Harness the Power of GPUs:
An Introduction to GPGPU Programming
Lecture 9: Parallelization Techniques and Optimization

Guido Juckeland
Visiting Scholar
Technische Universität Dresden, Germany

June 20, 2014
Goals for parallelization

• Shorten compute time
• Solve larger problems
• More precise solutions

• In sum: Require more computation speed
Good starting point

- Very large problem (spatial)
- Complex problem (multiple components)
Implementation steps

- Identify parallelism in the problem
- Design an algorithm that exploits the parallelism
- Implement the algorithm
- Performance tuning
Challenges

• Identify parallelism
• Determine dependencies
• Performance is a fragile thing
  – Overhead for parallelization
  – Load imbalances
  – Insufficient data reuse
  – Insufficient resources (e.g. memory bandwidth)
Problem distribution

- Distribution into tasks
  - Determine concurrent activity (CUDA streams, OpenACC async)
  - Put enough work into tasks (overhead)
- Data distribution
  - Same task for different parts of the overall data
  - Determine local data per task
- Group and order tasks
- Determine common data
- Consider and judge alternatives
Process

Decomposition
- Task Decomposition
- Data Decomposition

Dependence Analysis
- Group Tasks
- Order Tasks
- Data Sharing

Design Evaluation

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE408, University of Illinois, Urbana-Champaign
Example: Problem distribution

- C = A * B of SIZE * SIZE
  - One task computes one element of C
  - All tasks are independent and can be executed concurrently

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE408, University of Illinois, Urbana-Champaign
Grouping of tasks

- Group of tasks compute a segment of P
  - Every input element will be used multiple times (better usage of memory bandwidth)
  - Threads in the group synchronize their work (overhead)
Selecting a “good” algorithm

- Things to consider
  - Iterations/steps until a solution is found
  - Parallelism in the algorithm
  - Computational intensity (compute operations per memory access)
- Good strategies
  - Tiling of data
  - Gather instead of scather
  - Double buffering
  - Cutoffs
Finding performance problems

- Profile your application using nvprof

$ nvprof ./laplace2d_acc
Jacobi relaxation Calculation: 4096 x 4096 mesh
==10619== NVPROF is profiling process 10619, command: ./laplace2d_acc
...
total: 134.259326 s
==10619== Profiling application: ./laplace2d_acc
==10619== Profiling result:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>49.59%</td>
<td>44.0095s</td>
<td>17000</td>
<td>2.5888ms</td>
<td>864ns</td>
<td>2.9822ms</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>45.06%</td>
<td>39.9921s</td>
<td>17000</td>
<td>2.3525ms</td>
<td>2.4960us</td>
<td>2.7687ms</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>2.95%</td>
<td>2.61622s</td>
<td>1000</td>
<td>2.6162ms</td>
<td>2.6044ms</td>
<td>2.6319ms</td>
<td>main_56_gpu</td>
</tr>
<tr>
<td>2.39%</td>
<td>2.11884s</td>
<td>1000</td>
<td>2.1188ms</td>
<td>2.1023ms</td>
<td>2.1374ms</td>
<td>main_68_gpu</td>
</tr>
<tr>
<td>0.01%</td>
<td>12.431ms</td>
<td>1000</td>
<td>12.430us</td>
<td>12.192us</td>
<td>12.736us</td>
<td>main_63_gpu_red</td>
</tr>
</tbody>
</table>
Finding performance problems (2)

- Profile your application using PGI tools

```
$ PGI_ACC_TIME=1 ./laplace2d_acc
Accelerator Kernel Timing data
/home/jlarkin/openacc-workshop/exercises/001-laplace2D-parallel/laplace2d.c
  main NVIDIA devicenum=0
time(us): 89,242,926
  56: compute region reached 1000 times
  56: data copyin reached 8000 times
device time(us): total=22,334,806 max=3,022 min=2,747 avg=2,791
  56: kernel launched 1000 times
  grid: [4094]  block: [256]
device time(us): total=2,643,298 max=2,841 min=2,629 avg=2,643
  elapsed time(us): total=2,654,729 max=2,855 min=2,640 avg=2,654
  56: reduction kernel launched 1000 times
  grid: [1]  block: [256]
device time(us): total=19,182 max=75 min=17 avg=19
  elapsed time(us): total=29,669 max=87 min=28 avg=29
  68: data copyout reached 8000 times
device time(us): total=20,100,500 max=2,797 min=2,494 avg=2,512
...```
NVIDIA Visual Profiler

The NVIDIA Visual Profiler is a tool that helps you trace the execution flow of your application from host to device memory and back. It is designed to help you find and understand performance bottlenecks in your application. The tool provides a detailed view of the execution timeline, including thread activity, device command execution, and memory access patterns. This information can be used to optimize your application's performance by identifying areas where the execution is stalling or where the memory is being accessed inefficiently.
Vampir for analyzing all level of parallelism
Typical performance problems

- **Accuracy**
  - Consider also the accuracy of your input data
- **Bad thread distribution**
- **Poor resource usage**
- **Non-coalesced memory accesses**
- **Not enough work per thread**
- **Branch divergence**
Thread execution

Size does matter!

Logical 2-D organization

Linear order

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE408, University of Illinois, Urbana-Champaign
Programmer View of Register File

- There are 64k registers in each SM in GK110
  - This is an implementation decision, not part of CUDA
  - Registers are dynamically partitioned across all Blocks assigned to the SM
  - Once assigned to a Block, the register is NOT accessible by threads in other Blocks
  - Each thread in the same Block only access registers assigned to itself

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE 408, University of Illinois, Urbana-Champaign
Matrix Multiplication Example

- If each Block has 16X16 threads and each thread uses 40 registers, how many threads can run on each SM?
  - Each Block requires $40 \times 256 = 10240$ registers
  - $32768 = 3 \times 10240 + \text{change}$
  - So, three blocks can run on an SM as far as registers are concerned

- How about if each thread increases the use of registers by 4?
  - Each Block now requires $44 \times 256 = 11264$ registers
  - $32768 < 11264 \times 3$
  - Only two Blocks can run on an SM, $\frac{1}{3}$ reduction of thread-level parallelism (TLP)!!!
Memory layout of a 2D matrix in C
Memory Coalescing

- When using global memory, best performance is reached when neighboring threads access neighboring data
Memory layout of a 2D matrix in C
Memory layout of a 2D matrix in C

Access direction in the kernel

Time Period 1

Time Period 2

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010

ECE408, University of Illinois, Urbana-Champaign
Using shared memory to improve coalescing

Original access pattern

Blocked access pattern

Copy data to shared memory

Carry out multiplication with data from shared memory

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE 408, University of Illinois, Urbana-Champaign