Outline

Goals of this session

- Use cuda-memcheck to detect invalid memory accesses
- Use Nisght Eclipse Edition to debug a CUDA program
- Gain performance insight with NVIDIA Visual Profiler/nvprof

Contents

Debugging
  - cuda-memcheck
  - cuda-gdb
  - Nsight Eclipse Edition

Tasks

Profiling
  - nvprof
  - Visual Profiler
  - Others
  - Tasks
Debugging
cuda-memcheck

Command-line memory access analyzer

- Memory error detector; similar to Valgrind’s memcheck
- Has sub-tools, via cuda-memcheck --tool NAME:
  - memcheck: Memory access checking (default)
  - racecheck: Shared memory hazard checking
  - Also: synccheck, initcheck

- Remember to compile your program with debug information: add -g (host) or -G (device)

→ http://docs.nvidia.com/cuda/cuda-memcheck/
cuda-memcheck
Command-line memory access analyzer

- Memory error detector; similar to Valgrind’s memcheck
- Has sub-tools, via cuda-memcheck --tool NAME:
  - memcheck: Memory access checking (default)
  - racecheck: Shared memory hazard checking
  - Also: synccheck, initcheck
- Remember to compile your program:
  add -g (host) or -G (device)

→ http://docs.nvidia.com/cuda/cuda-memcheck/
Example

Start: `cuda-memcheck PROGRAM`

```
$ cuda-memcheck task1-cuda-memcheck
====== CUDA-MEMCHECK
Invalid __global__ write of size 4
    at 0x00000138 in /home/aherten/NVAL/Courses/CUDA-Course-2016-Aug/task1-cuda-memcheck.cu:20:set(int, float*, float)
    by thread (255,0,0) in block (0,0,0)
    Address 0x305ba22fc is out of bounds
Saved host backtrace up to driver entry point at kernel launch time
Host Frame:/usr/lib64/libcuda.so.0.1 (cuLaunchKernel + 0x2c5) [0x4745e5]
    Host Frame:task1-cuda-memcheck [0x172a1]
    Host Frame:task1-cuda-memcheck [0x31ab3]
    Host Frame:task1-cuda-memcheck [0x2da9]
    Host Frame:task1-cuda-memcheck [0x2b0a]
    Host Frame:task1-cuda-memcheck [0x2ce3]
    Host Frame:task1-cuda-memcheck [0x20a]
    Host Frame:/lib64/libc.so.6 (libc_start_main + 0xf5) [0x21b05]
    Host Frame:task1-cuda-memcheck [0x2859]
```

```
====== CUDA-MEMCHECK
Invalid __global__ write of size 4
    at 0x00000138 in /home/aherten/NVAL/Courses/CUDA-Course-2016-Aug/task1-cuda-memcheck.cu:20:set(int, float*, float)
    by thread (254,0,0) in block (0,0,0)
```
cuda-gdb

**Symbolic debugger**

- Powerful symbolic debugger for CUDA code
- Built on top of gdb
- Full usage: own course needed

**cuda-gdb 101**

- `run` Starts application, give arguments with `set args 1 2 ...`
- `break L` Create breakpoint
  
  \[L: \text{function name}, \text{line number LN, or FILE:LN}\]
- `continue` Continue running
- `print i` Print content of i
- `set variable i = 10` Set i to 10
- `info locals` Print all currently set variables
- `info cuda threads` Print current thread configuration
- `cuda thread N` Switch context to thread number N

→ `cheat sheet`

**cuda-gdb**

*Example*

**Start:** `cuda-gdb app` → `run`

Set breakpoint with `break func` or `break L` or `break file.c:L`
Nsight Eclipse Edition

The CUDA IDE

- Full-fledged IDE for CUDA development; based on Eclipse
  - Source code editor with CUDA C / C++ highlighting
  - Project / file management with integration of version control
  - Build system
  - Remote invocation capabilities
  - Graphical interface for debugging heterogeneous applications (cuda-gdb under the hood)
  - Integrated NVIDIA Visual Profiler

- Also: Nsight Visual Studio Edition (only Windows)

```c
#include <stdio.h>

#define CUDA_CHECK_RETURN(value) {
    cudaError_t _m_cudaStat = value;
    if (_m_cudaStat != cudaSuccess) {
        fprintf(stderr, "Error %s at line %d in file %s\n",
                cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__);
        exit(1);
    }
}

__global__ void set(const int n, float* __restrict__ const a_d, const float value) {
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    if (i < n) {
        a_d[i + 2*n] = value;
    }
}

int main() {
    int n = 1024;
    CUDA_CHECK_RETURN(cudaSetDevice(0));
    float* a_d = NULL;
    CUDA_CHECK_RETURN(cudaMalloc(void**, &a_d, n * sizeof(float)));
    float value = 3.14f;
    set<<<n/256, 256>>>(n, a_d, value);
    CUDA_CHECK_RETURN(cudaGetLastError());
}
```
Debug CUDA Program with Nsight EE

Setup

Start nsight
Debug CUDA Program with Nsight EE

Setup

Configure debugging
Debug CUDA Program with Nsight EE

Setup

Choose C/C++ Application
Debug CUDA Program with Nsight EE

Setup

Create *New launch configuration*
Debug CUDA Program with Nsight EE

Setup

Insert path to executable
Debug CUDA Program with Nsight EE

Setup

Insert path to executable
Debug CUDA Program with Nsight EE

Setup

Click Debug
Debug CUDA Program with Nsight EE

Usage
Debug CUDA Program with Nsight EE

Usage
Debug CUDA Program with Nsight EE

Usage
Debug CUDA Program with Nsight EE

Usage

![Diagram of CUDA Debugger interface showing variables and CUDA information.](image-url)
Task 1

*Use cuda-memcheck to identify error*

- **Location of code:** CUDATools/exercises/tasks/task1
- **Steps (see also Instructions.md)**
  - **Build:**
    - `make`
  - **Run:**
    - `make run`
  - **Fix!**
    - Look into `task1-cuda-memcheck.cu` and fix error; `cuda-memcheck` should run without errors!
Task 2

*Debug with Nsight Eclipse Edition/cuda-gdb*

- **Location of code:** CUDATools/exercises/tasks/task2
- **Steps (see also Instructions.md)**
  - Build program:
    - `make`
  - Start Nsight Eclipse Edition:
    - `nsight`
  - Setup debug session:
    - See above
  - Let thread 4 from first block print 42 (instead of 0)
    - Do not change the source code! Use the variable view.
  - **Alternative:** Use `cuda-gdb` for this
Profiling
Motivation for Measuring Performance

- **Improvement** possible only if program is measured
  
  *Don’t trust your gut!*

- **Identify:**
  
  - **Hotspots** Which functions take most of the time?
  - **Bottlenecks** What are the limiters of performance?

- Manual timing possible, but tedious and error-prone
  
  Feasible for small applications, impractical for complex ones

→ **Profiler**

- In-detail insights
- No code changes needed!
- Easy access to hardware counters (*PAPI, CUPTI*)
**nvprof**

*Command-line GPU profiler*

- Profiles CUDA kernels and API calls; also CPU code!
- Basic default profiling data, much more available with:
  - `--events E1,E2`: Measure specific events
    List available events via `--query-events`
  - `--metrics M1,M2`: Measure combined metrics
    List available metrics via `--query-metrics`
- Further useful options
  - `--export-profile`: Generate profiling data for Visual Profiler
    See later slide
  - `--print-gpu-trace`: Show trace of function calls
  - `--unified-memory-profiling per-process-device`: Print unified memory profiling information
    CUDA 7.5: Prevent zero-copy fallback: `CUDA_MANAGED_FORCE_DEVICE_ALLOC=1`
  - `--help`: For all the rest...

nvprof

Example I

Start: nvprof PROGRAM

```
abherten@jrl11:~/NVAL/Courses/CUDA-Course-Aug-2016$ srun nvprof ./task3-scale_vector_um
==32741== NVPROF is profiling process 32741, command: ./task3-scale_vector_um
==32741== Profiling application: ./task3-scale_vector_um
==32741== Profiling result:
Time(%)  Time   Calls  Avg    Min     Max    Name
100.00%  4.096us 1  4.096us  4.096us  4.096us  scale(float, float*, float*, int)

==32741== API calls:
Time(%)  Time   Calls  Avg     Min     Max     Name
99.15%   215.4ms  2  107.7ms  44.070ms 215.37ms  cudaMallocManaged
50.58%   1.269ms  166 7.6470us 100ns  298.86us  cuDeviceGetAttribute
0.09%    204.3us  2  102.19us 20.279us  184.11us  cudaFree
0.06%    138.6us  2  69.327us  69.133us  69.521us  cuDeviceTotalMem
0.06%    124.7us  2  62.365us  55.657us  69.074us  cuDeviceGetName
0.03%    74.989us 1  74.989us  74.989us  74.989us  cudaLaunch
0.01%    17.613us 1  17.613us  17.613us  17.613us  cudaDeviceSynchronize
0.00%    10.225us 1  10.225us  10.225us  10.225us  cudaSetDevice
0.00%    10.041us 4  2.5100us  143ns  8.9340us  cudaSetupArgument
0.00%    1.7620us 2  881ns  335ns  1.4070us  cuDeviceGetCount
0.00%    1.7040us 1  1.7040us  1.7040us  1.7040us  cudaConfigureCall
0.00%     935ns  4  233ns  116ns  469ns  cuDeviceGet
Passed!
```
nvprof --metrics inst_execu[...] --cpu-profiling on PROGRAM

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Metric Description</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst_executed</td>
<td>Instructions Executed</td>
<td>1088</td>
<td>1088</td>
<td>1088</td>
</tr>
<tr>
<td>inst_issued</td>
<td>Instructions Issued</td>
<td>1576</td>
<td>1576</td>
<td>1576</td>
</tr>
<tr>
<td>issued_ipc</td>
<td>Issued IPC</td>
<td>0.094433</td>
<td>0.094433</td>
<td>0.094433</td>
</tr>
<tr>
<td>flop_count_sp</td>
<td>Floating Point Operations (Single Precision)</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>flop_count_dp</td>
<td>Floating Point Operations (Double Precision)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Passed!

CPU profiling result (bottom up):
- 38.89% cudbgGetAPIVersion
- 38.89% start_thread
- 38.89% clone
- 30.56% cuDevicePrimaryCtxRetain
- 30.56% cuda::contextStateManager::initPrimaryContext(cudart::device*)
Visual Profiler
The insight provider

- Timeline view of all things GPU (API calls, kernels, memory)
- View launch and run configurations
- Guided and unguided analysis, with (among others):
  - Performance limiters
  - Kernel and execution properties
  - Memory access patterns
- NVIDIA Tools Extension NVTX (for annotation)

→ https://developer.nvidia.com/nvidia-visual-profiler
Visual Profiler

Example

**Start:** nvvp → *File* → *New Session*
Visual Profiler and nvprof

Interoperability

- `nvprof` can produce the input for Visual Profiler, options:
  - `-o f1` Write profile to file f1
  - `--analysis-metrics -o f2` Measure metrics needed for Visual Profiler’s guided analysis, write to file f2

→ Import to Visual Profiler (`nvvp → File ↔ Import…` or Ctrl+I)
Visual Profiler and nvprof

Interoperability

nvprof can produce the input for Visual Profiler, options:
- `-o file` Write profile to file
- `--analysis-metrics -o file` Measure metrics needed for Visual Profiler's guided analysis, write to file

Import to Visual Profiler (`nvvp`!
`File`, `Import…` or Ctrl+I)

Andreas Herten | CUDA Tools | 24 April 2017
Other Profilers

Because there’s so much more

- Special measurement registers (performance counters) of GPU exposed to third-party applications via **CUPTI** (CUDA Profiling Tools Interface)

→ Enables professional profiling tools for GPU!

**PAPI** API for measuring performance counters, also GPU
For example: `cuda::device:0:threads_launched`

**Score-P** Measures CPU and GPU profile of program
Prefix `nvcc` compilation with `scorep`, set `SCOREP_CUDA_ENABLE=yes`, run

**Cube** Displays performance report from Score-P concisely

**Vampir** Display report form Score-P in timeline view, also multiple MPI ranks
Score-P
Analysis with Cube

Andreas Herten | CUDA Tools | 24 April 2017
Task 3

Analyze and profile `scale_vector_um`

- **Location of code:** CUDATools/exercises/tasks/task3/
- **See Instructions.md**
- **Do any (all?) of the following:**
  - **A** Use `nvprof` to gather profile, Visual Profiler for viewing
    - Use `nvprof` to write `scale_vector_um`’s timeline to file
    - Start Visual Profiler (`nvvp`) on JURON’s login node; import timeline
    - Use `nvprof` to add metric information to timeline
    - Import, run guided analysis in Visual Profiler
  - **B** Use Visual Profiler for everything
    - Start an interactive session on JURECA
    - Launch Visual Profiler
    - Start, profile, and run guided analysis in Visual Profiler

- **Objective:** Get to know the tools
- **Also:** What’s the runtime of the kernel?

Andreas Herten | CUDA Tools | 24 April 2017
Conclusions

What we’ve learned

- **Debugging**
  - Detect false memory accesses with **cuda-memcheck**
  - Debug from console with **cuda-gdb**
  - Debug with GUI in **Nsight Eclipse Edition**

- **Profiling**
  - Use **Visual Profiler** for analysis and optimization
  - **nvprof** in console, also for batch jobs

Thank you for your attention!

a.herten@fz-juelich.de
Appendix

Glossary
Glossary I

**API**  A programmatic interface to software by well-defined functions. Short for application programming interface. 38

**ATI**  Canada-based GPUs manufacturing company; bought by AMD in 2006. 38

**CUDA**  Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 7, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 26, 38

**GCC**  The GNU Compiler Collection, the collection of open source compilers, among other for C and Fortran. 38
Glossary II

**LLVM**  An open Source compiler infrastructure, providing, among others, Clang for C. 38

**NVIDIA**  US technology company creating GPUs. 9, 29, 38

**NVLink**  NVIDIA’s communication protocol connecting CPU ↔ GPU and GPU ↔ GPU with 80 GB/s. PCI-Express: 16 GB/s. 38

**OpenACC**  Directive-based programming, primarily for many-core machines. 38

**OpenCL**  The Open Computing Language. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 38
Glossary III

**OpenGL**  The *Open Graphics Library*, an API for rendering graphics across different hardware architectures. 38

**OpenMP**  Directive-based programming, primarily for multi-threaded machines. 38

**P100**  A large GPU with the Pascal architecture from NVIDIA. It employs NVLink as its interconnect and has fast HBM2 memory. 38

**SAXPY**  Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 38

**Tesla**  The GPU product line for general purpose computing of NVIDIA. 38
Thrust  A parallel algorithms library for (among others) GPUs. See https://thrust.github.io/.