CUDA: Performance Considerations

Mark Greenstreet & Ian M. Mitchell

CPSC 418 – March 17, 2017

- Thread Divergence
- Floating Point Foibles
- Memory Accesses
- Occupancy
- Granularity

Thread Divergence

- If threads in a warp are following different code paths, execution will be much slower.
- See “A Warped Example” from March 13 slides.

Try to minimize thread divergence within warps.

Remarks about floating point

- When working on my solution to last year's HW3, Q1,
  - I first wrote:

```
x = alpha*x*(1.0 - x);
```

  - and the performance was disappointing.
  - After many frustrating attempts to track down the problem, I added one, little `f`:

```
x = alpha*x*(1.0f - x);
```

  - and my code ran 5.5× faster.
- What happened?

Floats, doubles, and GPUs

- GPUs are optimized for single-precision floating point arithmetic.
- For the GeForce GTX 550 Ti, double precision arithmetic is way slower than single precision.
- In C, `1.0` is a **double precision** constant, and `1.0f` is single precision.
- When I wrote `x = alpha*x*(1.0-x)`, the compiler generated code that:
  - computes the product `alpha*x`.
    - both operands are single precision.
    - the computation is done using single precision arithmetic.
  - computes the difference `1.0-x`
    - 1.0 is double precision, x is single precision.
    - the computation is done using double precision arithmetic
    - and the result is double precision.
  - computes the product `alpha*x`*(1.0-x).
    - the computation is done using double precision arithmetic
    - and the result is double precision.
- When I wrote `x = alpha*x*(1.0f-x)`, everything stays in single-precision, and it's **much** faster.

Fused multiply adds

- Calculating $ax + b$ is very common
  - Example: dot product.
- The multiplier hardware is just a pipeline of adders.
  - When multiplying `a*x`, the hardware can start the pipeline from `b` instead of from `0`.
  - We get the sum for "free".
  - This is called a **fused** multiply-add.
- The marketing people like to count the fused multiply-add as **two** floating point operations.
  - This helps make some performance claims make sense.
- For the obsessive compulsive:
  - Rounding with a fused-multiply add can be slightly different than when doing two, separate operations.
  - Compilers usually let the users specify "strict" floating point (no fusing) or "fast" floating point (with fusing).
  - `nvcc` uses fused multiply add unless you give it an option not to.

Memory Access

Memory is slow.

- It takes a long time to identify, access and deliver data to/from a memory address.
- Delivery rate is limited by clock rate of the memory interface.

How can we get enough data to/from our thousands of threads?
Parallelism!

- Retrieve lots of data at once.
- Use multiple memory interfaces.
- Build with lots of independent memory components.

All standard techniques in the CPU world, but

- CPU design philosophy: Try to achieve maximum performance even if the programmer uses the RAM model.
- GPU design philosophy: Expose (almost) everything and let the programmer figure it out.

Memory System Parallelism 1: Get Lots of Data

Memory is addressed per byte, but you retrieve a bunch of (sequential) bytes at once.

- GDDR5 DRAM: 32-bit bus per chip and transfers are in 16 word bursts (so 64 bytes per access per chip).
- GPU global memory (from GDDR DRAMs): Accessed by 32-, 64- or 128-byte transactions.
  - Transactions must be "naturally" aligned: First address must be a multiple of the transaction size.
  - CC 2.x: L1 cache (1 per SM) serviced by 128-byte transactions, L2 cache (shared by SMs) by 32-byte transactions.
  - CC 6.x: Same as 2.x, but L1 cache rules are complicated.
- GPU shared memory (on-chip SRAM): Access in 32-bit words.

Amortize addressing overhead and thereby increase bandwidth.

Memory System Parallelism 2: Multiple Interfaces

If one memory component cannot give you enough bandwidth, use a bunch (see March 13 slides).

- Global memory: K&H(3) calls these “channels” (March 13 slide 2).
- Shared memory: Mark called these “banks” (March 13 slide 11) and NVidia documentation does too.
  - Do not confuse with K&H(3) “banks” (see next slide).

Consecutive chunks are placed into components in a round-robin fashion, where “chunk” means

- 32-bytes (64 more recently?) in global memory.
- 32- or 64-bits in shared memory.

Separate subsystems can all provide data at their native rate and thereby increase bandwidth.

Memory System Parallelism 3: Independent Memory Components

Even after memory address is delivered, it still takes time for the DRAM to return the data.

- Rather than let the memory bus sit idle while waiting, pipeline a bunch of memory requests to different memory components.
  - K&H(3) calls these “banks”.
  - Mark called these “tiles”.
- Consecutive memory chunks are assigned first to channels / banks (see previous slide).
  - These subsystems allow concurrent access because they have independent communication lines.
- Then assign next set of consecutive chunks to banks / tiles.
  - These subsystems allow sequential but pipelined access because they share communication lines

Pipelining increases throughput (although latency remains).

- Only relevant for global memory.
- Shared memory achieves dramatically lower latency with SRAM.

Is This on the Final?

No (sort of): This is not a course on memory system design and implementation.

- Mark and I are far from experts.
- Details depend on the particular GPU chip and card, and change regularly.
- Program correctness does not depend on getting it right.

Yes (sort of): By following some simple rules, speed can be improved dramatically.

- Design global memory access pattern to allow accesses from threads in the same warp to be **coalesced** into a single memory transaction.
- Design memory access pattern to avoid channel / bank / tile **collisions**.

Implications for Shared Memory

See CUDA Toolkit Documentation C Programming Guide Figure 17 and Figure 18.

- Consider shared memory address bits:
  - 48KB / thread block requires 16 bits to address.
  - Bottom two bits specify the byte within a 32-bit word of data.
  - Next five bits specify which of 32 banks.
  - Top nine bits specify which word within the bank.
- Key takeaway: If two threads in a warp access a memory location in the same bank (same middle five bits of address):
  - If threads access the same location (same top nine bits), then broadcast (on read) or one value wins (on write).
  - If threads access different location, access is serialized (slower but still correct).

Implications for Global Memory

Try to get memory access addresses from threads in a warp to be very close together.

- Accesses to consecutive (or nearly so) addresses are coalesced into a single transaction on the off-chip memory bus.
  - You should already be doing this for your CPU designs so that your caches can take advantage of spatial locality.
- Best coalescing occurs when the set of addresses is naturally aligned.
  - For two and higher dimensional arrays, that may mean padding thread block and array width allocation in memory to be a multiple of the warp size.
- Possibility of channel / bank collisions would argue for avoiding addresses with the same “middle” bits.
  - I could not find NVidia documentation of these details.
  - How do caches interact with channels / banks?

Comments from Mark?

SMs and Thread Occupancy

- Occupancy: how many warps are available for the SM
  - Why we care: the SP pipelines have long latencies.
  - The CUDA approach is to run lots of threads simultaneously to keep the pipelines busy.
- Limits to occupancy
  - How many blocks per SM.
  - How much shared-memory per block.
  - How many threads per block.
  - How many registers per thread.
- Figuring it out
  - `nvcc -O3 -c --ptxas-options -v examples.cu`
  - The nVidia occupancy calculator: CUDA_Occupancy_calculator.xls
  - But we can do it manually?

Occupancy with CUDA 2.1

- Different GPUs at level CUDA 2.1 have differing numbers of SMs.
  - But the SMs all look the same, even for different GPUs.
- CUDA 2.1 SMs
  - An SM has warps of 32 threads
  - An SM can simultaneously execute up to 1536 threads (48 warps).
  - An SM has 32K (2^{15}) 32-bit registers (128K/bytes, 1K registers/SP).
  - An SM has 48K bytes of shared memory.
  - An SM can simultaneously execute up to 8 blocks.
  - Each block can have up to 1024 threads.

Why all these numbers?

- When designing a new generation of GPUs, the GPU architects run lots of simulations to estimate the performance for various choices of the architectural parameters.
- For example, if more warps are allowed in the scheduling pool
  - The SM will have useful instructions to dispatch more often $\Rightarrow$ better performance.
  - **BUT** the on-chip circuitry to hold and manage the scheduling pool will be larger.
  - This means instruction scheduling will be slower $\Rightarrow$ a longer clock period.
  - Instruction scheduling will use more power $\Rightarrow$ a longer clock period, or fewer SMs, or more expensive chip cooling.
  - The real-estate on the chip could have been used for something else. Is this the **best** use of that area.
  - Note that CUDA 5 made the increase to 64 warps/SM.
- Architects explore these trade-offs to optimize performance for graphics applications, the main source of revenue.
- Architects are also risk-adverse: make the chip as much like the last one that worked as you can.
- These hard-wired constraints have a large impact on program performance.

SMs, blocks, and threads

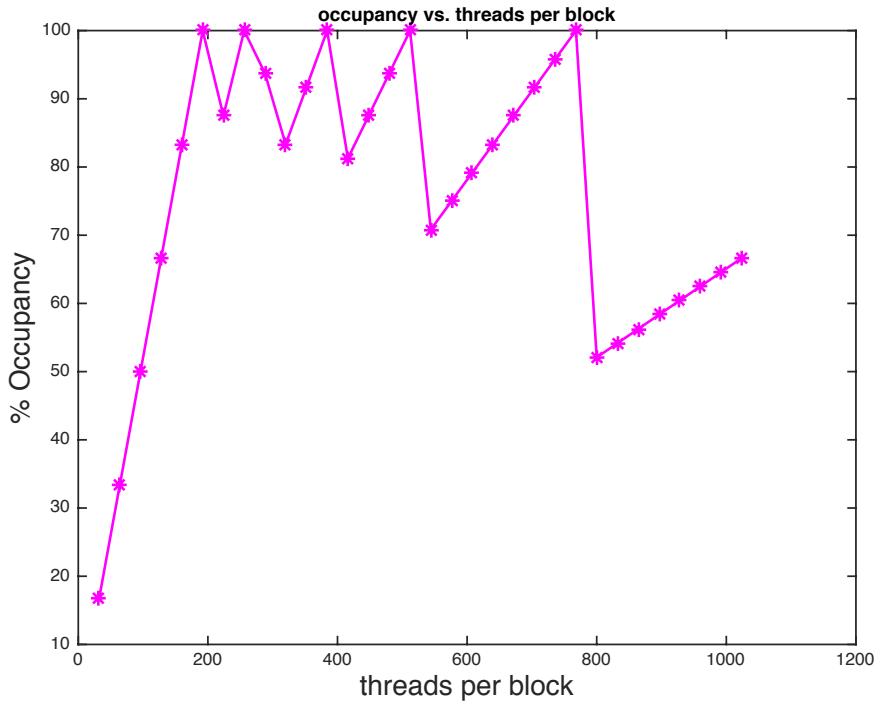
- A SM can have simultaneously execute most 8 blocks.
- All blocks have the same number of threads.
- Thus, a SM can execute at most

$$\min\left(8, \left\lfloor \frac{1536}{threadsPerBlock} \right\rfloor\right)$$

  blocks.
- The ratio of the number of threads executing to the maximum possible is called the "thread occupancy":

$$threadOccupancy \leq \min\left(8, \left\lfloor \frac{1536}{threadsPerBlock} \right\rfloor\right) \frac{threadsPerBlock}{1536}$$

SMs, blocks, and threads – the plot



- I get 100% occupancy when $threadsPerBlock \in \{192, 384, 768\}$, but the CUDA calculator doesn't.
  - I'll have to try some experiments – stay tuned.
- This assumes the grid had enough blocks to keep the SMs busy.
  - A grid with a single block will have poor performance.

SMs, threads, and registers

- Each SM has 32K registers – that's 1K registers per SP.
- This is another constraint:

$$nblks \leq \frac{1024}{registersPerThread}$$

- An SM can run 48 warps simultaneously
  - But only if each warp uses at most 21 registers.

Hitting the register constraint

What if each thread uses 22 registers?

- $22 * 48 = 1056 > 1024 \rightarrow$ can't run 48 warps.
- $\left\lfloor \frac{1024}{22} \right\rfloor = \lfloor 46.\overline{54} \rfloor = 46.$
- Can we run 46 warps?
  - One block with 46 warps would have $46 * 32 = 1472 > 1024$ threads. Not allowed.
  - Two block with 23 warps each would each have 736 threads. That should work.
  - But, the plot with the occupancy calculator only shows warp counts that are multiples of 8.
  - Have I overlooked another architectural constraint?
    - ★ probably
- Let's assume that with 23 registers per thread, the SM can run at most 40 warps simultaneously.
  - Then either each thread must have enough instruction-level parallelism to keep the SPs busy.
  - Or, we'll see a drop in performance.

How many registers does my thread use?

- use the `--ptxas-options -v` option for `nvcc`

```
nvcc--ptxas-options -v -O3 -c examples.cu
ptxas info :   0 bytes gmem
ptxas info :   Compiling entry function '_Z8sh_mem_2jiiPj' for 'sm_20'
ptxas info :   Function properties for _Z8sh_mem_2jiiPj
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info :   Used 17 registers, 4096 bytes smem, 56 bytes cmem[0]
ptxas info :   Compiling entry function '_Z8sh_mem_1jiiPj' for 'sm_20'
ptxas info :   Function properties for _Z8sh_mem_1jiiPj
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info :   Used 14 registers, 4096 bytes smem, 56 bytes cmem[0]
```

- Translation:
  - kernel `sh_mem_2` uses 17 registers per thread.
  - kernel `sh_mem_1` uses 14 registers per thread.
  - both kernels use 4024 bytes of shared memory per block.
  - neither kernel spills registers to global memory (good).

Granularity

How much work should a kernel do?

- Do more work within a kernel: Launching each kernel takes time.
- Do less work within a kernel: New kernels allow for changes in block and grid size, and ensure synchronization between threads even in different blocks.
- Either way: Minimize movement of data to and from the host.

How much work should a thread do?

- Do more work in a single thread: Fewer chances for memory collisions, easier synchronization, less register contention.
- Do less work in a single thread: More potential parallelism, more chance for latency hiding.
- Tradeoff will depend on GPU resources, typically SM block, thread and register limits.

Bigger Kernels

```
__global__myKernel(...)  {
    do something
}
```

Unless *do something* is big, kernel launch takes most of the time.

- We can launch a big-grid
  - If we have a huge number of array elements than each need a small amount of work, this can be a good idea.
  - BUT we're likely to create a memory-bound problem.
- Or, we can make each thread do many somethings.

```
__global__myKernel(int m, ...)  {
    for(int i = 0; i < m; i++)
        do something
}
```

Loop Limitations

- It takes two or three instructions per loop iteration to manage the loop:
  - One to update the loop index
  - One or two to check the loop bounds and branch.
  - If *do something* is only three or four instructions, then 40-50% of the execution time is for loop management.
- If each iteration of *do something* depends on the previous one
  - Then the long latency of the SP pipelines can limit performance.
  - Even if we have 48 warps running.

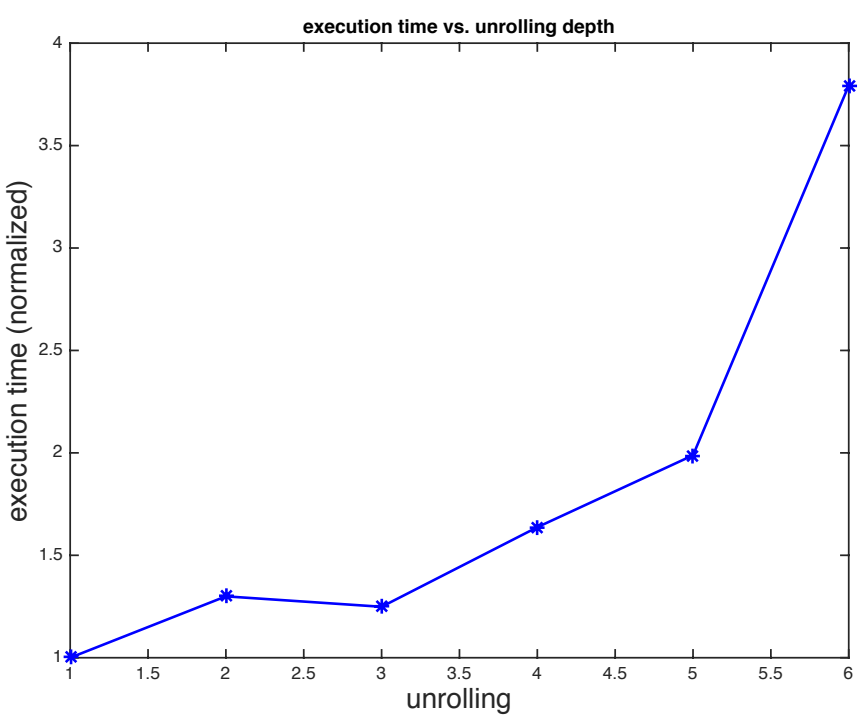
Loop Unrolling

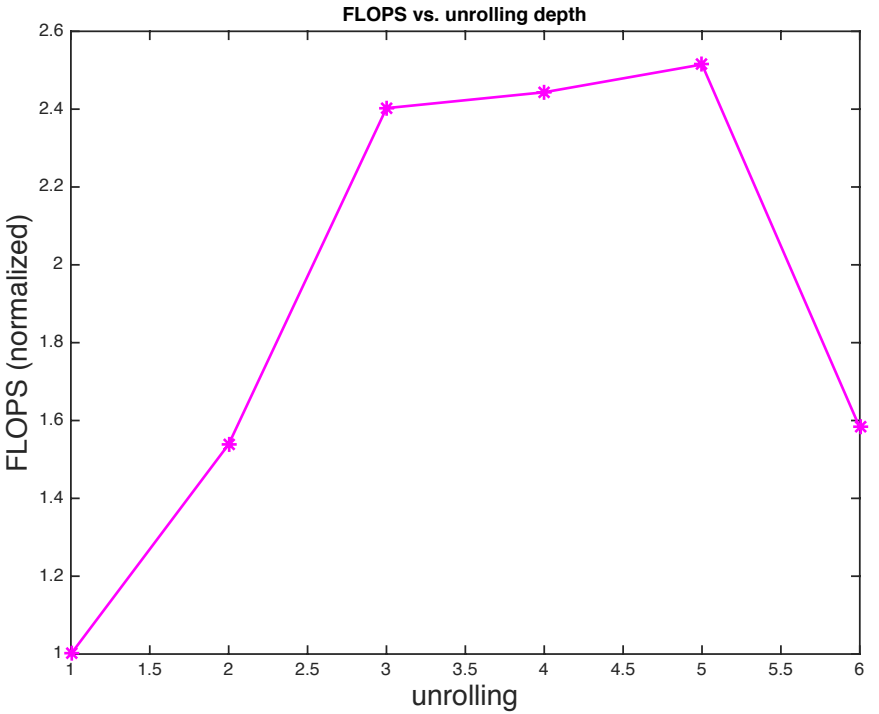
- Have each loop iteration perform multiple copies of the loop body

```
__global__myKernel(int m, ...)  {
    for(int i = 0; i < m; i += 4) {
        do something 1
        do something 2
        do something 3
        do something 4
    }
}
```

- More “real work” for each time the loop management code is executed.
- Need to make sure that `m` is a multiple of four, or handle end-cases separately.
- Often, we need more registers.

Unrolling – the plots





This example is from last year's HW3, Q1.

Where's λ?

- Communication between the CPU and GPU
  - Kernel launch overhead
  - Transfering data between CPU memory and GPU memory
    - ★ Is this solved with more recent GPUs that can access the CPU memory directly?
    - ★ Not really, the data still needs to be transfered.
    - ★ And it's one more memory level for the programmer to keep track of.
- Communication between blocks
  - Write global memory and end the kernel.
  - Launch a new kernel and read the global memory.
  - The same strategy applies if the shape for the required grid changes between phases of a larger computation.
- Communication between warps in a block
  - `__syncthreads__`
- **AND**,
  - There's a built-in energy cost of the big register file.
  - Trade-offs of energy, latency, and parallelism. large numbers of threads.

Preview

March 20: Matrix multiplication, Part 1
March 22: Matrix multiplication, Part 2
March 24: Complete CUDA
March 27 – April 3: this may change
March 27: Using Parallel Libraries
March 29 – April 3: Verification of/and Parallel Programs
April 5: Party: 50^{th} Anniversary of Amdahl's Law