

Optimizing for CUDA I

Nikos Hardavellas

Some slides/material from:
UToronto course by Andreas Moshovos
UIUC course by Wen-Mei Hwu and David Kirk
UCSB course by Andrea Di Blas
Universitat Jena by Waqar Saleem
NVIDIA by Simon Green, Mark Haris, Paulius Micikevicius and many others
Real World Technologies by David Kanter
AnandTech by Ryan Smith

1

GTX680 hardware

- Stream Multiprocessor Array (i.e., the GPU device)
 - 8 Stream Multiprocessors (SMs)
 - Organized into 4 Graphics Processing Clusters (GPCs)
 - Texture Cache: 48KB per GPC
- Stream Multiprocessors
 - 192 Stream Processors (6 functional unit clusters x 32 cores)
 - 32-bit integer ALU (5 out of 6 clusters can do ints in Kepler GK104)
 - 32-bit FP (NaN, denormals → ± 0 , round to nearest even)
 - 2 clusters x 16 Ld/St units
 - 2 x 16 Special Function Units (log2, sin, cos, rsqrt, etc.; 10s of cycles)
 - 1 x 8 Double-Precision 64-bit FP Units (one extra functional unit cluster)
 - 2 x 16 Interpolation Units (per-pixel interpolation)
 - 2 x 8 Texture Units
 - 16/32/48KB Shared Memory / 32 Banks / 32-bit interleaved
 - 64K Registers
- Minimum execution unit
 - 32 Threads (1 warp); even if you request fewer threads, HW rounds up

2

GeForce GTX 680 – Functional Unit Clusters



- **SM (a.k.a. SMX, SMP)**
 - Streaming Multiprocessor
 - Multi-threaded processor
 - 192 CUDA cores
 - 1 to 2048 threads active
 - Shared instruction fetch per 32 threads
 - Fundamental processing unit for CUDA thread block
- **SP (a.k.a. CUDA core)**
 - Streaming Processor
 - Scalar ALU for a single CUDA thread
- **SFU**
 - Special function unit
- **LDST**
 - Memory access unit

3

GTX680 scheduling per SM

- 4 Warp schedulers
 - Warp scheduler: schedule 2 instructions per cycle (ILP – coming up!)
- Each instruction targets one *functional unit cluster* (out of 15)
 - 6 clusters of 32 CUDA cores each (int, FP arithmetic)
 - 2 clusters of 16 Ld/St units each
 - 2 clusters of 16 Special Function units each
 - 2 clusters of 16 Interpolation units each
 - 2 clusters of 8 Texture units each
 - 1 cluster of 8 FP64 cores (does not appear in NVIDIA's diagrams)
 - FP64 cores run at full speed → 1/24 the FP32 performance
- One warp executes all 32 threads in parallel
 - Static scheduling of math instructions within warp (known math latency)
 - Scoreboarding for long-latency operations (loads, texture)
 - Scoreboarding for scheduling across warps
- Thread
 - 63 registers

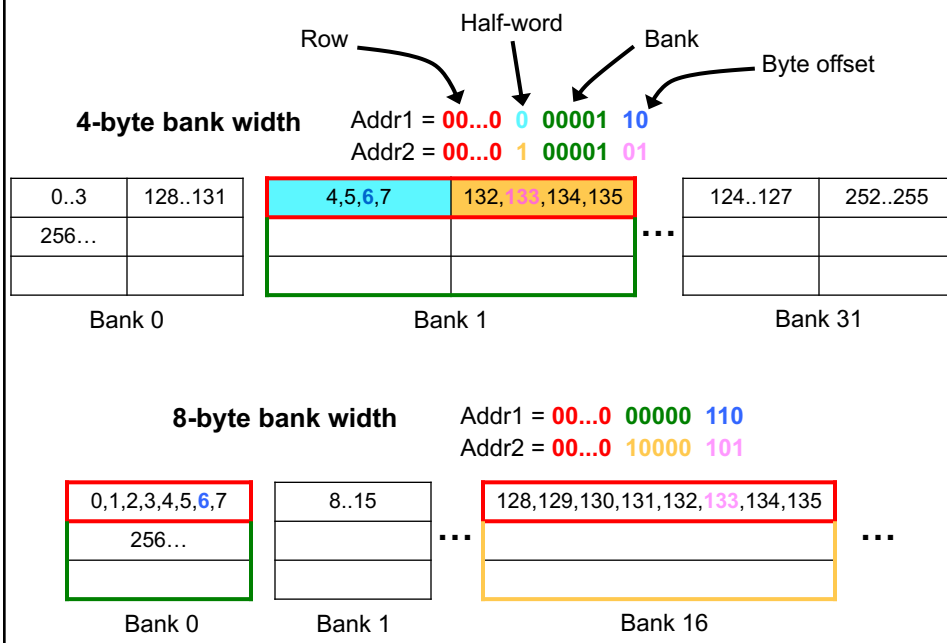
4

GTX680 memory

- Constant memory
 - 64KB in DRAM / cached
- Local memory
 - 48KB (?) in DRAM / cached in L1
- Texture cache per GPC
 - 48KB, treated as fast read-only cache
- Shared memory/L1
 - Three configurations: 48/16KB, 32/32KB, 16/48KB
 - Set by `cudaDeviceSetCacheConfig()`
 - Up to 48KB ShMem, 32 banks, 8-bytes wide (4-byte default), multi-cast
 - 4- or 8-byte access set by `cudaDeviceSetSharedMemConfig()`
 - Address =**B**B**B**B**B**xxx, “B” designates bank, “x” word offset (**8-byte**)
 - Address =**H**B**B**B**B**Bxx, “H” selects half word (**4-byte addressing**)
 - Banks address-interleaved at 32-bit granularity
 - 1.5-2 TB/s, 20-30x lower latency than global memory (13-40 cycles)
- L2 cache
 - 512KB, all accesses to global memory go through L2 (CPU, GPU)

5

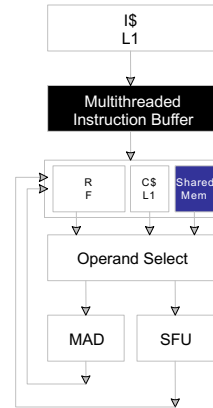
Shared memory addressing example



6

SM memory architecture

- Threads in a Thread Block can share data & results
 - In Global Memory and **Shared Memory**
 - Synchronize at barrier instruction
- Shared Memory allocated per Thread Block
 - Visible to all threads within a thread block
 - Dynamically allocated (upon block scheduling on SM)
 - Limited resource



7

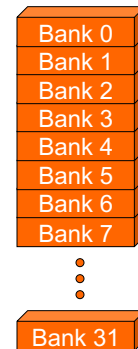
How to get high-performance #1

- Programmer managed shared memory**
 - e.g., `__shared__ int SharedVar;`
 - Accessed in parallel by threads in block
 - Keeps data close to processor (fast)
 - Minimize trips to global memory (slow)
 - a.k.a. "scratchpad memory"
- Programmer needs to:
 - Decide what to bring and when
 - Bring data in from global memory, reuse
 - Decide which thread accesses what and when
 - Coordination paramount (remember data races?)
- Key Performance Enhancement
 - Move data in shared memory**
 - Operate in there**

8

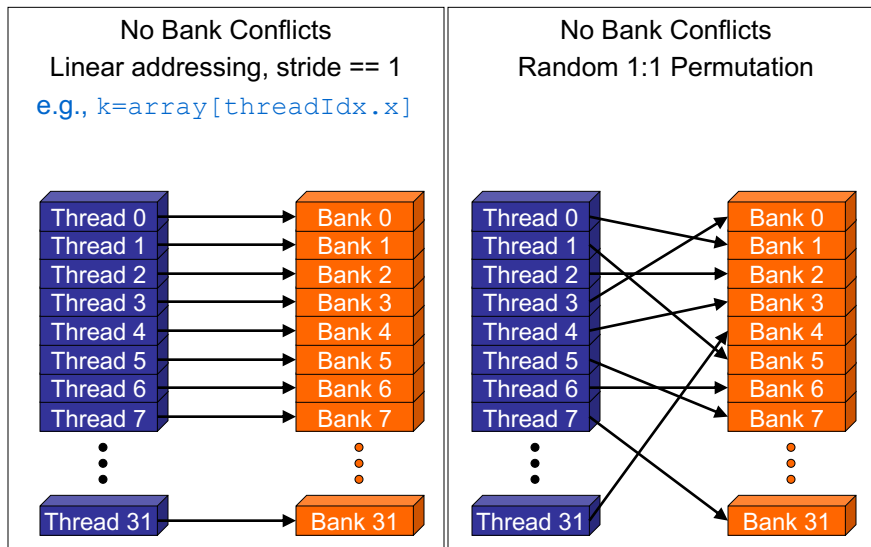
Parallel memory architecture considerations

- In a parallel machine, many threads access memory
 - Therefore, memory is divided into banks
 - Essential to achieve high bandwidth
- Each bank can service one address per cycle
 - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
 - Conflicting accesses are serialized



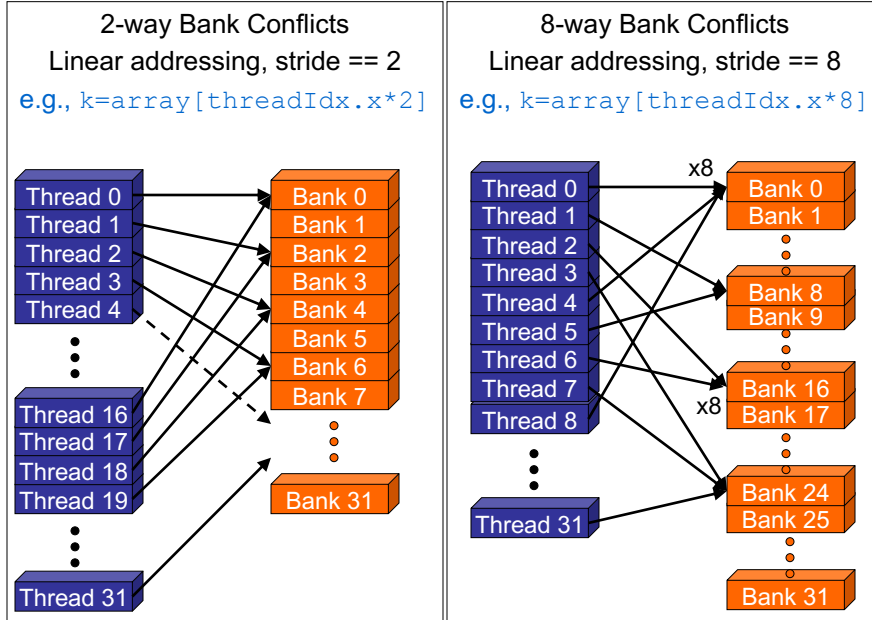
9

Bank addressing examples



10

Bank addressing examples



11

Shared memory bank conflicts

- Shared memory is as fast (latency) as registers if there are no bank conflicts
- The fast case:
 - If all threads of a warp access **different banks**, there is no bank conflict
 - If all threads of a warp access the **same address**, there is no bank conflict (broadcast)
- The slow case:
 - Bank Conflict: multiple threads in the same warp access the same bank for different addresses
 - Must serialize the accesses
 - Cost = max # of simultaneous accesses to a single bank

12

GTX680 global memory (DRAM) considerations

- Multiple banks per chip
 - eight 2-Gbit chips → 2GB global memory
- Global memory loads cached in L2
- Timing constraints
 - 400-800 cycles
- GDDR5
 - 6.008 Ghz → 6.008 Gbit/sec/pin
 - 256-bit interface → 32 bytes/mem_clock
 - 32 bytes/mem_clock * 6.008e+9 mem_clocks/sec → 192.256 GB/sec
 - Load granularity = 128 bytes (32 threads x 4-bytes each per GPU clock)

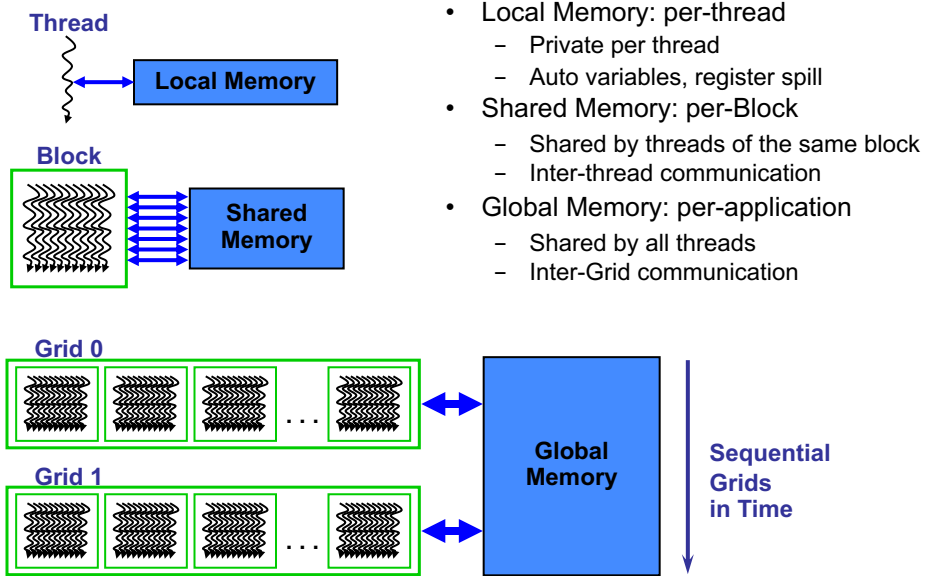
13

How to get high performance #2

- Coalesce global memory accesses
 - 32 threads access memory together
 - Can coalesce into **single** reference, if addresses within a 128-byte block
 - Instead of issuing 32 memory accesses of 4 bytes each
 - Issue 1 memory access that is 128 bytes wide (load granularity)
 - GDDR5 provides 32 bytes/mem_clock → 128 bytes in 4 Mem cycles
 - But GDDR5 @ 6.008GHz, while cores @ 1.006GHz
 - one core cycle is 5+ memory cycles
 - get 128 bytes in 1 core cycle
 - Coalesce requests to get one access as wide as possible
 - e.g., `a[threadID]` works well
 - Ideal: 1 warp → 128 bytes of consecutive memory
 - Aligned to 128-byte boundary...

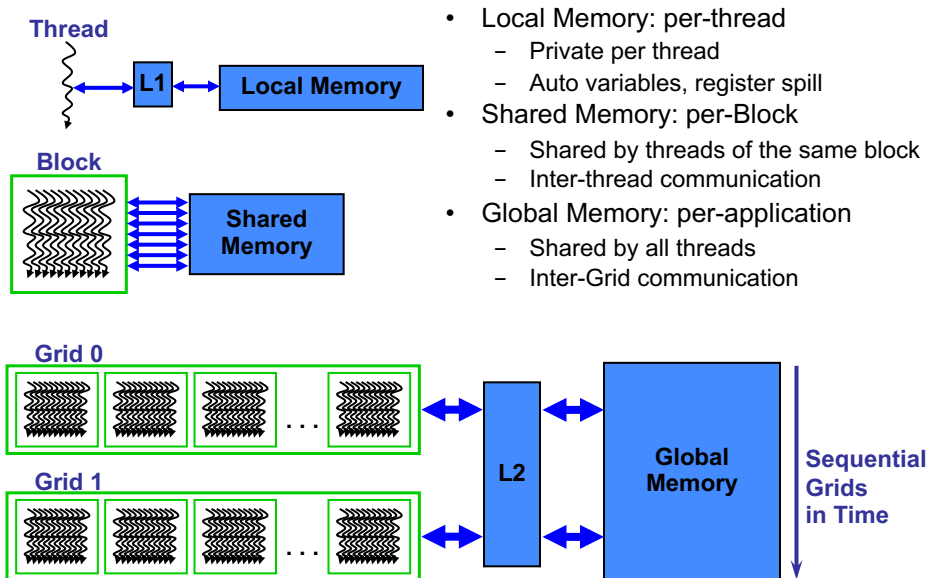
14

Parallelism in the memory system



15

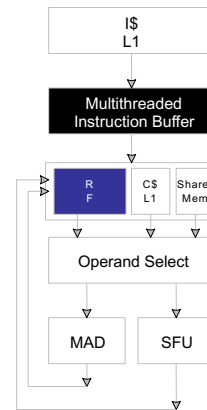
Parallelism in the memory system – with caches



16

SM register file

- Register File (RF)
 - 256KB
 - 64K 32-bit registers
- Texture pipe can also read/write RF
 - 2 Texture (TEX) clusters per SM
- Load/Store pipe can also read/write RF



17

Programmer's view of register file

- There are 64K registers in each SM in GTX680
 - This is an implementation decision, not part of CUDA
 - Registers are dynamically partitioned, like shared memory
 - Partitioned across all Blocks assigned to the SM
 - Registers are **thread-private**, unlike shared memory
 - Once assigned to a thread, NOT accessible by other threads
 - Each thread only accesses registers assigned to itself



18

Register use implications example

- **Matrix Multiplication**
- If each Block has 16X16 threads and each thread uses 42 registers, how many threads can run on each SM?
 - Each Block requires $42 \times 16 \times 16 = 10752$ registers
 - $65536 / 10752 = 6.09\dots$
 - So, **six** blocks can run on an SM as far as registers are concerned
- How about if each thread increases the use of registers by 1?
 - Each Block now requires $43 \times 16 \times 16 = 11008$ registers
 - $65536 / 11008 = 5.95\dots$
 - Only **five** Blocks can run on an SM
 - **5/6 parallelism, i.e., 17% parallelism loss**

19

Dynamic partitioning

- Dynamic partitioning: more flexibility to compilers/programmers
 - e.g., run a smaller number of threads that require many registers each
 - e.g., run a large number of threads that require few registers each
 - Allows for finer grain threading than traditional CPU threading models
 - The compiler can tradeoff between instruction-level parallelism (ILP) and thread level parallelism (TLP)

20

Instruction-Level Parallelism (ILP) example

Program:

$a = a + (*bmem)$

$c = c + 10 * b$

Original register allocation:


a, b, c in r2, r4, r5, respectively

bmem addr in r3, **r1 is a temp.**

$r2 = r2 + \text{memory}(r3)$

$r5 = r5 + 10 * r4$

load r1, 0(r3)
add r2, r2, r1
mul r1, r4, 10
add r5, r5, r1



- Add cannot proceed
– **Waits for load**
- Blue instructions cannot overlap load because they use **r1** (the load's destination register)
– **dependency**

21

Instruction-Level Parallelism (ILP) example

Program:

$a = a + (*bmem)$

$c = c + 10 * b$

Original register allocation:

a, b, c in r2, r4, r5, respectively

bmem addr in r3, **r1 is a temp.**


$r2 = r2 + \text{memory}(r3)$

$r5 = r5 + 10 * r4$

load r1, 0(r3)
add r2, r2, r1
mul r1, r4, 10
add r5, r5, r1

Say we add one more register:

load r1, 0(r3)
add r2, r2, r1
mul r6, r4, 10
add r5, r5, r6



blue vs. red
can execute
in any order

22

Instruction-Level Parallelism (ILP) example

Original version:

```
load r1, 0(r3)
add r2, r2, r1
mul r1, r4, 10
add r5, r5, r1
```

- Add cannot proceed
 - Waits for load
- Blue instructions cannot overlap load because they use **r1** (the load's destination register)
 - dependency

One more register:

```
load r1, 0(r3)
mul r6, r4, 10
add r5, r5, r6
add r2, r2, r1
```

- Extra register breaks dependency
 - Blue instructions no longer use register r1
- Overlap memory access time (load) with computation (blue instr.)

23

Within or across Thread Parallelism (ILP vs. TLP)

- Device: 64K registers per SM, 32 SPs per SM int cluster, 6 int clusters (cap.=3.0), global loads: 400 cycles
- Kernel: 128-thread Blocks, **42** registers per thread
- Program: **40** independent instructions between consecutive global memory loads
- Warps / block = ? $128/32 = 4$
- #blocks can run on each SM = ? $\lfloor (64K/42)/128 \rfloor = 12$
- #warps can run on each SM = ? $12 * 4 = 48$
- #warps to tolerate mem latency (**one int cluster**) = ? $\lceil 400/40 \rceil = 10$
- #warps to tolerate mem latency (**all int clusters**) = ? $10 * 6 = 60$
- Will memory latency be exposed? Yes! $48 < 60$
- Memory latency will be exposed, performance degrades!

24

Within or across Thread Parallelism (ILP vs. TLP)

- Device: 64K registers per SM, 32 SPs per SM int cluster, 6 int clusters (cap.=3.0), global loads: 400 cycles
- Kernel: 128-thread Blocks, **43** registers per thread
- Program: **60** independent instructions between consecutive global memory loads
- Warps / block = ? $128/32 = 4$
- #blocks can run on each SM = ? $\lfloor (64K/43)/128 \rfloor = 11$
- #warps can run on each SM = ? $11 * 4 = 44$
- #warps to tolerate mem latency (**one int cluster**) = ? $\lceil (400/60) \rceil = 7$
- #warps to tolerate mem latency (**all int clusters**) = ? $7 * 6 = 42$
- Will memory latency be exposed? No! $44 > 42$
- Conclusion: could be better!
- If a compiler can use **one more register** to change the dependence pattern so that **60** independent instructions exist for each global memory load, we could hide all memory latency!

25

Load/Store clustering/batching

- Use LD to hide LD latency (non-dependent LDs only)
 - Use same thread to help hide own latency
- Instead of:
 - LD A0 (long latency)
 - Dependent MATH on A0
 - LD A1 (long latency)
 - Dependent MATH on A1
- Do:
 - LD A0 (long latency)
 - LD A1 (long latency - hidden)
 - MATH on A0
 - MATH on A1
- Compiler handles this!
 - But, you must have enough **non-dependent LDs and Math**

26

How many resources is my kernel using?

- **NVCC flag: `--ptxas-options=-v`**
- ptxas info : Compiling entry function 'acos_main'
ptxas info : Used 4 registers, 60+56 bytes lmem,
44+40 bytes smem, 20 bytes cmem[1], 12 bytes cmem[14]
- For shared memory per block (smem):
 - 44 bytes explicitly allocated by the user
 - 40 were implicitly allocated by the system/compiler
- Local memory (lmem):
 - local memory per thread
- Constant memory (cmem):
 - Program variables [1]
 - Compiler generated constants [14]
- To control register usage, NVCC flag: **`--maxrregcount=N`**
 - N=desired maximum registers / kernel
 - At some point “spilling” into LMEM may occur
 - Reduces performance because LMEM is slow (global DRAM)

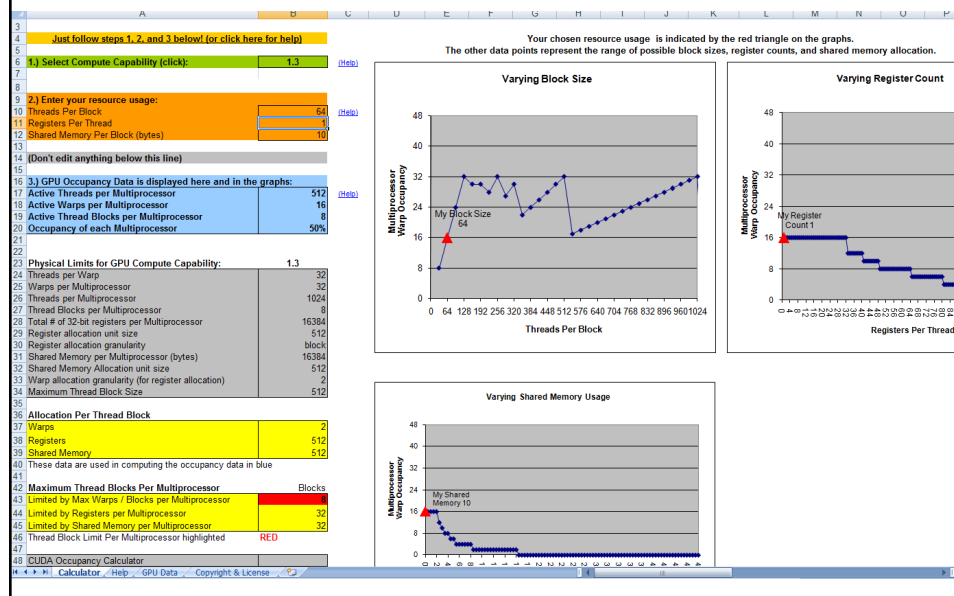
27

CUDA warp occupancy calculator

Included in the SDK

Multiprocessor Warp Occupancy = (# active Warps) / (# maximum active Warps)

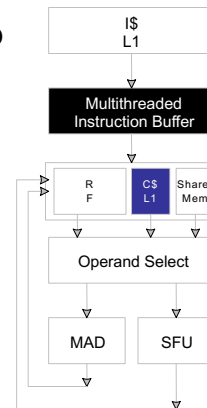
Goal: Maximize SM warp memory latency



28

Constants

- e.g., `__constant__ int ConstVar;`
- Constants stored in DRAM, cached on chip
 - L1 per SM
 - 64KB total in DRAM
- A constant value can be broadcast to all threads in a Warp
 - Extremely efficient way of accessing a value that is common for all threads in a Block
- Examples of constants
 - Immediate address constants
 - Indexed address constants
 - Image fading factor



29

How to get high performance #3

- Control flow
 - 32 threads run (logically) together
 - One instruction is broadcast to all threads
 - If they diverge there is a performance penalty


```

if (array[threadIdx.x])
    foo( );
else
    bar( );
                    
```

Which instruction to broadcast if half the threads want to execute foo() and the other half bar() ?
 - Goal: eliminate or at least minimize divergence

30

How to get high performance #4

- Think carefully of numerical accuracy needs
- Can do single-precision floating point (32-bit)
 - Mostly OK, with some minor discrepancies
- Can do double-precision floating point (64-bit)
 - 1/8 to 1/24 the throughput
 - GTX680: 8 FP64 cores vs. 192 FP32 cores → 1/24
 - Better on newer hardware
- Mixed methods
 - Break numbers into two single-precision values
 - Must carefully check for stability/correctness
- Goal: use double-precision only when necessary
 - Gets better at successive HW generations

31

Are GPUs really that much faster than CPUs?

- 50x – 200x speedups typically reported
- Prior work argued that:
 - Not enough effort goes into optimizing code for CPUs
 - If it was, performance gap would narrow
 - Intel paper (ISCA 2010)
 - http://portal.acm.org/ft_gateway.cfm?id=1816021&type=pdf
 - GPUs “only” 3x better on average
- But:
 - The learning curve and expertise needed for CPUs is much larger
 - Then, so is the potential and flexibility
 - 3x is still a big gap

32

Reading Assignment

- Please read Lecture 4.b on Canvas
- It offers a little more information on
 - CUDA vector types
 - Error handling functions and macros
 - Execution time measurement methodology
 - Handling large data sets
 - Why CUDA makes no guarantees on block/thread execution order?
 - Floating point considerations
 - Tools
 - Compilation
 - Linking
 - Debugging
 - Profiling