Lecture 2.4 – Introduction to CUDA C

Introduction to the CUDA Toolkit
Objective

– To become familiar with some valuable tools and resources from the CUDA Toolkit
  – Compiler flags
  – Debuggers
  – Profilers
GPU Programming Languages

- Numerical analytics
  - MATLAB, Mathematica, LabVIEW
- Fortran
  - CUDA Fortran
- C
  - CUDA C
- C++
  - CUDA C++
- Python
  - PyCUDA, Copperhead, Numba, NumbaPro
- F#
  - Alea.cuBase
CUDA - C

Applications

Libraries
- Easy to use
- Most Performance

Compiler Directives
- Easy to use
- Portable code

Programming Languages
- Most Performance
- Most Flexibility
NVCC Compiler

- NVIDIA provides a CUDA-C compiler
  - `nvcc`
- NVCC compiles device code then forwards code on to the host compiler (e.g. g++)
- Can be used to compile & link host only applications
Example 1: Hello World

```c
int main() {
    printf("Hello World!\n");
    return 0;
}
```

Instructions:
1. Build and run the hello world code
2. Modify Makefile to use nvcc instead of g++
3. Rebuild and run
CUDA Example 1: Hello World

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Instructions:
1. Add kernel and kernel launch to main.cu
2. Try to build
CUDA Example 1: Build Considerations

- Build failed
  - Nvcc only parses .cu files for CUDA
- Fixes:
  - Rename main.cc to main.cu
  OR
  - nvcc -x cu
    - Treat all input files as .cu files

Instructions:
1. Rename main.cc to main.cu
2. Rebuild and Run
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Output:

```bash
$ nvcc main.cu
$ ./a.out
Hello World!
```

- `mykernel` *(does nothing, somewhat anticlimactic!)*
Developer Tools - Debuggers

NSIGHT
CUDA-GDB
CUDA MEMCHECK

NVIDIA Provided

allinea DDT
TotalView

3rd Party

Compiler Flags

- Remember there are two compilers being used
  - NVCC: Device code
  - Host Compiler: C/C++ code

- NVCC supports some host compiler flags
  - If flag is unsupported, use –Xcompiler to forward to host
    - e.g. –Xcompiler –fopenmp

- Debugging Flags
  - -g: Include host debugging symbols
  - -G: Include device debugging symbols
  - -lineinfo: Include line information with symbols
CUDA-MEMCHECK

- Memory debugging tool
  - No recompilation necessary
    -> cuda-memcheck ./exe
- Can detect the following errors
  - Memory leaks
  - Memory errors (OOB, misaligned access, illegal instruction, etc)
  - Race conditions
  - Illegal Barriers
  - Uninitialized Memory
- For line numbers use the following compiler flags:
  - -Xcompiler -rdynamic -lineinfo

http://docs.nvidia.com/cuda/cuda-cuda-memcheck
Example 2: CUDA-MEMCHECK

Instructions:
1. Build & Run Example 2
   Output should be the numbers 0-9
   Do you get the correct results?
2. Run with cuda-memcheck
   `> cuda-memcheck ./a.out`
3. Add nvcc flags “-Xcompiler -rdynamic -lineinfo”
4. Rebuild & Run with cuda-memcheck
5. Fix the illegal write

http://docs.nvidia.com/cuda/cuda-memcheck
CUDA-GDB

- cuda-gdb is an extension of GDB
  - Provides seamless debugging of CUDA and CPU code
- Works on Linux and Macintosh
  - For a Windows debugger use NSIGHT Visual Studio Edition

http://docs.nvidia.com/cuda/cuda-gdb
Example 3: cuda-gdb

Instructions:
1. Run exercise 3 in cuda-gdb
   %> cuda-gdb --args ./a.out
2. Run a few cuda-gdb commands:
   (cuda-gdb) b main //set break point at main
   (cuda-gdb) r     //run application
   (cuda-gdb) l     //print line context
   (cuda-gdb) b foo //break at kernel foo
   (cuda-gdb) c     //continue
   (cuda-gdb) cuda thread //print current thread
   (cuda-gdb) cuda thread 10 //switch to thread 10
   (cuda-gdb) cuda block //print current block
   (cuda-gdb) cuda block 1 //switch to block 1
   (cuda-gdb) d       //delete all break points
   (cuda-gdb) set cuda memcheck on //turn on cuda memcheck
   (cuda-gdb) r       //run from the beginning
3. Fix Bug

http://docs.nvidia.com/cuda/cuda-gdb
Developer Tools - Profilers

NSIGHT

NVVP

NVPROF

NVIDIA Provided

TAU

VampirTrace

3rd Party

NVPROF

Command Line Profiler
– Compute time in each kernel
– Compute memory transfer time
– Collect metrics and events
– Support complex process hierarchy's
– Collect profiles for NVIDIA Visual Profiler
– No need to recompile
Example 4: nvprof

Instructions:
1. Collect profile information for the matrix add example
   \%> nvprof ./a.out
2. How much faster is add_v2 than add_v1?
3. View available metrics
   \%> nvprof --query-metrics
4. View global load/store efficiency
   \%> nvprof --metrics
gld_efficiency, gst_efficiency ./a.out
5. Store a timeline to load in NVVP
   \%> nvprof -o profile.timeline ./a.out
6. Store analysis metrics to load in NVVP
   \%> nvprof -o profile.metrics --analysis-metrics ./a.out
NVIDIA’s Visual Profiler (NVVP)

Timeline

Guided System

Analysis
Example 4: NVVP

Instructions:
1. Import nvprof profile into NVVP
   - Launch nvvp
   - Click File/ Import/ Nvprof/ Next/ Single process/ Next / Browse
   - Select profile.timeline
   - Add Metrics to timeline
   - Click on 2\textsuperscript{nd} Browse
   - Select profile.metrics
   - Click Finish
2. Explore Timeline
   - Control + mouse drag in timeline to zoom in
   - Control + mouse drag in measure bar (on top) to measure time
Example 4: NVVP

Instructions:
1. Click on a kernel
2. On Analysis tab click on the unguided analysis

2. Click Analyze All
   Explore metrics and properties
   What differences do you see between the two kernels?

Note:
If kernel order is non-deterministic you can only load the timeline or the metrics but not both.
If you load just metrics the timeline looks odd but metrics are correct.
Example 4: NVVP

Let’s now generate the same data within NVVP

1. Click File / New Session / Browse
   Select Example 4/a.out
   Click Next / Finish

2. Click on a kernel
   Select Unguided Analysis
   Click Analyze All
NVTX

- Our current tools only profile API calls on the host
  - What if we want to understand better what the host is doing?
- The NVTX library allows us to annotate profiles with ranges
  - Add: `#include <nvToolsExt.h>`
  - Link with: `-lnvToolsExt`
- Mark the start of a range
  - `nvtxRangePushA("description");`
- Mark the end of a range
  - `nvtxRangePop();`
- Ranges are allowed to overlap

NVTX Profile
NSIGHT

- CUDA enabled Integrated Development Environment
  - Source code editor: syntax highlighting, code refactoring, etc
  - Build Manager
  - Visual Debugger
  - Visual Profiler

- Linux/Macintosh
  - Editor = Eclipse
  - Debugger = cuda-gdb with a visual wrapper
  - Profiler = NVVP

- Windows
  - Integrates directly into Visual Studio
  - Profiler is NSIGHT VSE
Example 4: NSIGHT

Let’s import an existing Makefile project into NSIGHT

Instructions:
1. Run nsight
   Select default workspace
2. Click File / New / Makefile Project With Existing CodeTest
3. Enter Project Name and select the Example15 directory
4. Click Finish
5. Right Click On Project / Properties / Run Settings / New / C++ Application
6. Browse for Example 4/a.out
7. In Project Explorer double click on main.cu and explore source
8. Click on the build icon
9. Click on the run icon
10. Click on the profile icon
Profiler Summary

- Many profile tools are available
- NVIDIA Provided
  - NVPROF: Command Line
  - NVVP: Visual profiler
  - NSIGHT: IDE (Visual Studio and Eclipse)
- 3rd Party
  - TAU
  - VAMPIR
Optimization

- Assess
- Parallelize
- Optimize
- Deploy
Assess

- Profile the code, find the hotspot(s)
- Focus your attention where it will give the most benefit
Parallelize

Applications

Libraries

Compiler Directives

Programming Languages
Optimize

Guided System

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound

Timeline

<table>
<thead>
<tr>
<th>Tesla K40c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context MPS (CUDA)</td>
</tr>
<tr>
<td>Memcpy (HtoD)</td>
</tr>
<tr>
<td>Memcpy (DtoH)</td>
</tr>
<tr>
<td>Compute</td>
</tr>
</tbody>
</table>

float const | Step10_cuda_k | Step10_cuda_kernel | Step10_cuda KernelInt | Step10_cuda KernelInt |

100.0% Step10_cuda_k | Step10_cuda | Step10_cuda | Step10_cuda | Step10_cuda

Streams

Analysis

Stall Reasons

- Instruction Fetch
- Other
- Synchronization
- Texture
- Data Request
- Execution dependency

1. Performance-Critical Kernels
2. Compute, Bandwidth, or Latency Bound

- Perform Compute Analysis
- Perform Memory Bandwidth Analysis
- Return Analysis

Instructions and memory latency and memory bandwidth are likely the primary performance bottlenecks for this kernel. You may also want to perform these analyses if you modify the kernel or need to rerun your application to update this analysis.
Bottleneck Analysis

- Don’t assume an optimization was wrong
- Verify if it was wrong with the profiler

129 GB/s ➞ 84 GB/s

---

**Shared Memory Alignment and Access Pattern**

Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

**Optimization:** Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each access pattern of the memory access.

- **main.cu** - `/home/luitjens/code/CudaHandsOn/Example19`
  - Line / File: 49
  - Shared Load Transactions/Access = 16, Ideal Transactions/Access = 1 (2097152 transactions for 131072 total executions)
Performance Analysis

**gpuTranspose_kernel(int, int, float const *, float)**

<table>
<thead>
<tr>
<th>Start</th>
<th>770.067</th>
</tr>
</thead>
<tbody>
<tr>
<td>End</td>
<td>770.324</td>
</tr>
<tr>
<td>Duration</td>
<td>256.714</td>
</tr>
<tr>
<td>Grid Size</td>
<td>[64,64,1]</td>
</tr>
<tr>
<td>Block Size</td>
<td>[32,32,1]</td>
</tr>
<tr>
<td>Registers/Thread</td>
<td>10</td>
</tr>
<tr>
<td>Shared Memory/Block</td>
<td>4.125 KiB</td>
</tr>
<tr>
<td><strong>Function Unit (Arithmetic)</strong></td>
<td>[ ]</td>
</tr>
<tr>
<td><strong>Memory (Device)</strong></td>
<td>[ ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Load Efficiency</td>
</tr>
<tr>
<td>Global Store Efficiency</td>
</tr>
<tr>
<td>Shared Efficiency</td>
</tr>
<tr>
<td>Warp Execution Efficiency</td>
</tr>
<tr>
<td>Non-Predicated Warp Execution Efficiency</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Achieved</td>
</tr>
<tr>
<td>Theoretical</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shared Memory Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Memory Requested</td>
</tr>
<tr>
<td>Shared Memory Executed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L3 Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Loads</td>
</tr>
<tr>
<td>Local Stores</td>
</tr>
<tr>
<td>Shared Loads</td>
</tr>
<tr>
<td>Shared Stores</td>
</tr>
<tr>
<td>Global Loads</td>
</tr>
<tr>
<td>Global Stores</td>
</tr>
<tr>
<td>Atomic</td>
</tr>
<tr>
<td>L3 Shared Total</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L2 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Reads</td>
</tr>
<tr>
<td>L1 Writes</td>
</tr>
<tr>
<td>Texture Reads</td>
</tr>
<tr>
<td>Atomic</td>
</tr>
<tr>
<td>Noncoherent Reads</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Texture Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
</tr>
<tr>
<td>Device Memory</td>
</tr>
<tr>
<td>Reads</td>
</tr>
<tr>
<td>Writes</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

84 GB/s → 137 GB/s
The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.