CUDA TOOLS
GSP GPU COURSE 2018

8 August 2018 | Andreas Herten | Forschungszentrum Jülich

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

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

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Debugging

`cuda-memcheck`

Command-line memory access analyzer

- Memory error detector; similar to Valgrind's `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`

`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)

`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)

Compile options for nvcc

`-g`	Debug info for **host** code
`-G`	Debug info for **device** code
`-lineinfo`	Line number for device code

`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)

Compile options for nvcc

`-g`	Debug info for **host** code *ok*
`-G`	Debug info for **device** code *slow*
`-lineinfo`	Line number for device code *ok*

`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/`

Compile options for nvcc

`-g`	Debug info for **host** code *ok*
`-G`	Debug info for **device** code *slow*
`-lineinfo`	Line number for device code *ok*

JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

Example

Launch: `cuda-memcheck` *`PROGRAM`*

Example

Launch: `cuda-memcheck` *`PROGRAM`*

```
aherten@juronc04:~/NVAL/Courses/CUDA-Course-Apr-2017/Tools/1-Cuda-Memcheck$ cuda-memcheck ./set_vector
========= CUDA-MEMCHECK
========= Invalid __global__ write of size 4
=========     at 0x00000218 in
  /gpfs/homeb/zam/aherten/NVAL/Courses/CUDA-Course-Apr-2017/Tools/1-Cuda-Memcheck/set_vector.cu:20:set(int,
  float*, float)
=========     by thread (127,0,0) in block (3,0,0)
=========     Address 0x110013e02dfc is out of bounds
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame:/usr/lib64/nvidia/libcuda.so.1 (cuLaunchKernel + 0x24c) [0x281b1c]
=========     Host Frame:./set_vector [0xfd8c]
=========     Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xc4) [0x24b74]
=========
========= Invalid __global__ write of size 4
=========     at 0x00000218 in
  /gpfs/homeb/zam/aherten/NVAL/Courses/CUDA-Course-Apr-2017/Tools/1-Cuda-Memcheck/set_vector.cu:20:set(int,
  float*, float)
=========     by thread (126,0,0) in block (3,0,0)
```

cuda-gdb

Symbolic debugger

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

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: function name, line 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

Symbolic debugger

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

→ http://docs.nvidia.com/cuda/cuda-gdb/

cuda-gdb 101

`run`	Starts application, give arguments with `set args 1 2 …`
`break` *L*	Create breakpoint *L: function name, line 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

Launch: `cuda-gdb` *`app`* $\rightarrow$ `run`

Set breakpoint with `break` *`func`* or `break` *`L`* or `break` *`file.c:L`*

```
aherten@juronc04:~/NVAL/Courses/CUDA-Course-Apr-2018/Tools/2-Cuda-Gdb$ cuda-gdb gpu_print
NVIDIA (R) CUDA Debugger, 9.1 release
Portions Copyright (C) 2007-2017 NVIDIA Corporation
GNU gdb (GDB) 7.12
For help, type "help".
Reading symbols from gpu_print...done.
(cuda-gdb) run
Starting program: /gpfs/homeb/zam/aherten/NVAL/Courses/CUDA-Course-Apr-2018/Tools/2-Cuda-Gdb/gpu_print
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x10000755f190 (LWP 155595)]
blockIdx.x = 0, threadIdx.x = 0, i = 0
blockIdx.x = 0, threadIdx.x = 1, i = 0
blockIdx.x = 0, threadIdx.x = 2, i = 0
```

Nsight Eclipse Edition

The CUDA IDE

- Full-fledged IDE for CUDA development; based on Eclipse

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

Nsight Eclipse Edition

The CUDA IDE

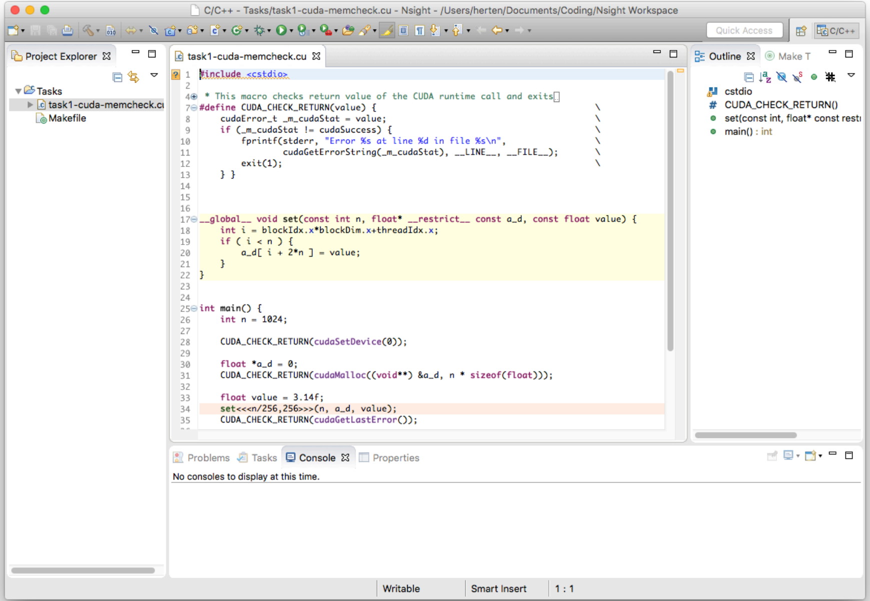
- Full-fledged IDE for CUDA development; based on Eclipse
  - Source code editor with CUDA C / C++ highlighting
  - Project / file management with integration of version control
  - Build system
  - Remote invocation capabilities
  - Graphical interface for debugging heterogeneous applications (`cuda-gdb`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

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*)

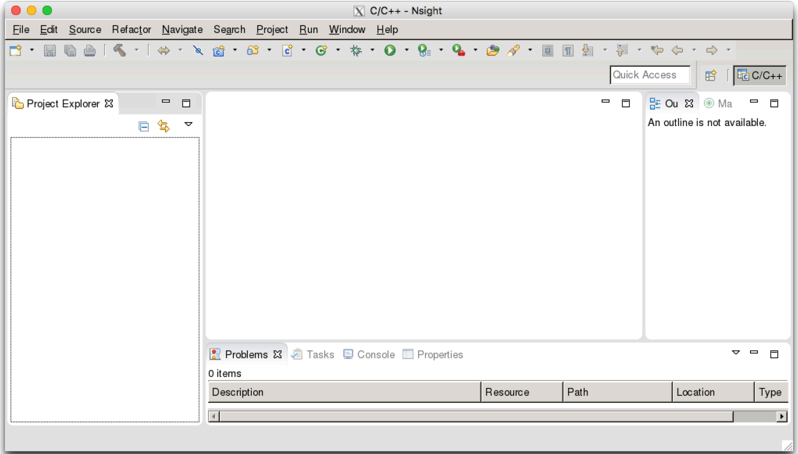
→ `https://developer.nvidia.com/nsight-eclipse-edition/`


C/C++ - Tasks/task1-cuda-memcheck.cu - Nsight - /Users/herten/Documents/Coding/Nsight Workspace
Quick Access
C/C++
Project Explorer
Tasks
task1-cuda-memcheck.cu
Makefile
task1-cuda-memcheck.cu
#include <cstdio>
 * This macro checks return value of the CUDA runtime call and exits
#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 = 0;
    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());
Outline
Make T
cstdio
CUDA_CHECK_RETURN()
set(const int, float* const restr
main() : int
Problems
Tasks
Console
Properties
No consoles to display at this time.
Writable
Smart Insert
1 : 1

Debug CUDA Program with Nsight EE

Setup

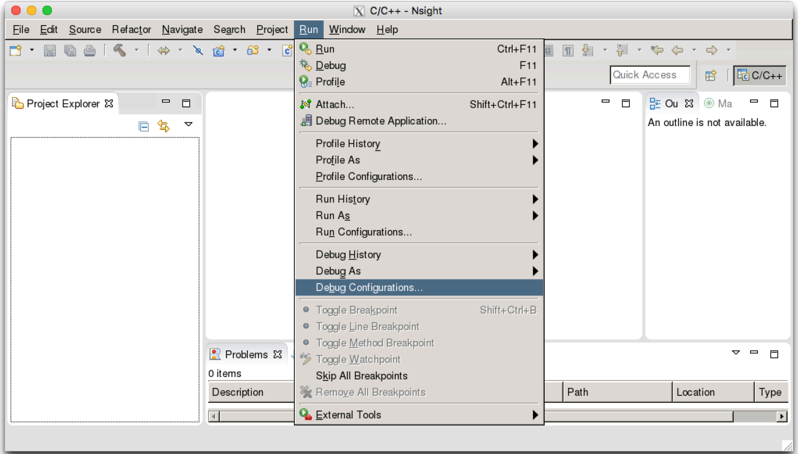
Start `nsight`



Debug CUDA Program with Nsight EE

Setup

Configure debugging

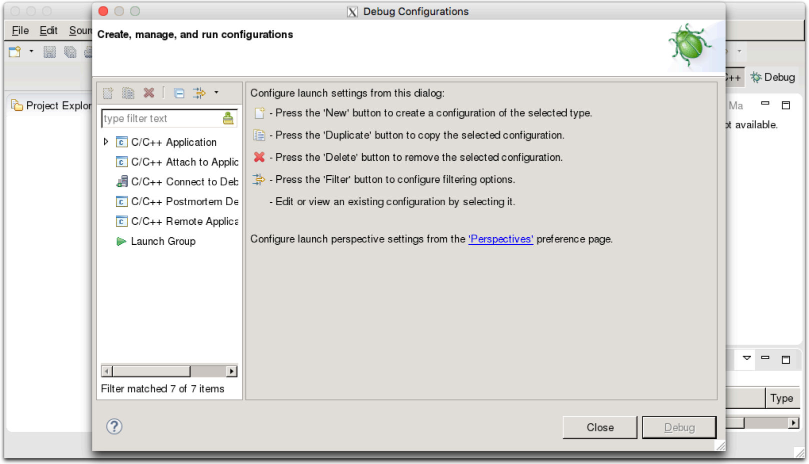


JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

Debug CUDA Program with Nsight EE

Setup

Choose *C/C++ Application*

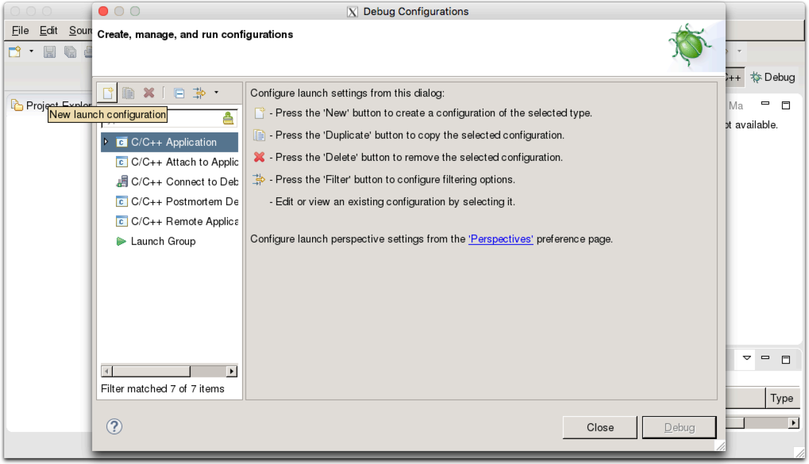


JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Debug CUDA Program with Nsight EE

Setup

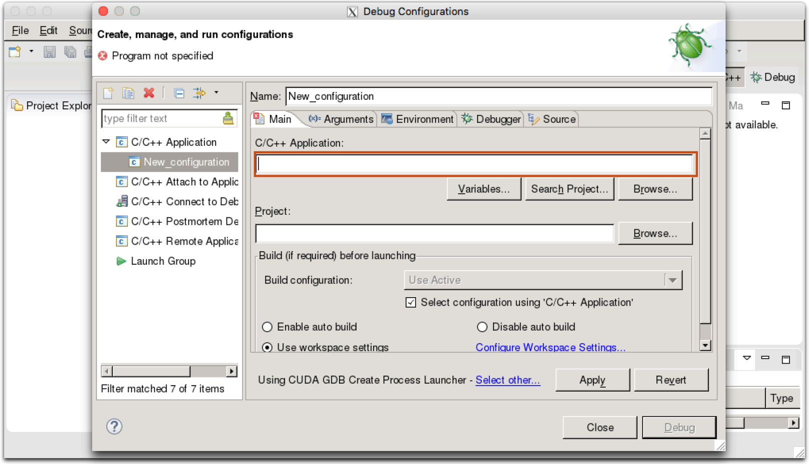
Create *New launch configuration*



Debug CUDA Program with Nsight EE

Setup

Insert path to executable

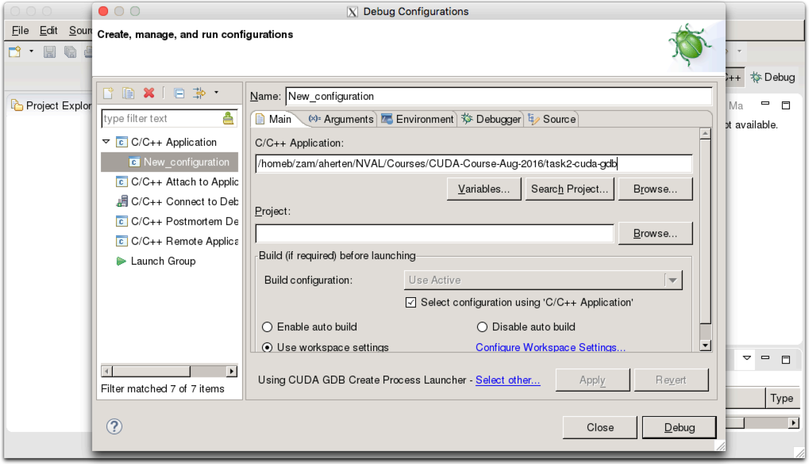


JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Debug CUDA Program with Nsight EE

Setup

Insert path to executable

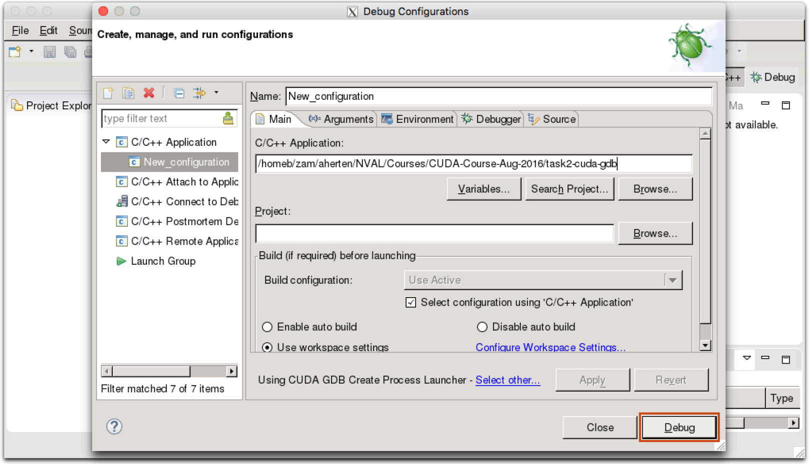


JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Debug CUDA Program with Nsight EE

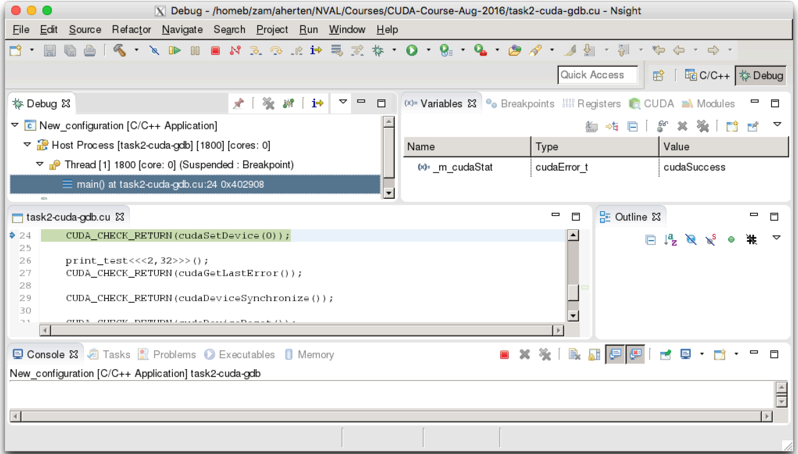
Setup

Click *Debug*



Debug CUDA Program with Nsight EE

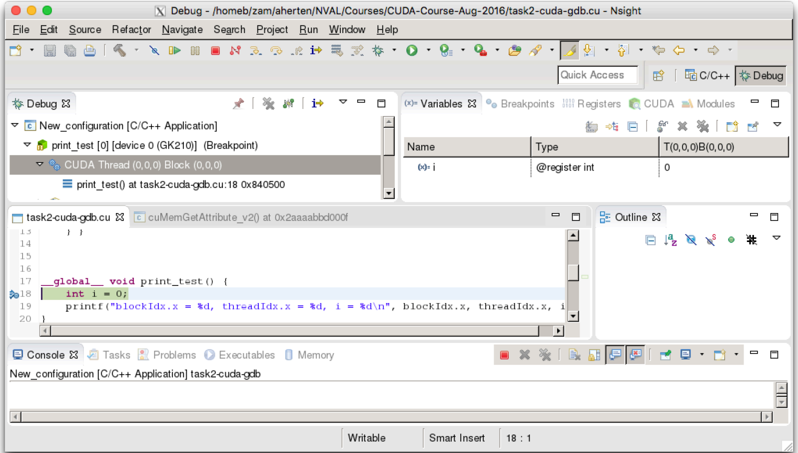
Usage



JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

Debug CUDA Program with Nsight EE

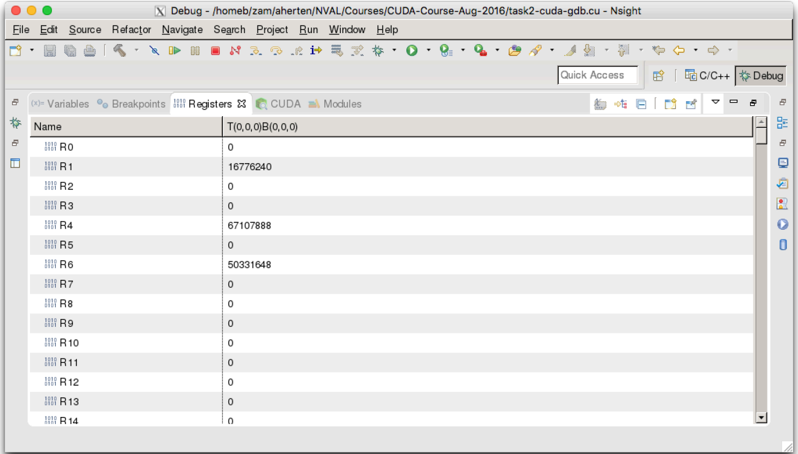
Usage



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

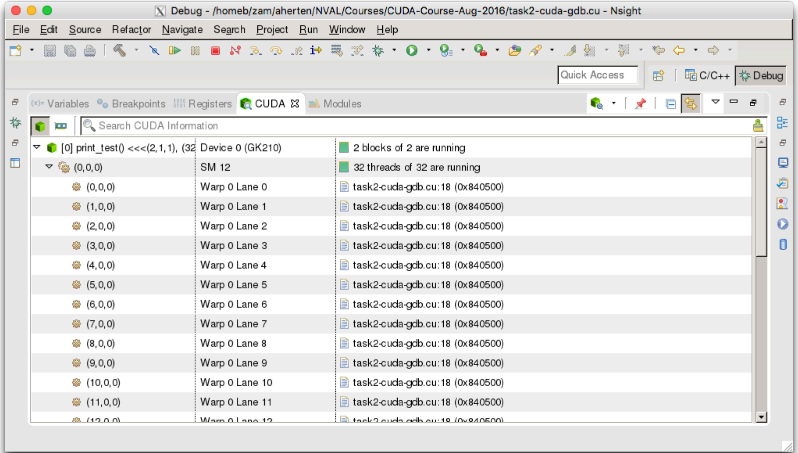
Debug CUDA Program with Nsight EE

Usage



Debug CUDA Program with Nsight EE

Usage



Task 1

TASK 1

Use `cuda-memcheck` to identify error

- Location of code: `2-CUDA-Tools/exercises/tasks/task1`
- Steps (see also `Instructions.md`)
  - Build:
    `make`
  - Run:
    `make run`
  - Fix!
    Use cuda-memcheck to fix error in `task1-cuda-memcheck.cu`; `cuda-memcheck` should run without errors!

Task 2

TASK 2

Debug with Nsight Eclipse Edition/cuda-gdb

- Location of code: `2-CUDA-Tools/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` instead of Nsight EE

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!

`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`

`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
  - `--print-gpu-trace`: Show trace of function calls
  - `--unified-memory-profiling per-process-device`: Print unified memory profiling information
  - `--help`: For all the rest…

`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
  - `--print-gpu-trace`: Show trace of function calls
  - `--unified-memory-profiling per-process-device`: Print unified memory profiling information
  - `--help`: For all the rest…

→ `http://docs.nvidia.com/cuda/profiler-users-guide/`

`nvprof`

Example I

Launch: `nvprof` *`PROGRAM`*

nvprof

Example I

Launch: `nvprof PROGRAM`

```
aherten@juronc04:~/NVAL/Courses/CUDA-Course-Apr-2017/Tools/3-Scale-Vector$ nvprof ./scale_vector_um
==155639== NVPROF is profiling process 155639, command: ./scale_vector_um
==155639== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  271.05us         1  271.05us  271.05us  271.05us  scale(float, float*, float*,
      API calls:   97.43%  180.19ms         2  90.093ms  8.8760us  180.18ms  cudaMallocManaged
                    1.36%  2.5067ms       360  6.9630us     256ns  280.85us  cuDeviceGetAttribute
                    0.83%  1.5355ms         4  383.87us  376.56us  404.13us  cuDeviceTotalMem
                    0.15%  274.34us         1  274.34us  274.34us  274.34us  cudaDeviceSynchronize

==155639== Unified Memory profiling result:
Device "Tesla P100-SXM2-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       9  113.78KB  64.000KB  256.00KB  1.000000MB  44.00000us  Host To Device
      13  157.54KB  64.000KB  896.00KB  2.000000MB  96.73600us  Device To Host
```

nvprof

Example II

Launch: `nvprof --metrics inst_execu[…] --cpu-profiling on` *`PROGRAM`*

```
aherten@juronc04:~/NVAL/Courses/CUDA-Course-Apr-2017/Tools/3-Scale-Vector$ nvprof --metrics
 inst_executed,inst_issued,issued_ipc,flop_coun_sp,flop_count_dp --cpu-profiling on ./scale_vector_um
==155720== NVPROF is profiling process 155720, command: ./scale_vector_um
==155720== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Replaying kernel "scale(float, float*, float*, int)" (done)
==155720== Profiling application: ./scale_vector_um
==155720== Metric result:
Invocations                     Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla P100-SXM2-16GB (0)"
    Kernel: scale(float, float*, float*, int)
          1                   inst_executed                     Instructions Executed      134472      134472      134472
          1                     inst_issued                       Instructions Issued      138422      138422      138422
          1                      issued_ipc                                Issued IPC    0.952913    0.952913    0.952913

======== CPU profiling result (bottom up):
Time(%)      Time  Name
 69.09%  771.01ms  ---
 66.36%  740.57ms  | start_thread
 66.36%  740.57ms  | | clone
```

Visual Profiler

The insight provider

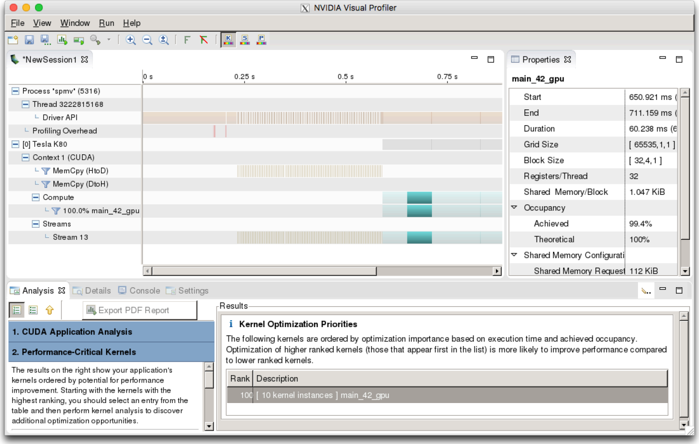
- Timeline view of all things GPU (API calls, kernels, memory)
- View launch and run configurations
- Guided and unguided analysis, with (among others):
  - Performance limiters
  - Kernel and execution properties
  - Memory access patterns
- NVIDIA Tools Extension NVTX (for annotation)

→ `https://developer.nvidia.com/nvidia-visual-profiler`

Visual Profiler

Example

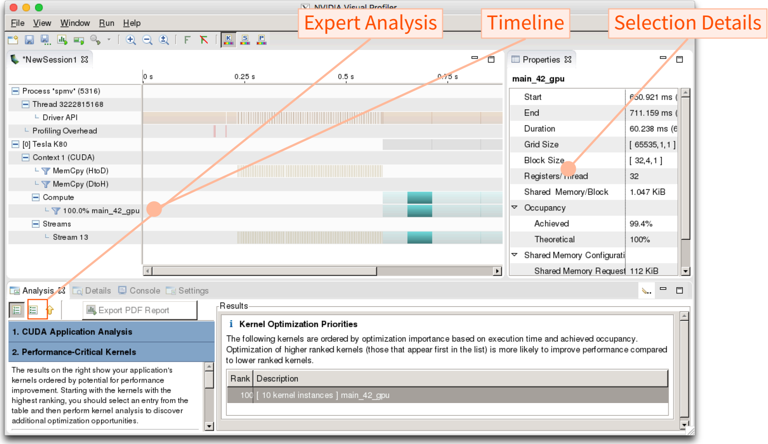
Launch: nvvp → *File* ↪ *New Session*



Visual Profiler

Example

Launch: 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)

Visua

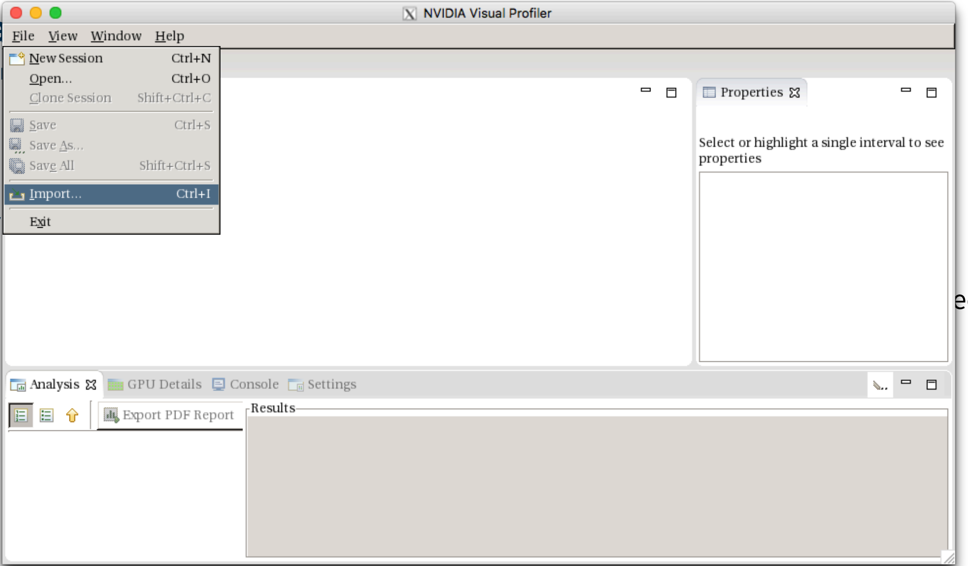
Interope

- nv

  – –

  ed

→ Im



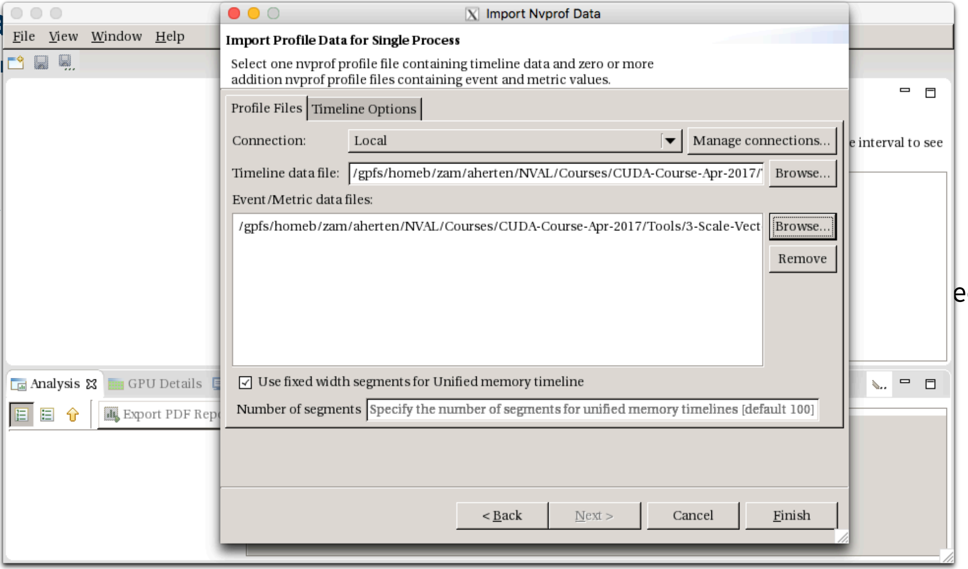
Visua

Interope

- nv

  -- ed

→ Im



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 Interace*)

→ Enables professional profiling tools for GPU!

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 Interace*)

→ Enables professional profiling tools for GPU!

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

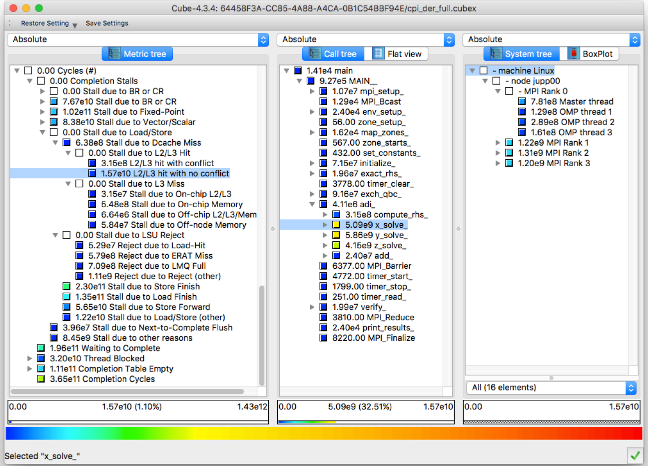
Score-P: Measures CPU and GPU profile of program
Prefix `nvcc` compilation with `scorep`, set `SCOREP_CUDA_ENABLE=yes`, run

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



Actually, no GPU information displayed here…

JÜLICH Forschungszentrum

JÜLICH SUPERCOMPUTING CENTRE

Task 3

TASK 3

Analyze and profile `scale_vector_um`

- Location of code: `2-CUDA-Tools/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 JURECA (JUWELS?); 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 JUWELS: `srun [...] --forward-x`~~
    - ~~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?

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

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

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

Appendix

Glossary

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Glossary I

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 12, 13, 14, 16, 17, 18, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 36, 37, 38, 39

JSC Jülich Supercomputing Centre, the supercomputing institute of Forschungszentrum Jülich, Germany. 57

JURECA A multi-purpose supercomputer with 1800 nodes at JSC. 52

JUWELS Jülich's new supercomputer, the successor of JUQUEEN. 52

NVIDIA US technology company creating GPUs. 16, 17, 18, 19, 43, 57

CPU Central Processing Unit. 49, 50

GPU Graphics Processing Unit. 43, 49, 50, 51, 57

References: Images, Graphics I

[1] Martin Oslic. *Bug*. Freely available at Unsplash. URL: `https://unsplash.com/photos/Qi93Pl5vDRw`.

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE