Advanced CUDA topics – Part 2

Optimization: Assess performance

Pierre-Yves Taunay

Research Computing and Cyberinfrastructure
224A Computer Building
The Pennsylvania State University
University Park
py.taunay@psu.edu

July 17, 2013

PENNSTATE
1855

Outline

Outline

- Quick introduction
- Tools for evaluating the performance of an application

Introduction

Objectives

- Understand optimization steps and cycle
- Understand what general metrics are available to the CUDA programmer
- Determine the *kernel mode* of a CUDA kernel
- Choose the right metric for measuring performance of a CUDA application
- Familiarize yourself with the CUDA Profiler
- Obtain finer metrics with the CUDA Profiler

Optimization cycle

- HPC code development follows an APOD cycle[1]
  - ↪ Assess
  - ↪ Parallelize
  - ↪ Optimize
  - ↪ Deploy
- **Remark**: subject to diminishing returns

PENNSTATE 1855

Optimization steps – 1/5

- Get it to work first
- Evaluate the performance: use of pertinent metrics
- Target specific routines
- Check your results

Optimization steps – 2/5

- Get it to work first
  - $\hookrightarrow$ Covered in the debugging seminar

Optimization steps – 3/5

- Evaluate the performance of the application
  - $\hookrightarrow$ Time measurements
  - $\hookrightarrow$ Gold standard comparison
  - $\hookrightarrow$ Kernel mode
  - $\hookrightarrow$ NVIDIA Profiler
- Use the right metrics: bandwidth is not relevant for a kernel not limited by data transfer

PENNSTATE 1855

Optimization steps – 4/5

- Target specific routines, and apply optimization techniques
  - $\hookrightarrow$ Compiler optimization
  - $\hookrightarrow$ Architecture–specific optimizations

Optimization steps – 5/5

- Check your results
  - ↪ New code will likely introduce bugs
  - ↪ Save old results and compare to new results
  - ↪ Add error handling for production code
  - ↪ Use RCS to roll back to previous code version if too many issues

Performance evaluation: general metrics

Time measurements

Timing routines by hand

- First metric: **time spent**: *how long does each part of the code take to execute ?*
  - ↪ CUDA Profiler can show the time spent
  - ↪ Possible to obtain that “by hand” too: includes the kernel invocation overhead
- Use OpenMP or CUDA routines to assert the time spent

Time measurements

Timing routines by hand with OpenMP

- Can use OpenMP to time host functions and device kernels
- Make sure to add **cudaDeviceSynchronize** after kernel calls for accurate time measurements

```
#include <omp.h>

int main() {
   ...
   double begin,end = 0.0;
   begin = omp_get_wtime();
   // Insert routine to time here
   end = omp_get_wtime();
   printf("Time spent in routine: %.15f\n",end-begin);
   ...
}
```

PENNSTATE 1855

Time measurements

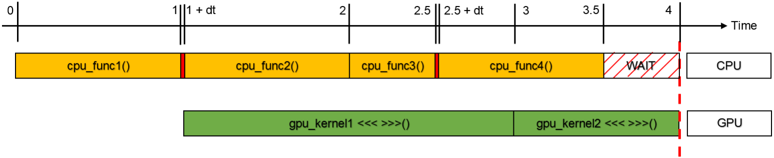
Timing routines by hand with OpenMP

- To compile OpenMP code with NVCC, simply use the -Xcompiler flag
  nvcc -Xcompiler -fopenmp main.cu
- -Xcompiler (“cross-compiler”) is used in general for host code options
- **Remark**: remember that CUDA kernels are by nature **asynchronous**. Need a call to **cudaDeviceSynchronize()** after a kernel call to get relevant timing

PENNSTATE 1855

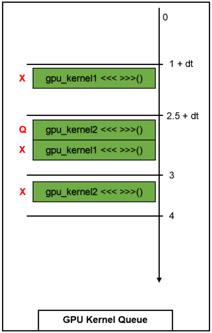
Time measurements

Clarification of asynchronous behavior



[Some_program.cu]

```
cpu_func1();
gpu_kernel1 <<< >>> ();
cpu_func2();
cpu_func3();
gpu_kernel2 <<< >>> ();
cpu_func4();
cudaDeviceSynchronize();
```



Time measurements

Timing routines by hand with CUDA

```
#include <cuda.h>

int main() {
    ...
    cudaEvent_t start, stop;
    float time;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord( start, 0 );
    mykernel<<<grid,threads>>> (...);
    cudaEventRecord( stop, 0 );
    cudaEventSynchronize( stop );
    cudaEventElapsedTime( &time, start, stop );
    cudaEventDestroy( start );
    cudaEventDestroy( stop );
    ...
}
```

Comparison to gold standard

- Second metric: **speed-up**: *how does my code compare to CPU or previous GPU versions ?*
  - ↪ Compare total GPU time with a pure CPU routine
  - ↪ $SU = \frac{t_{GPU}}{t_{CPU}}$
- To time the total time of an application with system calls, use **time**

```
time ./myprogram
real 0m0.156s
user 0m0.007s
sys 0m0.127s
```

- You want to use "real" timings
- For a very good explanation of real, user, and sys time, see Ref. 2

Other general metrics

- Bandwidth: total memory exchanged in the kernel divided by total time to complete the memory instructions
  - ↪ Easy to obtain by commenting out everything but the memory instructions
- Latency: number of clock cycles for a *warp* to execute one instruction
  - ↪ Example: access to global memory has a 200-300 cycles latency

Performance evaluation: kernel mode

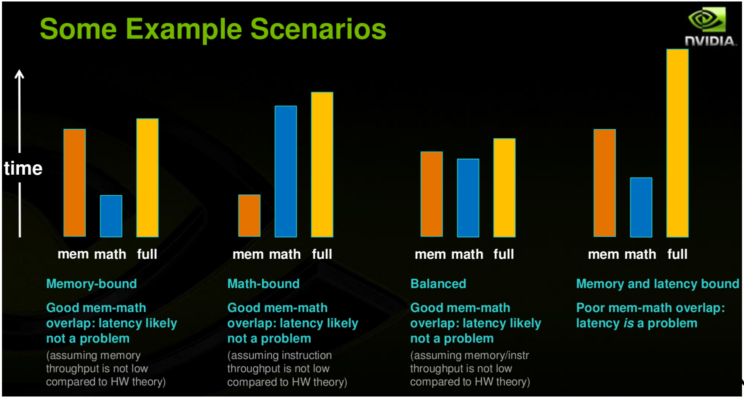
PENNSTATE

Kernel mode – 1/5

- Is a kernel spending most of its time in R/W operations, or is it spending most of its time doing computations ?
  - $\hookrightarrow$ Useful information as to what can be done for optimization
  - $\hookrightarrow$ Also gives insights on which metric to use
- First mode: memory bound
- Second mode: compute bound
- Third mode: balanced

Kernel mode – 2/5

- Example scenarios from Ref. 3



Kernel mode – 3/5

Determining the kernel mode

- Determine the time spent in memory R/W operations
  - ↪ Comment out as much arithmetic as possible, without modifying access patterns
  - ↪ Check with the profiler that the number of R/W instructions is the same as normal code: use **gld_request** and **gls_request** events
- Determine the time spent in arithmetic
  - ↪ Harder since the compiler **deletes** kernels that do not do any R/W operations
  - ↪ Have to trick the compiler by using conditional statements that are never met (see next slide)...
  - ↪ ... the compiler is however getting smarter, so have to provide "intricate" conditions

Kernel mode – 4/5

Determining the kernel mode

- Here is an example, inspired by Ref. 3

```
__global__ void mykernel(..., int flag){
  ...
  int myvar = some_previous_var + some_other_var;
  if( myvar*flag == -1 ) {
    global_vector[idx] = myvar;
  }
}
```

- if(myvar*flag == -1) is necessary. The compiler otherwise moves the computation of **myvar** in the conditional[3].

PENNSTATE 1855

Kernel mode – 5/5

Exercise

- Determine the kernel mode of each of the kernels listed in the file *kernel-mode_exercise/kernel-mode_exercise.cu*
- A Makefile is provided for convenience
- The code will have to be modified/commented to measure the time taken by memory–only operations or math–only operations

Performance evaluation: profiler

PENNSTATE 1855

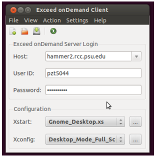
NVIDIA Profiler

- **Extremely** useful profiling tool
- Able to show a lot of meaningful data about a CUDA application: cache hits, bandwidth, etc.
- Further seminars will show how to make sense of the data provided

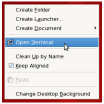
NVIDIA Profiler

Invocation – 1/2

- NVVP uses a GUI: need X-forwarding
- 1. Use Exceed on Demand to connect to Hammer



- 2. Open a terminal on Hammer to SSH to Lion–GA with X forwarding



PENNSTATE 1855

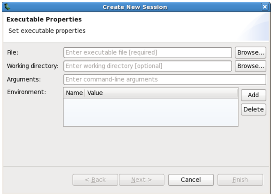
NVIDIA Profiler

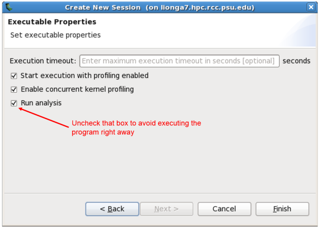
Invocation – 2/2

- 3. Connect to Lion–GA with X–forwarding
  - ↪ `[pzt5004@hammer2~] ssh -X -Y lionga`
- 4. On Lion–GA, submit an interactive job with X–forwarding
  - ↪ `qsub -X -I -l nodes=1:ppn=1:gpus=1,walltime=1:00:00`
- 5. You should now be on a particular lion–GA node. Load the CUDA module, and launch the NVIDIA profiler
  - ↪ `[pzt5004@lionga1~] module load cuda/5.0`
  - ↪ `[pzt5004@lionga1~] nvvp`

NVIDIA Profiler

Setting up a new session

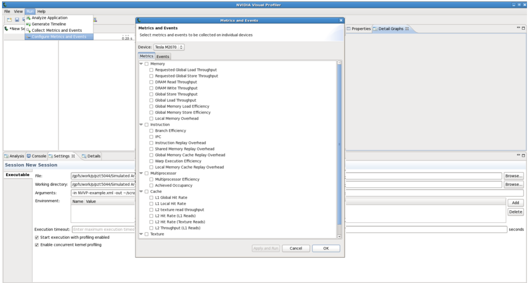




NVIDIA Profiler

Adding metrics and events

- Go to **Run** >**Configure metrics and events**
- Add / Remove any metrics and events



- Most of the metrics are measured per **Streaming Multiprocessor** (SM)
- Hover over a metric to get a brief explanation

NVIDIA Profiler

Performing a run

- Once metrics have been chosen, click **Apply and run**
- Or close the window, then choose **Run** $>$**Collect metrics and events**
- The program will be run multiple times since not all metrics can be collected at once

PENNSTATE 1855

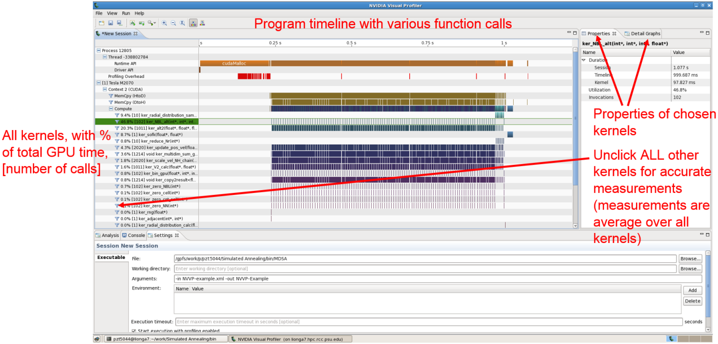
NVIDIA Profiler

Predefined analysis

- Predefined analyses can be found in the **Analysis** tab
- The profiler will also give you advice based on the results of each analysis stage

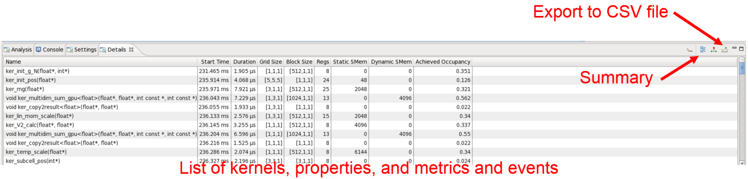
NVIDIA Profiler

Making sense of the data – 1/2



NVIDIA Profiler

Making sense of the data – 2/2



Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Achieved Occupancy
ker_init_g_N(float*, int*)	231.465 ms	1.905 µs	[1,1,1]	[512,1,1]	8	0	0	0.351
ker_init_pos(float*)	235.914 ms	4.068 µs	[5,5,5]	[1,1,1]	24	48	0	0.126
ker_rng(float*)	235.971 ms	7.921 µs	[3,1,1]	[512,1,1]	25	2048	0	0.321
void ker_multidim_sum_gpu<float>(float*, float*, int const *, int const *)	236.043 ms	7.229 µs	[1,3,1]	[1024,1,1]	13	0	4096	0.562
void ker_copy2result<float>(float*, float*)	236.055 ms	1.933 µs	[1,3,1]	[1,1,1]	8	0	0	0.022
ker_lin_mom_scale(float*)	236.133 ms	2.576 µs	[1,3,1]	[512,1,1]	15	2048	0	0.34
ker_V2_calc(float*, float*)	236.145 ms	3.255 µs	[1,1,1]	[512,1,1]	8	4096	0	0.337
void ker_multidim_sum_gpu<float>(float*, float*, int const *, int const *)	236.204 ms	6.596 µs	[1,1,1]	[1024,1,1]	13	0	4096	0.55
void ker_copy2result<float>(float*, float*)	236.216 ms	1.525 µs	[1,1,1]	[1,1,1]	8	0	0	0.022
ker_temp_scale(float*)	236.286 ms	2.074 µs	[1,1,1]	[512,1,1]	8	6144	0	0.34
ker_subcell_pos(int*)	236.327 ms	2.196 µs	[3,3,1]	[3,1,1]	8	0	0	0.024

NVIDIA Profiler

Exercise

- Get familiar with the NVVP Profiler on Lion–GA by profiling LAMMPS, a molecular dynamics application readily available
- The exercise instructions are included in the file README.txt in the folder *nvvp_exercise*

Conclusion

- This section presented the answer to “how do I evaluate my GPU code efficiency and quality ?”
- Next sections will be focused on GPU architecture and on priority targets for optimization

PENNSTATE 1855

References and further reading

References

[1] CUDA C Best Practices Guide
[2] http://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1
[3] Advanced Topics in CUDA; Cliff Woolley, NVIDIA

The CUDA documentation is located at
/usr/global/cuda/5.0/doc/pdf on Lion–GA

PENNSTATE 1855

Further reading

- Wilt, N. *The CUDA Handbook*
- Hwu, W.-M., and Kirk, D., *Programming Massively Parallel Processors*
- Sanders, J., and Kandrot, E., *CUDA by Example*
- CUDA C Best Practices
- CUDA C Programming Guide

PENNSTATE 1855