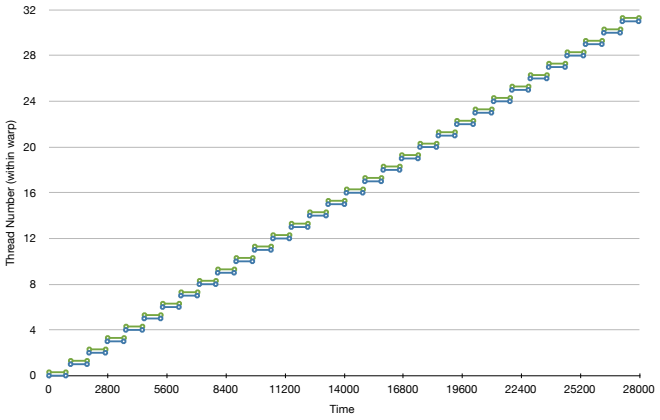

Fig. 6. Execution Timeline of Two 32-Way Divergent Warps.

the SFU. The __sinf(), __cosf(), and __exp2f() intrinsics each translates to a sequence of two dependent instructions operating on only one operand, and thus takes 48 cycles (2×24). __sqrt() is a sequence of two instructions: a rsqrt followed by reciprocal.

Figure 5 shows latency and aggregate throughput of dependent SP instructions (integer addition), as the number of warps on the SM increases. The observed latency is 24 cycles below 6 warps, when the pipeline is not full. Time-slicing increases the observed latency. Since all warps observe the same latency, the warp scheduler is fair. Throughput increases linearly while the pipeline is not full, then saturates at 8 operations per clock once the pipeline is full.

The scheduler does not manage to schedule optimally (does not fill the pipeline) when there are 6 or 7 warps in the SM, leading to increased latency and decreased total throughput than the SM should be capable of. The Programming Guide states that 6 warps (192 threads) should be sufficient to hide register read-after-write latencies, which would have been possible with better warp scheduling.

C. Control Flow

1) Branch Divergence: A warp executes a single common instruction at a time. Every instruction issue time, the SIMT unit selects a warp to execute and issues the instruction to the active threads of the warp. 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]. Figure 6 shows the execution timeline for two concurrent warps in a block whose threads diverge 32 ways, where each thread takes a different path based on its thread ID. The figure shows that within a single warp, each path is executed serially, while the execution of different warps may overlap. Threads that take the same path are executed concurrently with each other.

2) Reconvergence: When the execution of the different paths is complete, the threads converge back to the same execution path [1]. The compiler inserts one instruction before a potentially-diverging branch, which provides the hardware with the location of the reconvergence point. It also inserts the reconvergence point using a field in the instruction encoding.

When threads diverge, each path is executed serially until the reconvergence point, at which time the other path begins executing immediately after the divergent branch.

According to [5], a branch synchronization stack is used to manage independent threads that diverge/converge. We use the kernel shown in Listing 2 to confirm this statement.

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

Listing 2. Reconvergence Stack Test

We use the array c to specify the order in which the threads are executed. We observed that when a warp reaches a conditional branch, the taken path is always executed first. In this case, for each if statement the else path is the taken path, and is hence 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])) executed last.

Figure 7 shows the execution timeline of this kernel when the array c contains the increasing sequence {0, 1, ..., 31}. Thread 0 executes the first code block, thread 1 executes the second code block, and so on. In this case, the code block for thread 31 is the first to execute.

Figure 8 shows the results of the same test when the array c contains a decreasing sequence {31, 30, ..., 0}.

The above two tests show that the thread ID does not affect execution order. We observed execution ordering that was consistent with the taken path being executed first, and the fall-through path being pushed on a stack. Other tests showed that the number of active threads on a path also has no effect on which path is executed first.

3) Effects of Serialization: The programming guide states that for the purposes of correctness, the programmer can essentially ignore the SIMT behavior [1]. In this section, we

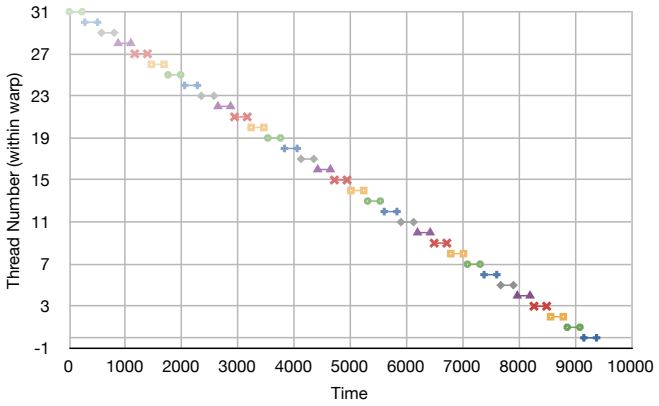

Fig. 7. Execution Timeline of Kernel Shown in Listing 2. Array c contains the increasing sequence {0, 1, ..., 31}

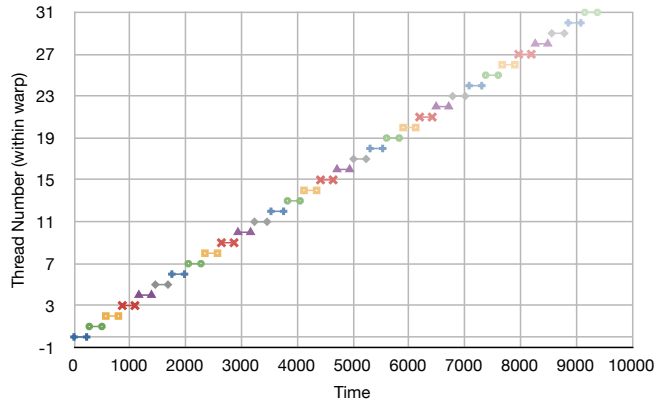

Fig. 8. Execution Timeline of Kernel Shown in Listing 2. Array c contains the decreasing sequence {31, 30, ..., 0}

show an example of code that would work if threads were independent, but breaks due to the SIMT behavior.

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

Listing 3. Example code that breaks due to SIMT behavior

In this code, the variable `sharedvar` starts out as 0. Intuitively, the first thread would break out of the while loop and increment `sharedvar`, which would cause each consecutive thread to do the same, fall out of the while loop and increment `sharedvar` for 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 until those threads also reach the reconvergence point. This causes deadlock as thread 0 cannot increment the shared variable.

D. Barrier Synchronization

Synchronization between warps of a single block is done using __syncthreads(), which acts as a barrier at which all warps wait until all warps reach the barrier before any warp is allowed to resume execution. The __syncthreads() operation is implemented as a single instruction with fairly low overhead. When a single warp executes a sequence of __syncthreads() operations, we observed a latency of 20 clock cycles for each __syncthreads().

1) __syncthreads() for Threads of a Single Warp: If only some threads within a warp execute __syncthreads(), the kernel does not hang, since __syncthreads() works at the granularity of a warp. The programming guide states that __syncthreads() acts as a barrier for all *threads* in the same block. The test in Listing 4 shows that __syncthreads() waits for all *warps* (all 1 warps in this test) in a block before continuing. This kernel is executed for a single warp, in which the first half of the warp writes an array of shared counters, and the second half of the warp reads the elements of the same array.

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

Listing 4. Example code that shows __syncthreads() synchronizes at warp granularity

In this example, the second half of the warp does not read the updated values in the array count, showing that __syncthreads() does not synchronize diverged threads within one warp as the programming guide's description would suggest.

2) __syncthreads() Across Multiple Warps: When a warp calls __syncthreads(), it waits at the barrier until all other warps in the block either call __syncthreads() or terminate, then resumes execution. The kernel does not hang if only some warps within the block call __syncthreads(), as long as those that do not call __syncthreads() eventually terminate. If __syncthreads() waits for warps that for any reason do not terminate, then __syncthreads() will wait indefinitely, as there is no time-out mechanism. Listing 5 shows one example where one warp spins waiting for data generated after the __syncthreads() in the other warp.

The programming guide states that __syncthreads() is allowed in conditional code, but only if the conditional evaluates identically across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects [1]. Listing 6 shows the details of what happens when the above recommendation is violated.

In this example, the second and third __syncthreads() synchronize, while the first and fourth __syncthreads() synchro-

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

Listing 5. Example code that deadlocks due to __syncthreads()

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

Listing 6. Example code that produces unintended results due to __syncthreads()

nize together. (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; Diverged warps are no exception, as each serialized path executes __syncthreads() separately and waits for all other warps in the block (i.e. warp1) to also execute __syncthreads() (or terminate).

E. Register File

The register file contains 16,384 32-bit registers (64 KB), as the CUDA Programming Guide states. The number of registers used by a thread is rounded up to a multiple of 4. We did not notice any quantization of the allocation of the registers in the register file beyond this constraint.

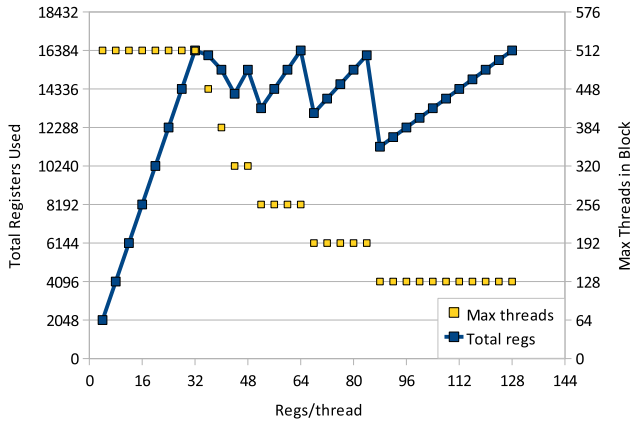

Fig. 9. Total registers used by a block is limited to 16,384 (64 KB). Maximum threads in a block is quantized to 64 threads, indicating 64 banks.

Attempting to launch a block of threads that use a total of more than 64 KB of registers results in a failed launch. Likewise, attempting to use more than 128 registers per thread fails. The *Total regs* series in Figure 9 illustrates this limit. It can be seen that the number of registers used in a block is at most 16,384. The region below 32 registers per thread is limited by 512 threads per block.

It can also be seen in Figure 9 that the number of threads run in a block is quantized to 64 threads. This puts an additional limit on the number of registers that can be used, and is most visible when threads use 88 registers each: Only 128 threads can run in a block, and only 11,264 (88×128) registers can be used, as 192 threads would use more than 16,384 registers.

The quantizing of threads per block to 64 threads suggests the use of 64 memory banks in the register file. Note that this is different from quantizing the total register use.

The 64 register file banks would likely be distributed over the 8 SPs (8 banks per SP). Having at least four banks per SP *and* distributing the execution of each warp over four cycles allows the use of single-ported memories to provide sufficient bandwidth to execute three-read, one-write operand instructions (e.g. multiply-add) with no bank conflicts. Having eight banks per SP would presumably be caused by the need to provide extra bandwidth for the "dual-issue" feature using the SFUs (see Section IV-B) and for memory operations in parallel with arithmetic.

A similar limitation was alluded to in Section 5.1.2.6 of the CUDA Programming Guide 2.0 [1]. The Manual states that due to bank conflicts, "best results" are achieved if the number of threads per block is a multiple of 64. What we observed is 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

Threads belonging to the same block can cooperate by using data in shared memory. The documentation [1] states that shared memory is as fast to access as an L1 cache, with a latency of approximately two cycles. We ran a micro-benchmark that performed a sequence of dependent reads and measured the read latency to be 38 cycles.

Unfortunately, we could not identify the reason for this 12 cycle difference. (The pipeline latency is 24 cycles.) We note that Volkov and Demmel [6] report a similar latency of 36 cycles on the 8800GTX, the predecessor to the GT200. We used decuda [4] to see how reads are translated natively. Every read is translated into a *movsh* command: movsh.b32 $ofs1, s[$ofs1+0x0030], 0x00000002. This command uses a special offset register, performs an addition and a left shift by two.

The amount of shared memory allowed per block is 16KB. The kernel's function parameters occupy shared memory, and typically 16 bytes are reserved for system use. For our test that uses 5 parameters, the compiler reserved an additional 12 unused shared memory bytes, and we can allocate at most 16336 bytes of shared memory. Larger sizes give the following

kernel error when launched: "*too many resources requested for launch*".

G. Global Memory

The global memory is accessible by all running threads, even if they belong to different blocks. Our incentive in benchmarking the global memory was twofold: first, we wanted to verify the performance characteristics provided in the CUDA Programming Guide and second we wanted to understand how memory translation is performed in the GT200. Our findings for the latter are presented in Section IV-H.

An access to global memory has a documented latency of 400-600 cycles [1]. We implemented a benchmark which executed a sequence of pointer-chasing dependent reads to global memory. We found the read latency to be 440 to 441 cycles.

Global memory has no documented caches. To this respect, we ran a micro-benchmark that swept through a specific array, whose size we gradually increased. We confirm that no caching effects were witnessed, except when the size of the array was smaller or equal to 256 Bytes. In these few cases we witnessed a small decrease in memory latency in the range of two to three cycles. We consider these to be DRAM effects, possibly related to DRAM's column buffer size.

H. Memory Translation

To understand how memory translation is performed, we ran a sequence of dependent reads which were situated *stride* bytes apart in an array. We varied both the size of the array, as well as the stride. If strides are big enough to fall into different pages and we access many pages, we expect to see the presence, if any, of a translation lookaside buffer (TLB). In all our experiments our code fits in the instruction cache and the first iteration is ignored to eliminate skewing the results from cold instruction cache and TLB misses.

Figure 10 shows the presence of two TLBs. The horizontal axis depicts the stride in bytes, while the various series correspond to different array sizes. The read latency presented in the previous section includes the access to the first level TLB. The latency penalty of accessing the second level TLB is ∼50 cycles, while a miss there results in a total latency of approximately 700 cycles.

If the two TLB levels were fully associative, we could infer from this figure that the first level has 16 entries and the last level has 31 entries with the 32nd entry potentially dedicated to the code segment. We find that only the first level TLB is fully associative.

To validate our initial conclusions, we created another micro-benchmark where only a fixed number of array elements are accessed in a round-robin fashion. Figure 11 shows how the access latency (y axis) changes in respect to two parameters, the number of elements accessed (series) and the stride difference of these elements (x axis). When the strides are small enough, accesses will fall into the same page, while once the strides get larger than the page size, the pages will spill to lower TLB levels and to memory.

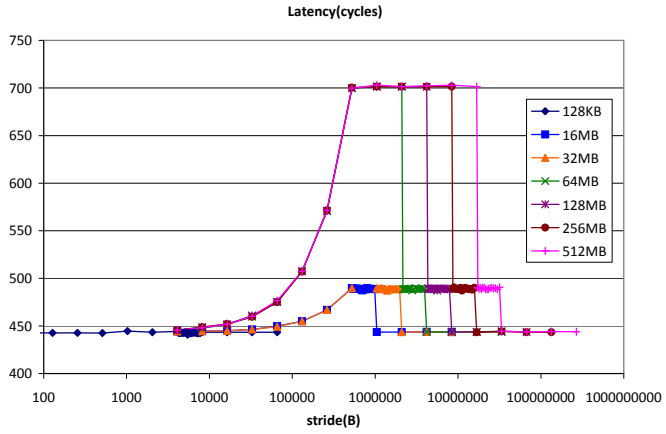

Fig. 10. Global Memory - Presence of two TLBs

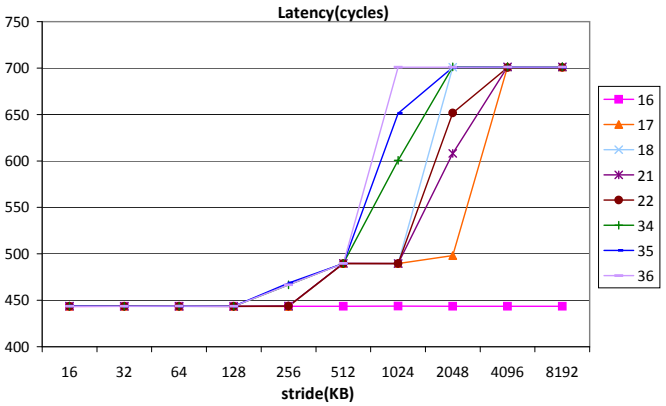

Fig. 11. Page Size and TLB entries

If we focus on the series where 17 elements are accessed we can see that the first stride where there are misses is at 512 KB. This is the page size. In addition we can see that for a 512 KB stride, any series that accesses more than 16 elements will have 100% miss rate on the first level TLB. For a 256 KB stride, which is half the page size, series that access up to 32 elements always hit in the first level. Even though the first level has 16 entries, every two accesses fall within the same page due to the stride pattern. If more than 32 elements are accessed then the miss rate is 50% for the first TLB level, resulting in a latency of approximately 467 cycles.

The second level TLB has 64 entries and is eight-way set-associative. The only outlier to the idea of a 64 entry eight-way set-associative L2 TLB is the 17 series. Specifically, the point at 2 MB should be at ∼600 cycles (50% miss rate) instead of 500. This behavior can explained by the presence of some extra storage, as explained later in this section.

Figure 12 presents the associativity of the two TLB levels. The number of elements in the line that transitions from one TLB level to another is the number of sets in the TLB. The first level TLB is fully associative and thus the transition from 441 cycles to 490 happens in one step. The transition from 490 cycles to 700 includes eight transition points. The points

appear to be nicely spaced and we would expect the presence of an eighth point at roughly 515 cycles, rather than the current ∼495 cycles. Its lack is explained in the following test.

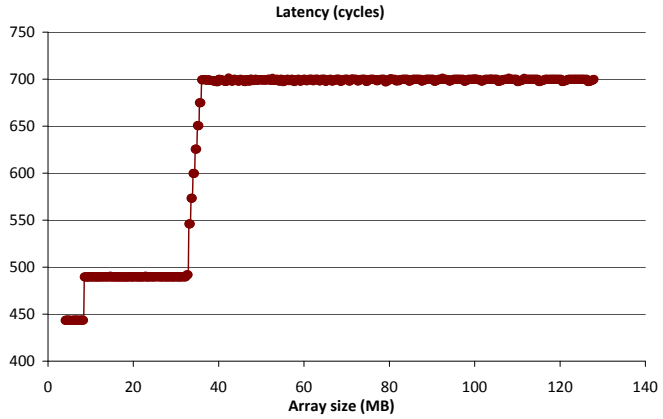

Fig. 12. TLB associativity

Figure 13 presents a possible explanation for the two inconsistencies we noted in figures 11 and 12. It shows the presence of another structure backing up the second level TLB. The latency of this structure appears to be approximately 17 cycles.

Our initial intention was to confirm that the associativity of the second TLB level is eight. This is possible to see, if all missing accesses to the first TLB level (TLB1) map to a single set in the second level TLB (TLB2). However, since TLB1 has 16 entries and is fully associative, accesses that will miss the TLB1 will already overflow a TLB2 set since its associativity is around eight. To overcome this issue we changed the stride access pattern of our micro-benchmark as follows: Our micro-benchmark accesses x number of elements, in a round-robin fashion. The first 16 elements are spaced 512KB apart, i.e., the page size, so that they fill the first two ways of TLB2. The remaining x - *16* elements are spaced 4MB apart, so that they map to a single set. 4MB is the page size (512 KB) multiplied by the number of sets (8).

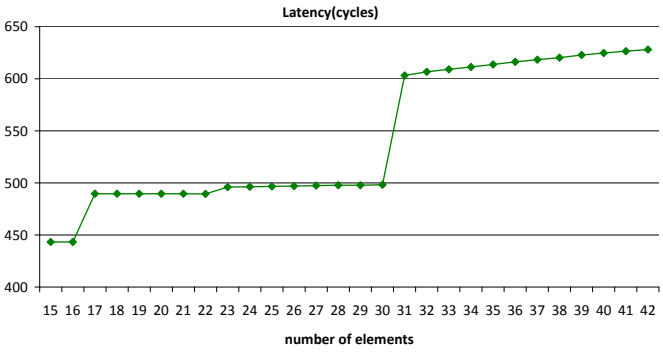

Fig. 13. Presence of an additional structure post the second TLB level

In Figure 13 the horizontal axis shows the number of elements accessed (i.e., the value of x). When up to 16 elements are accessed, all of them hit in TLB1 resulting in an average latency of 441 cycles. When the number of elements is between 16 to 22, then they all hit in TLB2. Specifically in the 22 case, the first 16 occupy two ways and the remaining six fill in the available 6 ways of a given set. The latency is 490 cycles.

If more than 22 elements are accessed, we expect a latency increase since some of these accesses will be satisfied in memory. However, there will always be 14 hits in TLB2 since seven out of the eight sets will have no conflicts and thus 14 out of the 16 elements allocated in the first two ways will always hit.

In summary, with our benchmark, we expected to see a linear increase in the access latency, that would be half the latency to memory when 28 elements were accessed. In our results, this linear increase start at 31 elements. From 23 to 30 elements, we do see an increase but it is only by six cycles. This extra step is indicative of another hardware structure that acts as a victim cache to the second level TLB. Some of our results indicate that this structure could have eight entries and be fully associative or it could have a single entry. Further research is required to identify its organization.

The presence of this structure explains the two inconsistencies mentioned earlier. In Figure 11 in the 17 series the point at 2 MB has a latency of 500 because it hits in this extra structure. Similarly, in Figure 12 the missing eighth point exists but has a much lower latency than expected because it hits in this backup TLB.

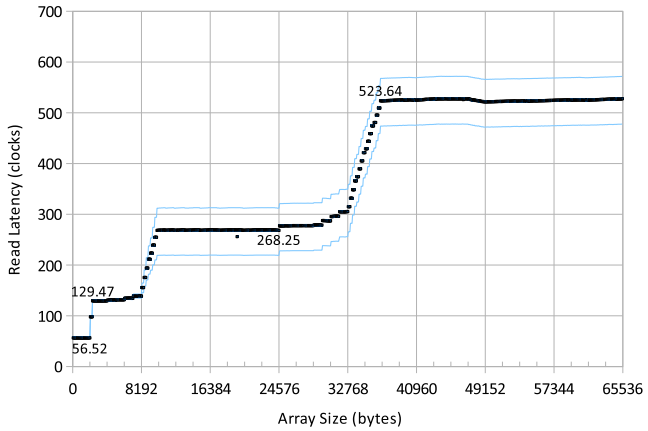

Fig. 14. Constant Memory. Latency includes one address computation instruction. Three levels of caching are seen, with the L3 having non-uniform access times.

I. 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) [3]. The user-accessible segment is limited to 64 KB. The Programming Guide states that the constant memory space is cached.

We detected three levels of caching of the constant memory space. The access latency using a stride of 256 bytes, including one address computation, is plotted in Figure 14. Addressing constant memory space appears to require the use of a special

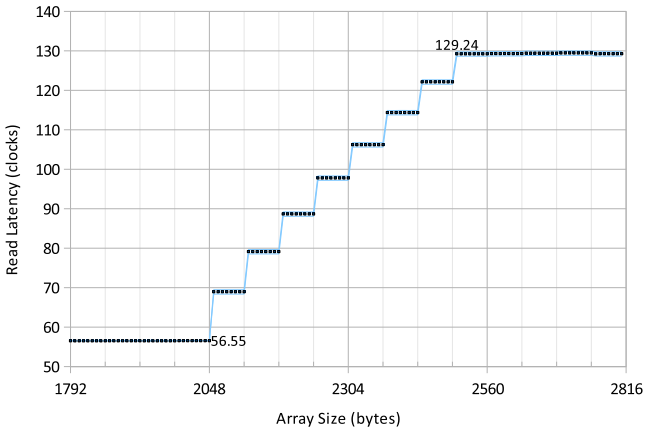

Fig. 15. Constant Memory, highlighting the 2 KB L1 cache. Latency includes one address computation instruction.

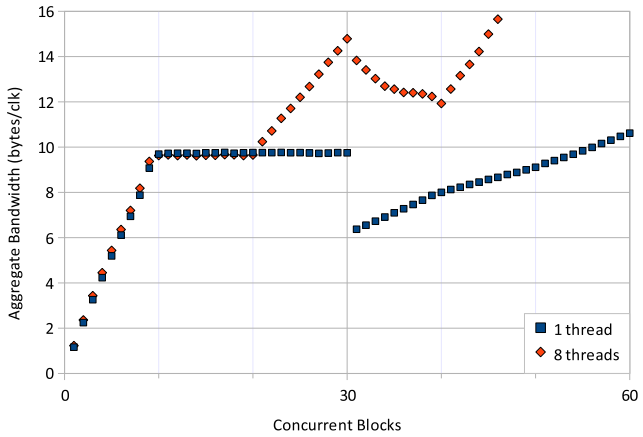

Fig. 16. Constant Memory L3 bandwidth: 9.75 bytes/clock

offset register ($ofs1) with an immediate offset operand (0 to 32,767). Modifying $ofs1 appears to be impossible by an instruction which reads constant memory, thus address computations add at least one extra instruction of overhead. The latency measurements are averaged over all placements of the test code onto TPCs. The maximum and minimum latencies over the different placements are also plotted.

The plot clearly shows three levels of caching of sizes 2 KB, 8 KB, and 32 KB. The access latency of each cache level is roughly 56, 129, 268. The access latency when all caches miss is 524 cycles, which is higher than the observed latency for uncached global memory (See Section IV-G), partly due to the need for a separate address computation instruction. The measured latency includes the latency of two arithmetic instructions (one address computation, one to load the value), so the raw memory access time would be roughly 48 cycles lower than the numbers reported here.

1) L1: A 2 KB L1 instruction cache is located in each SM (See Section IV-I4). The L1 cache has a 64-byte cache line size, and is 4-way set associative with 8 sets (32 lines total). It can be seen from Figure 15 that L1 constant cache misses begin at 2 KB, and increase over 8 steps from 56.5 clocks to 129.3 clocks, with each step being 64 bytes in size. The step size indicates the cache line size (64 bytes) and a total cache size of 32 lines (2,048 bytes = 32 lines $\times$ 64 bytes). Eight steps indicate a set-associative cache containing 8 sets, implying a 4-way set associativity for a cache with 32 lines.

2) L2: A 8 KB L2 constant cache is located in each TPC and is shared with instruction memory (See Section IV-I4). The L2 cache has a 256-byte cache line size and is 4-way set associative with 8 sets (32 lines total). The region near 8,192 bytes in Figure 14 shows 8 distinct latency steps (8 sets) between L2 hit and L3 hit, with each step being 256 bytes wide (256-byte cache line size). Eight cache sets in a 32-line cache indicates 4-way set associativity.

3) L3: We detected a single 32 KB L3 constant cache, shared between all TPCs. The L3 cache has a 256-byte cache line size and is 8-way set associative with 16 sets (128 lines total). The L3 constant cache is connected to the TPCs using a non-uniform interconnect.

We observe cache parameters in the region near 32,768 bytes in Figure 14. There are 16 distinct latency steps (16 sets) between L3 hit and L3 miss, with each step being 256 bytes wide (256-byte line size), implying 8-way set associativity for a cache with 64-lines. We believe this is the last level of caching, as the L3 miss latency is already greater than the latency of uncached DRAM (See section IV-G).

As can be seen in Figure 14, the minimum and maximum access latencies for the L3 cache (8-32 KB region) differ significantly depending on which TPC executes the test code. This suggests that there is a single L3 cache located at a fixed position on a non-uniform interconnect, where some TPCs are closer to the L3 cache than others. The L1 and L2 cache accesses do not see this spread of access times when TPC placement is varied, indicating this interconnect connects TPCs together, and is not used when memory traffic stays within the TPC. We leave investigation of the interconnect topology for future work.

We also measured the L3 cache bandwidth. Figure 16 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 of each other.

The bandwidth of the L3 constant cache appears to be $\sim$9.75 bytes/clock, seen in the aggregate bandwidth when running between 10 and 20 blocks. Each TPC appears capable of generating $\sim$1.2 bytes/clock of traffic.

Two variants of the tests were run: one using one thread per block, and one using eight threads per block to increase constant cache fetch demand. Both tests show similar behavior below 20 blocks. This suggests that not only is there a limit to L3 bandwidth (9.75 bytes/clock), but that each TPC is only capable of requesting 1.2 bytes/clock of fetches regardless of demand within the TPC as long as the TPC has only one block executing.

The measurements are not valid above 20 blocks in the eight-thread case, as there are not enough unique data sets and the per-TPC L2 cache hides some of the requests from the L3, causing apparent aggregate bandwidth to increase. Above 30

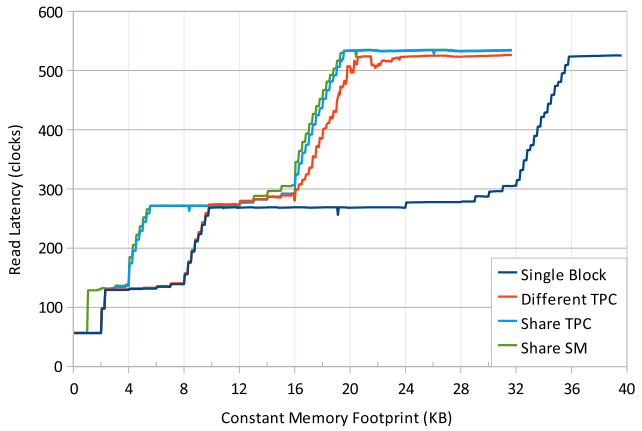

Fig. 17. Constant Memory Sharing. Per-SM L1 cache, per-TPC L2, global L3.

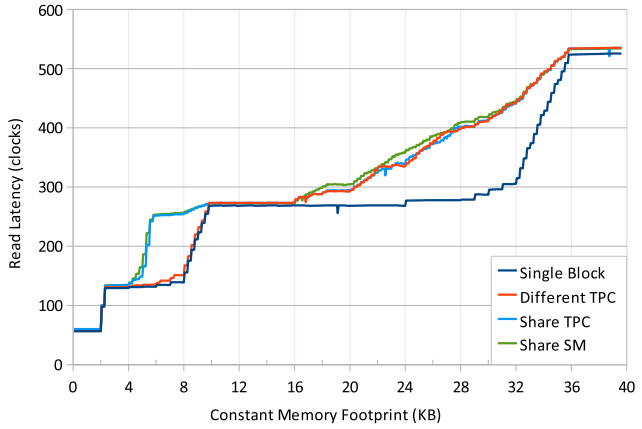

Fig. 18. Constant Memory Instruction Cache Sharing. L2 and L3 caches are shared with instructions.

blocks, the one-thread case has some SMs running more than one block.

4) Cache Sharing: The L1 constant cache is private to each SM, the L2 is shared within a TPC, and the L3 is global. This was tested by measuring the latency profile using two concurrent blocks with varying placement (same SM, same TPC, different TPC). If caches are shared, then the two blocks' memory footprints will both occupy space in the shared cache, and the observed cache size will be halved. Figure 17 shows the results of this test.

The baseline with only one block is plotted for comparison: The baseline shows the three caches with size 2 KB, 8 KB, and 32 KB. In all cases with two blocks, the memory footprint interferes at the L3 cache, causing the observed cache size to be halved to 16 KB. Thus the L3 cache is shared regardless of placement of the two blocks. When the two blocks are placed within the same TPC (but not when on different TPCs) the observed L2 cache size is halved to 4 KB, indicating L2 caches are per-TPC. Following similar logic, only when the two blocks are placed on the same SM is the observed L1 cache size halved to 2 KB, so L1 caches are private to each SM.

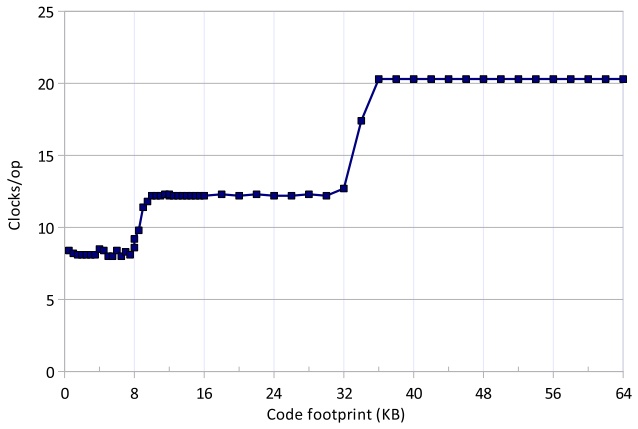

Fig. 19. Instruction cache latency. This test fails to detect the 4 KB L1 cache. The L2 cache is 8 KB and the L3 cache is 32 KB.

5) Cache Sharing with Instruction Memory: It has been suggested that the constant cache hierarchy was the same as the instruction caches [7], [8]. It seems likely that the L2 and L3 constant caches may also be the L2 and L3 instruction caches, as the measured cache parameters of the L2 and L3 constant caches match the instruction caches (See Section IV-J). The L1 parameters differ from the L1 instruction cache, so we do not expect the L1 to be shared. 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-I4, we ran two blocks of code with varying placements. We measured the effect on constant memory access times when another block uses a large code footprint by looping through 10 KB of code. The result is plotted in Figure 18. The access times of the L3 constant cache is increased when instructions compete for it, so the L3 is shared. Likewise, the observed L2 constant cache size is decreased when instructions compete for the cache in the same TPC. The L1 access times are not affected by instruction fetch demand, even when the blocks run on the same SM, so the L1 caches are for constant memory only.

6) Multiple Memory Accesses in a Warp: We believe each warp is capable of one constant memory access at a time. We attempted to measure the L2 bandwidth, but have so far been unable to saturate the L2 cache bandwidth. We believe this is because each warp supports only one outstanding constant memory access at a time, even if memory accesses are independent: The architecture is unable to extract memory-level parallelism from within a single warp. Although accesses within a warp are serialized, it appears accesses from different warps can occur in parallel. We will investigate this conjecture in future work.

J. Instruction Supply

Each instruction executed by an SM needs to be fetched from memory. This section discusses our observations on the cache hierarchy that services the instruction supply.

We detected three levels of instruction caching, of size 4 KB, 8 KB, and 32 KB. Because of the constraints of not

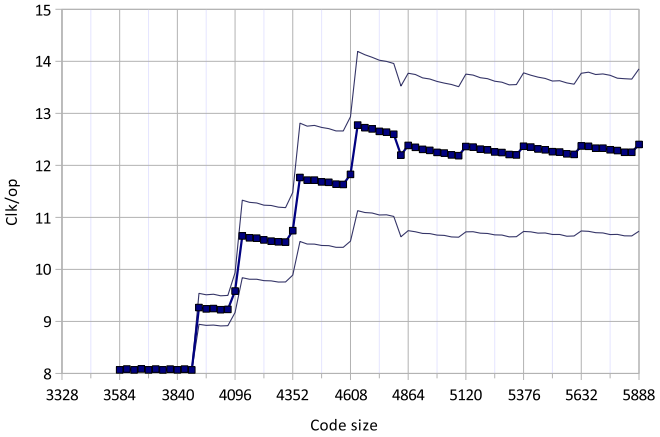

Fig. 20. Instruction cache L1 latency. The L1 is 4 KB, 4-way, 256-byte lines. Latency is made visible by adding competition for the L2 cache in other blocks.

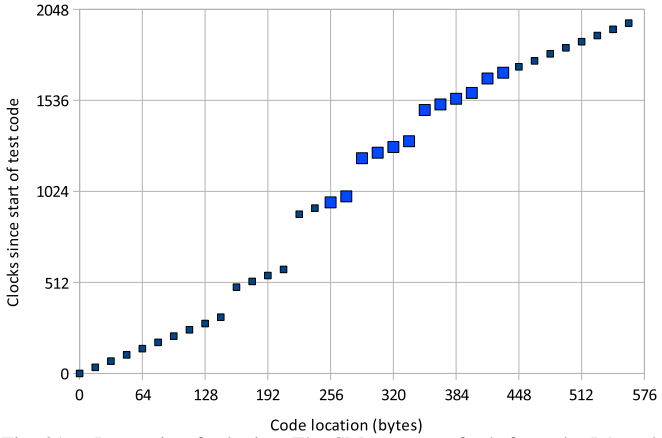

Fig. 21. Instruction fetch size. The SM seems to fetch from the L1 cache in blocks of 64 bytes.

being able to make random accesses in the instruction stream (branches are too slow), the L1 instruction cache was difficult to detect. Figure 19 shows the latency profile for looping through a block of code of varying size. The 8 KB L2 and 32 KB L3 caches are clearly visible, but the 4 KB L1 is not visible, probably due to a small amount of instruction prefetching that hides the L2 access latency. Note that the absolute magnitude of *clocks/op* does not directly reflect the access latency of the instruction cache or memory. In particular, 8 clocks per operation reflects the throughput limit of one warp in the arithmetic pipeline when all instructions are independent.

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

The L1 cache parameters were measured by running (on the same TPC) 5 concurrent blocks of code: Four to flood the L2 instruction cache, and one to measure. The competition for the L2 cache allows the L1 misses to create visible performance loss, instead of staying hidden as in Figure 19 near 4 KB. The resulting measurement is shown in Figure 20. The 256-byte line size is visible, as well as the presence of 4 cache sets. We know the L1 instruction cache is located in the SM because flooding the L2 from other SMs on the same TPC does not decrease the observed L1 instruction cache size of 4 KB.

2) L2: The L2 instruction cache resides in the TPC, and is 8 KB with 256-byte cache lines and 4-way set associativity.

We have shown in Section IV-I5 that the L2 instruction cache is also used for constant memory. Due to space constraints we omit the cache parameter test results which verified the cache parameters on the instruction fetch path are the same as for constant memory accesses.

3) L3: The L3 instruction cache is global, and is 32 KB with 256-byte cache lines and 8-way set associativity.

We have shown in Section IV-I5 that the L3 instruction cache is also used for constant memory. We again omit independent results here due to space constraints.

We measured the fetch bandwidth of the L3 instruction cache, and found it to be 8.45 bytes per clock, somewhat lower than the 9.75 bytes per clock measured for the L3 constant cache in Section IV-I3. Due to the limitations of measuring instruction streams, we believe the constant cache measurement is closer to the peak bandwidth of that cache.

4) Instruction Fetch: The SM appears to fetch instructions from the L1 instruction cache in blocks of 64 bytes. Figure 21 shows the execution time of 36 consecutive instructions while other warps running on the same SM continually evict the region indicated by the large points in the plot. We see that whole cache lines are evicted (spanning the region 160-416 bytes) and that the effect of a cache miss is only observed between but not within blocks of 64 bytes (4 instructions).

5) Instruction Cache Sharing: The L2 and L3 instruction caches are per-TPC and globally-shared, respectively. The L2 and L3 are the same caches as the L2 and L3 constant caches (See Section IV-I5). We omit presenting instruction cache sharing tests due to space constraints. See Section IV-I4 on the constant caches.

V. Related Work

Volkov and Demmel [6] benchmarked the 8800GTX GPU to tune linear algebra. They measured the characteristics of some form of data memory, but did not state to which memory spaces their results applied. Although we are using the successor to the 8800GTX, we expect many of the measurements to be similar.

They observed cache parameters quite different from our measurements of the instruction and constant caches. They also observed a 16-entry TLB using 512 KB pages, but did not observe a second-level TLB.

GPUBench [9] is an older set of GPU micro-benchmarks that measure some of the instruction and memory characteristics of programs running on GPUs. The measurements tend to be fairly high-level due to its use of the OpenGL ARB shading language.

VI. Future Work and Conclusions

The goal of this project was to construct a suite of micro-benchmarks targeting specific parts of the architecture, so that

we can understand in detail the hardware organization of the NVIDIA GT200 GPU.

Table V summarizes our findings.

Arithmetic Pipeline

	Latency (cycles)	Throughput (ops/cycle)
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	16K 32-bit registers, 64 banks (8 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
Global	∼441-cycle latency
TLBs	512 KB page size L1: 16 entries, fully associative L2: 64 entries, 8-way set associative Backup TLB: unknown organization
Constant	L1: 2 KB, 64-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 instruction memory
Shared	16KB, 38 cycles latency

TABLE V
GT200 ARCHITECTURE SUMMARY

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 the caching and TLB hierarchies. In addition, in some cases our findings deviated from the documented characteristics (e.g., for shared memory). Further exploration of these cases is left for future work.

In terms of future work, there are various open questions which we would like to address.

- Texture memory characteristics
- Memory throughput
- Memory access granularity for global and shared memory
- Organization of the backup TLB
- Explanation for the increased shared memory latency
- Interconnect network. Our instruction cache and global memory benchmarks revealed latency heterogeneity, based on which TPC the block was assigned.
- Atomic operations

Many of the issues we faced about how to write a microbenchmark that can identify various levels of caching, how to avoid compiler optimizations and similar challenges exist across all architectures. We believe there is room for algorithms that could be reused to identify specific characteristics of any underlying architecture. The ultimate goal is to know the hardware better, so that we can harvest its full potential.

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

[3] Nvidia. The CUDA Compiler Driver NVCC. http://www.nvidia.com/object/io_1213955090354.html.

[4] van der Laan, Wladimir J. http://www.cs.rug.nl/˜wladimir/decuda/.

[5] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[6] Vasily Volkov and James W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

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

[8] David Kirk and Wen-mei 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.

[9] Ian Buck, Kayvon Fatahalian, and Mike Houston. GPUBench. http://graphics.stanford.edu/projects/gpubench/.