

CUDA: Performance Considerations

Mark Greenstreet

CpSc 418 – Mar. 18, 2016

- Floating Point Foibles
- Shared Memory Accesses
- Global Memory Accesses
- Occupancy
- Instruction Mix

Remarks about floating point

- When working on my solution to [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 $5.5 \times$ 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 generate 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.
 - ▶ Note: this doesn't affect the problems for HW3.

Shared Memory

See the [March 16](#) lecture.

- Shared memory is fast, on-chip memory.
- Shared memory is much faster than global memory.
 - ▶ Global memory: coalescing references
 - ▶ Other memories on the GPU
- An example, and lessons learned
 - ▶ The example: shared-memory bank conflicts
 - ▶ Lessons learned

Global memory: coalescing references

- GPUs have relatively high off-chip memory bandwidth
 - ▶ compared with CPUs
 - ▶ still much slower than accessing registers or shared-memory with good interleaving.
- If all the warps in a thread access consecutive locations in the same load, the GPU can maximize the memory much faster than with random accesses.
- I tried modifying the code from examples.cu (March 16).
 - ▶ global memory is definitely slower than registers or shared memory.
 - ▶ coalesced accesses to global memory are faster than worst-case bank collisions with the shared memory.
 - ▶ I need to do more experiments to understand the effects of the on chip caches.

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/thread).
 - ▶ 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.
 - ▶ Architects explore these trade-offs to optimize performance for graphics applications, the main source of revenue.
 - ▶ Architects are also risk-averse: make the chip as much like the last one that worked as you can.
- Does it matter?
 - ▶ When writing low-level code, e.g. C/CUDA, you see all of these choices

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

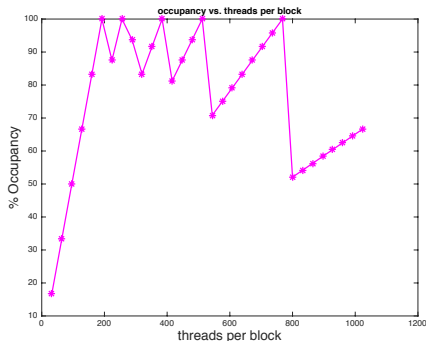
$$\min \left(8, \left\lfloor \frac{1536}{threadsPerBlock} \right\rfloor \right)$$

blocks.

- The ratio of the number of threads executing to the maximum possible is called the “thread occupancy”:

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

SMs, blocks, and threads – the plot



SMs, threads, and registers

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

$$nblks \leq \frac{1024 \text{ registers}}{\text{Thread}}$$

- 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.
- $\lfloor \frac{1024}{22} \rfloor = \lfloor 46.\overline{54} \rfloor = 46$.
- 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' f
```

```
ptxas info :    Function properties for _Z8sh_mem_2jiiPj
```

```
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill
```

```
ptxas info :    Used 17 registers, 4096 bytes smem, 56 bytes
```

```
ptxas info :    Compiling entry function '_Z8sh_mem_1jiiPj' f
```

```
ptxas info :    Function properties for _Z8sh_mem_1jiiPj
```

```
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill
```

```
ptxas info :    Used 14 registers, 4096 bytes smem, 56 bytes
```

- 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).

Instruction Mix

- We measure our program performance in terms of the critical, unavoidable operations
 - ▶ Typically “floating point operations” for matrix-multiplication or other scientific computing applications.
 - ▶ Often main memory accesses for sorting, or other data-intensive applications.
- But, the program does other operations as well
 - ▶ This is where you see me counting instructions on my fingers during lecture.
 - ▶ Optimizing performance can involve minimizing this overhead:
 - ★ Good algorithm design.
 - ★ Memory access optimization.
 - ★ Loop unrolling

Loop Limitations

- Initial code:

```
__global__ myKernel(...) {  
    do something  
}
```

- Unless *do something* is big, kernel launch takes most of the time. So, make each thread do many somethings.

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

- Two, typical performance limits if *do something* is simple.
 - ▶ 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

