CSE 746 - Parallel and High Performance Computing Lecture 6 - Profiling CUDA

Pawel Pomorski, *HPC Software Analyst*

SHARCNET, University of Waterloo

ppomorsk@sharcnet.ca

http://ppomorsk.sharcnet.ca/

Profiling

- profiling is examining code during its execution, to better understand its performance and find ways to improve it
- essential when developing computationally intensive codes
- as parallel codes are generally computationally intensive, they almost always benefit from profiling
- so far, to develop a fundamental understanding, we have used explicit timing calls, both from CUDA and standard C
- for more complex codes this would become very inconvenient
- fortunately, CUDA provides a set of powerful profiling tools
- profiling tools can provide significantly more information than just timings

Profiling tools in CUDA

- command line profiler: nvprof
  - very fast and convenient
- visual profiler: nvvp
  - can be used on its own to run executable and do profiling
  - can analyze output previously generated with nvprof
- Nsight IDE
  - has profiler built in
  - more cumbersome to use if you are only profiling, as it requires a project to be set up for any code to be analyzed
  - standalone nvvp provides a lot of functionality of Nsight profiler and it's easier to use

nvprof

- Documentation at: http://docs.nvidia.com/cuda/profiler-users-guide/
- available from command line
- will work on standard CUDA executable generated with nvcc, no special flags need to be supplied to nvcc to enable profiling
- basic usage (here test.x is the executable to be profiled):
  nvprof test.x
- more detailed output listing events on GPU in order of execution
  nvprof --print-gpu-trace   test.x
- more detailed output listing CUDA calls on host
  nvprof --print-api-trace   test.x

Important: compile for right architecture

- It's especially important when profiling to use the right -arch flag when compiling with nvcc, so it matches the architecture that the executable will be profiled on
- clearly, since performance optimization is important when profiling, one does not want to introduce any inefficiency by using executable targeted at wrong architecture
- also, if wrong architecture is selected, some of the profiling diagnostics will be inconsistent

CUDA concepts for today's lecture

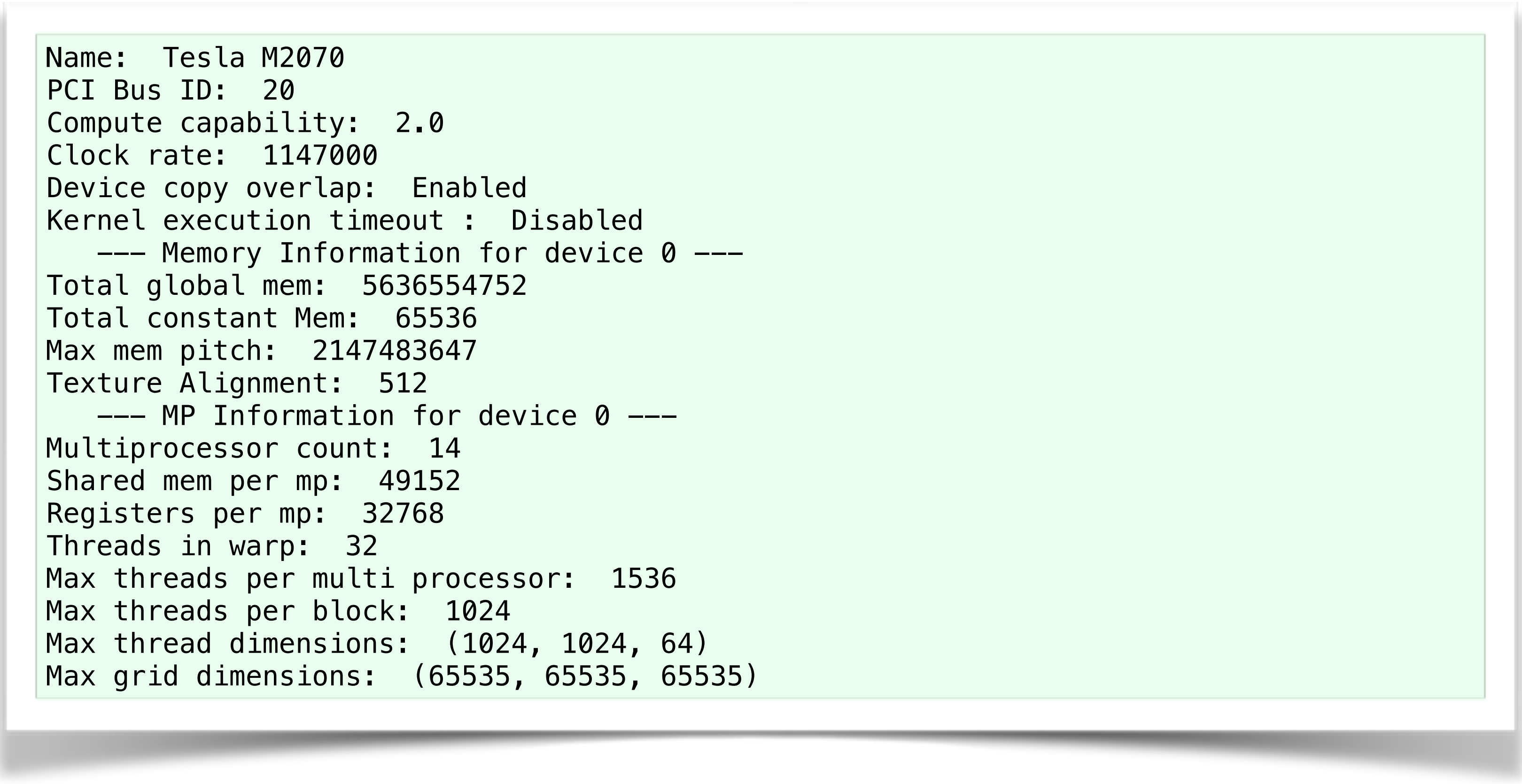
- **thread occupancy** ; getting the most out of the GPU by getting the optimum number of threads actively running on it at any given time
- **register pressure** ; limitations on occupancy and memory efficiency when kernel requires a significant number of registers
- **warp divergence** ; performance penalties if threads in the same warp are not executing identical code
- these will be studied with help of profiling tools

Occupancy

- when you execute a CUDA kernel with N threads, generally only a subset of these N threads are resident i.e. have their instructions actively executed at any one time. The remainder have either :
  - already finished executing their instructions
  - are still waiting to begin executing their instructions
- each multiprocessor inside a CUDA GPU has a hardware limit on the maximum number of threads which can be resident at any one time
- this limit is one of the properties which can be queried at runtime (with *maxThreadsPerMultiProcessor*)

Device diagnostic output

- from device_diagnostic.cu in lecture code
- output on mon54



```
Name:  Tesla M2070
PCI Bus ID:  20
Compute capability:  2.0
Clock rate:  1147000
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
   --- Memory Information for device 0 ---
Total global mem:  5636554752
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
Multiprocessor count:  14
Shared mem per mp:  49152
Registers per mp:  32768
Threads in warp:  32
Max threads per multi processor:  1536
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (65535, 65535, 65535)
```

Device diagnostic output

- from device_diagnostic.cu in lecture code
- output on mosaic.sharcnet.ca, node mos1

```
Name:  Tesla K20m
PCI Bus ID:  8
Compute capability:  3.5
Clock rate:  705500
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
   --- Memory Information for device 0 ---
Total global mem:  5032706048
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
Multiprocessor count:  13
Shared mem per mp:  49152
Registers per mp:  65536
Threads in warp:  32
Max threads per multi processor:  2048
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

Device diagnostic output

- from device_diagnostic.cu in lecture code
- output on minsky.uwo.sharcnet

```
Name:  Tesla P100-SXM2-16GB
PCI Bus ID:  1
Compute capability:  6.0
Clock rate:  1480500
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
   --- Memory Information for device 0 ---
Total global mem:  17071669248
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
Multiprocessor count:  56
Shared mem per mp:  49152
Registers per mp:  65536
Threads in warp:  32
Max threads per multi processor:  2048
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

Occupancy

- the ratio between the actual number of threads that are resident during execution and the theoretical maximum number of threads that could be resident on the GPU
- broadly speaking, if your occupancy is too low, that indicates inefficiency in your code and should be remedied. Low occupancy may indicate:
  - not all available CUDA cores being used
  - insufficient number of threads for latency hiding (i.e. compensating for slow global memory access by threads)
- on the other hand, for many codes you should not aim to achieve occupancy close to 1.0, as code will run sufficiently well even at lower occupancy as it is bound by other limitations

Occupancy - blocks and warps

- it’s also important to note that there is a similar limit on how many **blocks** of threads can be resident at any given time. This limit is:
  - 8 on devices with CC 2.0 (monk)
  - 16 on devices CC 3.5 (mosaic)

  This is why your occupancy will be low if your blocks are too small.
- may also have a problem if blocks too large, since it’s not possible to have only a subset of block resident
- analogously, the maximum number of resident warps is 48 for CC 2.0 and 64 for CC 3.0 and 3.5
- it may be more natural to look at warp occupancy i.e. how many warps are resident relative to the maximum

Reasons for occupancy < 1.0

- blocks too small, so that (maximum number of possible resident blocks) times (block size) is less than the theoretical limit
- (number of threads in block) does not divide the theoretical limit evenly (eg. block of 1024 threads on GPU with limit 1536)
- thread blocks consume too much of limited resources (shared memory and registers) so that not enough blocks can be resident to achieve full occupancy
- Note: in all this we assumed for simplicity only one kernel is running at any one time. Multiple concurrent kernels would reduce occupancy per kernel, though possibly enhance it overall.

Estimating occupancy

- occupancy can be estimated even before you run your code. This can often allow the programmer to pick the optimum grid and block dimensions
- useful tool is here in form of spreadsheet: http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls
  - performance somewhat buggy on Mac and Linux
- the occupancy calculated this way is the upper limit possible for your code - the actual measured occupancy will be slightly lower, due to overheads associated with launching threads

Exercise 1

- profile the starting code provided, using nvprof
- this code contains one simple kernel which copies memory inside the GPU, hence the bound on its performance is bandwidth to global memory of the GPU
- note the execution time and occupancy as a function of block size (these can be powers of 2)
- to see the occupancy , use nvprof with
  ```
  nvprof --print-gpu-trace --metrics achieved_occupancy  ./test.x
  ```
- profile on mon54, mosaic (mos1) and minsky
- be careful to compile for right architecture

Limited multiprocessor resources

- the amount of shared memory in a multiprocessor is limited
- if each block of a kernel uses some shared memory, then the total amount used by resident blocks at any one time cannot exceed the total shared memory available
- so, the use of shared memory may reduce the number of resident blocks and hence the number of resident threads, reducing the occupancy
- if number of resident blocks is too low (worst case: just one), then this can put a fundamental limit on occupancy. Thus on GPU in *mosaic* this would mean maximum possible occupancy of 0.5, as only 1024 threads (block maximum) can be resident at once if only one block is resident, and that is only half of the theoretical limit

Exercise 2

- in code from exercise 1, allocate a shared memory region for each block in your kernel, sufficiently large that it uses just over half of the available shared memory. Measure impact on performance.
- can be done in 3rd argument when invoking kernel <<< , , N>>> where N is the amount in bytes of shared memory to allocate
- in such code only one block will be resident on the GPU at any one time
- this addition of unneeded shared memory region is artificial in this program, but it will serve to simulate kernels that would actually use shared memory
- now reduce amount of shared memory to just below half of the total shared memory available to see how occupancy changes

Instruction parallelism

- thread parallelism is achieved by running with many threads. But what happens if we are limited in how many threads are resident at the same time?
- can compensate with instruction parallelism - giving threads more work to do, more instructions to perform

Exercise 3

- in code from previous exercise, increase instruction parallelism by having each thread perform more than just one copy. Ensure that in your modified code memory accesses are still coalesced

Load - store

- performance of kernels implementing instruction parallelism can be further improved by using thread local storage
- this means replacing:

```
output[i]=input[i]   // two global memory arrays

with


small loop
   temp[]  =input[]    //temp is thread-local
small loop
   output[]=temp[]
```

Load - store

- staging helps because in a thread the load operation is non blocking. If a line of code issues an instruction access a location in global memory (load operation) and stores it in a thread-local location **tmp**, this instruction is non-blocking
- the thread can proceed on to other instructions without waiting for the load operation to complete, until reaching instruction which uses **tmp**
- therefore, by using staging through a small tmp array, one can overlap multiple load operations, even in a single thread
- in this way performance can be improved for memory bandwidth limited code, even when occupancy is low

Thread-local memory

- this is memory for variables which are declared inside a thread. Each thread has its own individual copy of such a variable visible only to itself
- where does this memory physically reside? That depends.
  - ideally, it will all reside in registers for the duration of thread lifetime

Registers

- registers constitute the fastest possible memory located right on the multiprocessors. All memory data must go through them.
- Unfortunately, the number of (32-bit) registers on a multiprocessor is limited (32 K on CC 2.0, 64 K on CC 3.0 and 3.5) and they must be shared by all resident threads.
- So the number of registers each individual thread can get is limited to total number divided by number of resident threads. Additionally, there is an upper limit set by Compute Capability (63 on CC 2.0 and 3.0, 255 on CC 3.5).

Register spills

- what happens when the thread-local data is too big to fit in registers allocated to a thread?
- the data will then be "spilled" i.e. moved out from register to local memory
- the "spill" first attempts to store in L1, if that's not possible, it stores in L2, and if that's not possible, it stores in global memory
- when thread tries to get back "spilled" data, it will first try to find it in L1, if it's not there, it will try L2, and if it's not there, it will try global memory
- while accesses to L1 are fast, since is located right on the multiprocessor in same space as shared memory, accesses to L2 and global memory are slow

Register usage

- number of registers per thread is determined by compiler
  - compiler can output register usage if invoked with
  nvcc -Xptxas -v
- compiler decides how many registers to use. Typically it tries to keep the number used to a minimum, in order to increase occupancy

Register usage

- you can try to force the compiler to use fewer registers with the flag
  nvcc -maxrregcount=Nmax
  where Nmax is the maximum number of registers. This is a rather crude way to attempt to optimize as one should usually let the compiler decide, but it can be used for testing
- there is a better way which gives compiler hints on how many registers should be used, by adding to kernel definitions the parameters:
  _launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)

Register spilling

- the compiler will report how many times it has to spill registers for each thread, and how much local memory it is using
- the number of registers used will also be reported by the profiler
- be sure you profile with right architecture to get these numbers to match with each other
- but just knowing the number of register spills is not sufficient to indicate just how bad they are for performance, as that will depend on whether the spill goes only to L1, or further to L2, or furthest to global memory
- the nature of the spilling can be determined during profiling

Exercise 4

- take code from previous exercise and introduce a thread-local tmp array to stage copies, study how it affects performance
- vary the size size of tmp array to see how register and local memory usage reported by compiler changes

Impact of register spilling

- some small amount of register spilling may not be so bad, especially if spills only go to fast L1 cache
- problem arises if significant number of spills go to L2 cache. Not only is L2 cache slow, it is need for accesses to global memory, so having it spend a significant amount of time handling spills will impact the effective bandwidth to global memory
- also, spilling data to registers and then recovering it requires additional instructions in compiled code. More instructions means kernel runs slower. This is only likely to matter in codes which execute a lot of instructions per thread.

Impact of register spilling on L2 cache

- nvprof profiler can provide very specific counts for many events. To see all available, execute:
  nvprof --query-events
- to count events describing L1 cache access, you can run:
  nvprof --events l1_local_load_hit,l1_local_load_miss,l1_local_store_miss,l1_local_store_hit
- the ratio of hits and misses indicates how often data is located in fast L1 cache, and how often data is not found in L1 cache (and so has to be looked for in slow L2 cache)
- if you have lots of misses relative to hits, that is bad for performance

Impact of register spilling on L2 cache

- Too see quantitatively how bad, you have to measure what portion of L2 accesses is due to register spilling
- using nvprof and events is awkward, so use nvpp
- first run nvprof with:
  nvprof --analysis-metrics -o test.nvprof ./test.x
- the log into a machine that has nvvp installed, using an ssh connection with graphics support (ssh -X)
- start nvvp , get graphical interface
- import the output file generated by nvprof
- examine the output, one of the columns will be the ratio in question

Reducing impact of spilling to L2

- try increasing register count at compile stage (occupancy decrease)
- Make L1 larger. Default is 16 kB, with 48 kB shared memory. But you can switch to have L1 48 kB and shared 16 kB. If you are not using any shared memory, this is a very good idea.
- can do this for device with: cudaDeviceSetCacheConfig(cudaFuncCachePreferL1);
- can try non-caching loads, with -Xptxas -dlcm=cg . This means global memory accesses will avoid using L1 cache. The tradeoff is that access to global memory is slower, but there is more L1 cache to handle register spilling.
- (as an aside, non-caching load brings in data in 32-byte chunks, caching in 128-byte chunks. Sometimes one is better than other.)

Exercise 5

- use nvprof output in the nvvp visual profiler
- study cases where there is register spilling
- observe the fraction of L2 memory accesses caused by spilling
- try some of the suggested techniques to see impact on performance

Warp divergence

- warp divergence occurs when not all threads in a warp are executing identical code. This is typically caused by *if* statements that send threads inside a warp on different execution paths
- when warp divergence occurs, execution can become serialized, i.e. the threads in warp cannot execute in parallel (together at the same time). This slows down the calculation.
- different warps executing different code are not a problem
- different blocks executing different code are not a problem, since warps do not span across blocks

Warp divergence

- if divergence is “small” (eg. different threads multiply value by a different constant) then performance penalty will be negligible
- if divergence is significant, with threads going on completely different instruction paths, then code will generally serialize, with performance dropping by the factor roughly equal to the number of divergent paths
- just to emphasize, this will happen even if the divergent paths do the same amount of computational work

Exercise 6

- take the code we started the exercises with and introduce divergence, in this case by performing one instruction in half of the threads and another in another half
- first, have the even and odd threads multiply the input array by different constant factors. Compare performance with non-diverged code.
- second, have the even and odd threads multiply the array by sin(float(i)) and cos(float(i)), respectively. Compare performance.
- to improve performance, assume array is rearranged, so that:
  - you apply same operations but to even/odd blocks
  - you apply same operations but to even/odd warps