JÜLICH
FORSCHUNGSZENTRUM

CUDA Tools for Debugging and Profiling

CUDA Course, Jülich Supercomputing Centre

Andreas Herten, Forschungszentrum Jülich, 3 August 2016

Overview

JÜLICH FORSCHUNGSZENTRUM

What you will learn in this session

- Use `cuda-memcheck` to detect invalid memory accesses
- Use **Nisght Eclipse Edition** to debug a CUDA program
- Use the **NVIDIA Visual Profiler**, use `nvprof`

Contents

JÜLICH
FORSCHUNGSZENTRUM

Debugging

cuda-memcheck

Command-line memory access analyzer

JÜLICH FORSCHUNGSZENTRUM

- 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/`

Example

JÜLICH FORSCHUNGSZENTRUM

Start via `cuda-memcheck` *PROGRAM*

Tasks — herten@zam231 — ..ug-2016/Tasks — -zsh

```
$ 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(i
nt, float*, float)
=========     by thread (255,0,0) in block (0,0,0)
=========     Address 0x1305ba23fc is out of bounds
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame:/usr/lib64/libcuda.so.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 [0x2cb1]
=========     Host Frame:task1-cuda-memcheck [0x2ce3]
=========     Host Frame:task1-cuda-memcheck [0x2a9d]
=========     Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xf5) [0x21b05]
=========     Host Frame:task1-cuda-memcheck [0x2859]
=========
========= Invalid __global__ write of size 4
=========     at 0x00000138 in /home/aherten/NVAL/Courses/CUDA-Course-2016-Aug/task1-cuda-memcheck.cu:20:set(i
nt, float*, float)
=========     by thread (254,0,0) in block (0,0,0)
```

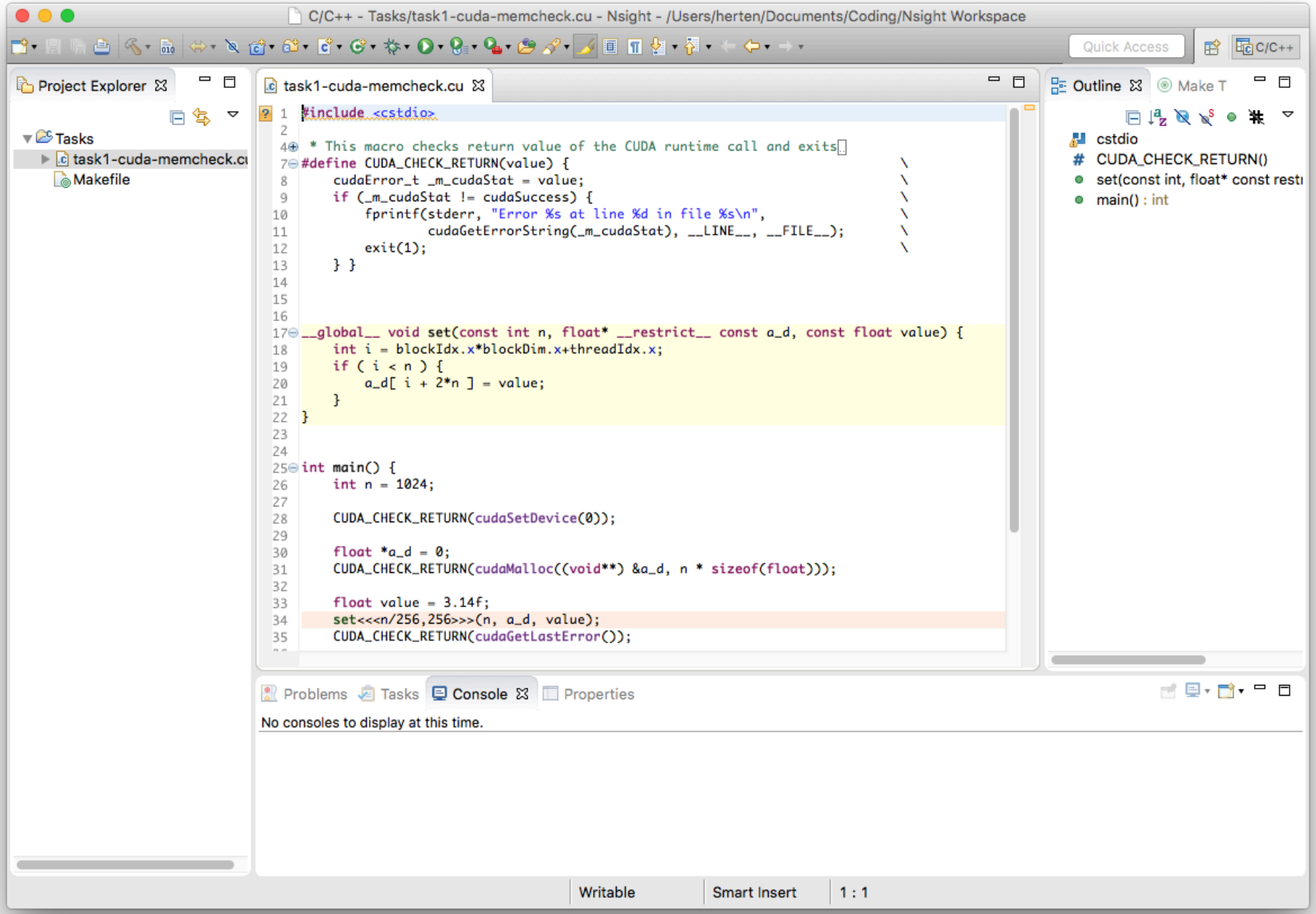
Nsight Eclipse Edition

The CUDA IDE

JÜLICH FORSCHUNGSZENTRUM

- 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 capabilities
  - Graphical interface for debugging heterogeneous applications (command line utility: `cuda-gdb`)
  - Integrated NVIDIA Visual Profiler
- Also: Nsight Visual Studio Edition (*only Windows*)

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

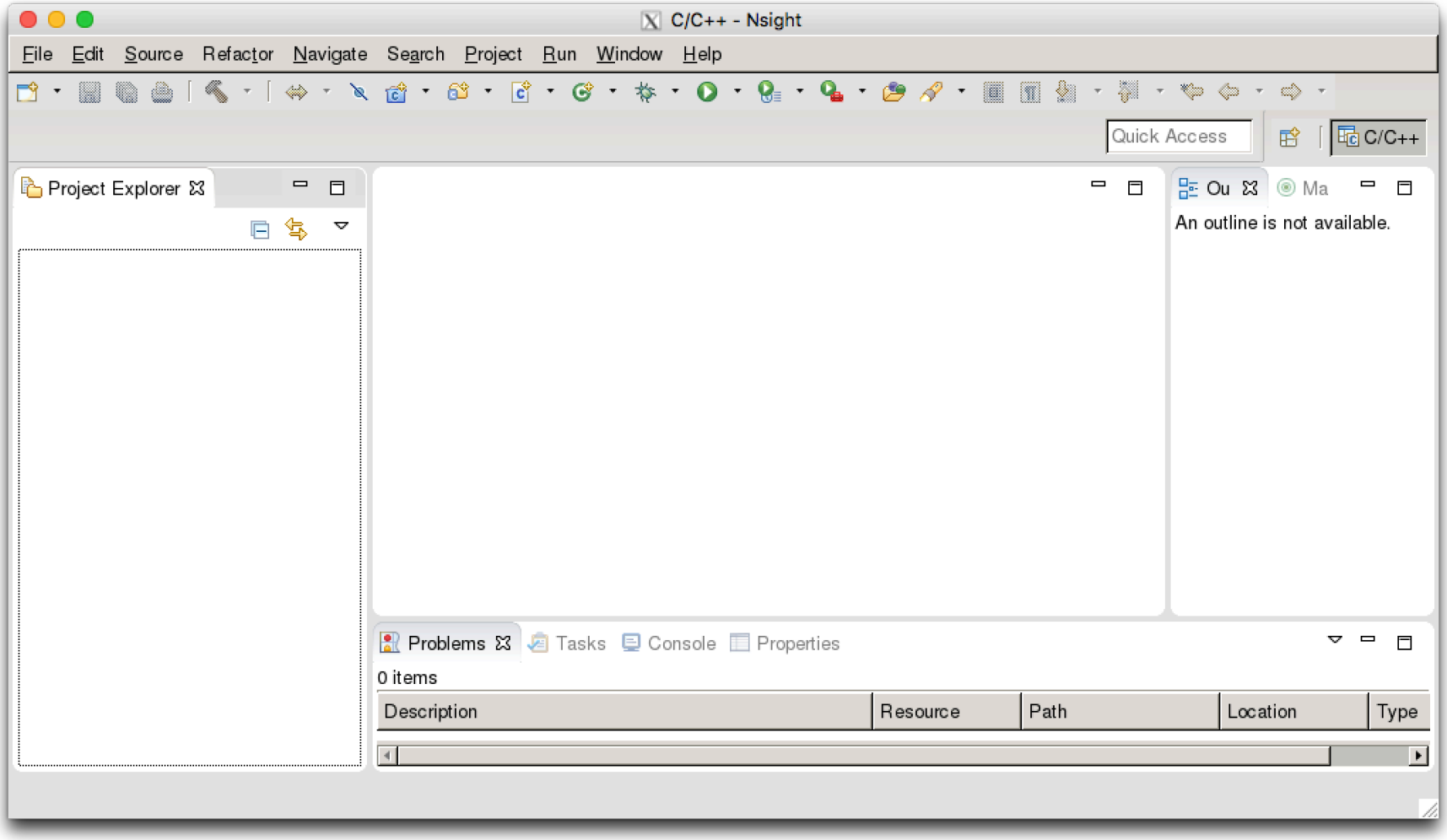

C/C++ - Tasks/task1-cuda-memcheck.cu - Nsight - /Users/herten/Documents/Coding/Nsight Workspace
Quick Access
C/C++
Project Explorer
Tasks
task1-cuda-memcheck.cu
Makefile
task1-cuda-memcheck.cu
#include <cstdio>
* This macro checks return value of the CUDA runtime call and exits
#define CUDA_CHECK_RETURN(value) {
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 resti
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

JÜLICH FORSCHUNGSZENTRUM

Start `nsight`

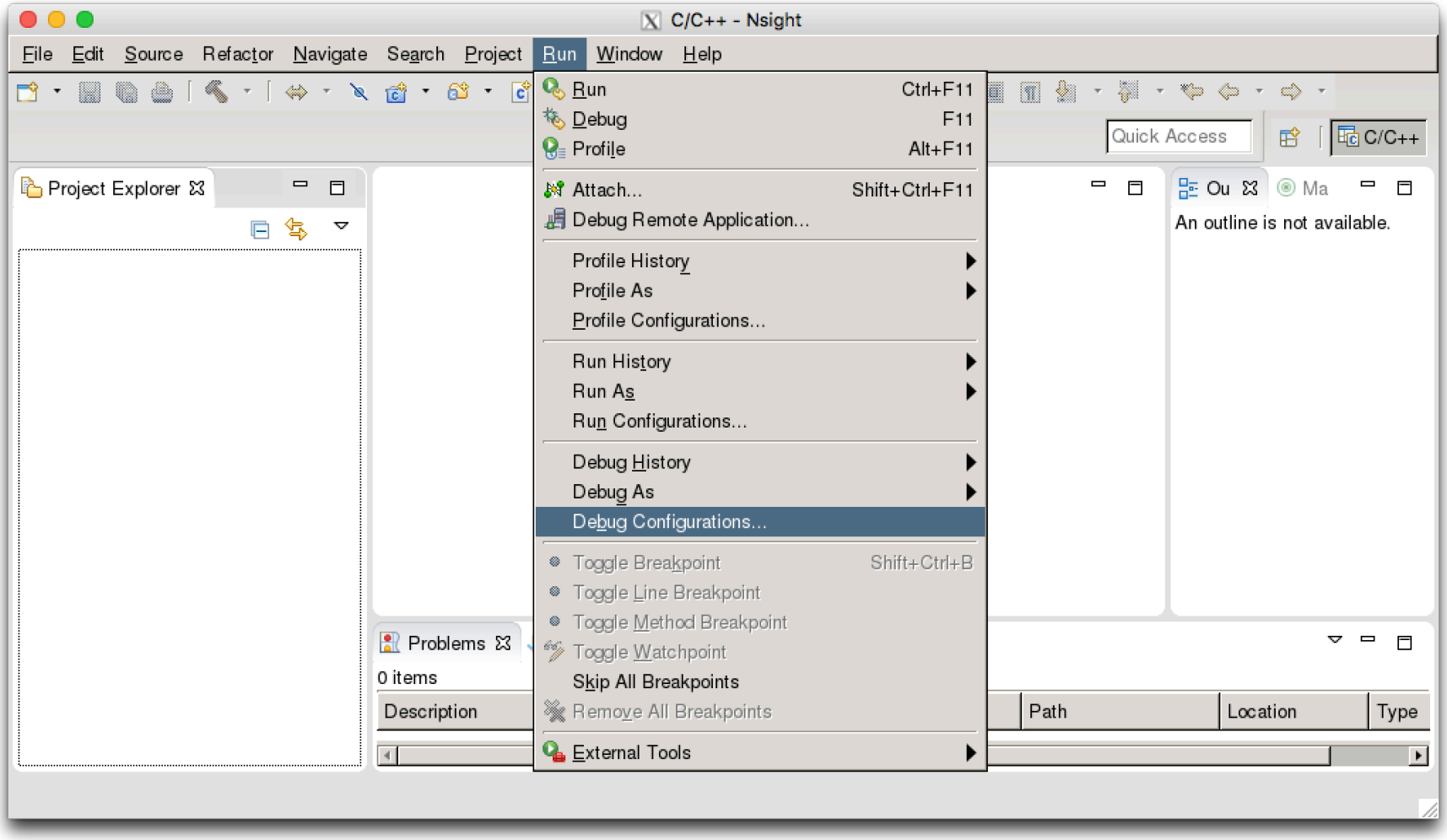


Debug CUDA Program with Nsight EE

Setup

JÜLICH FORSCHUNGSZENTRUM

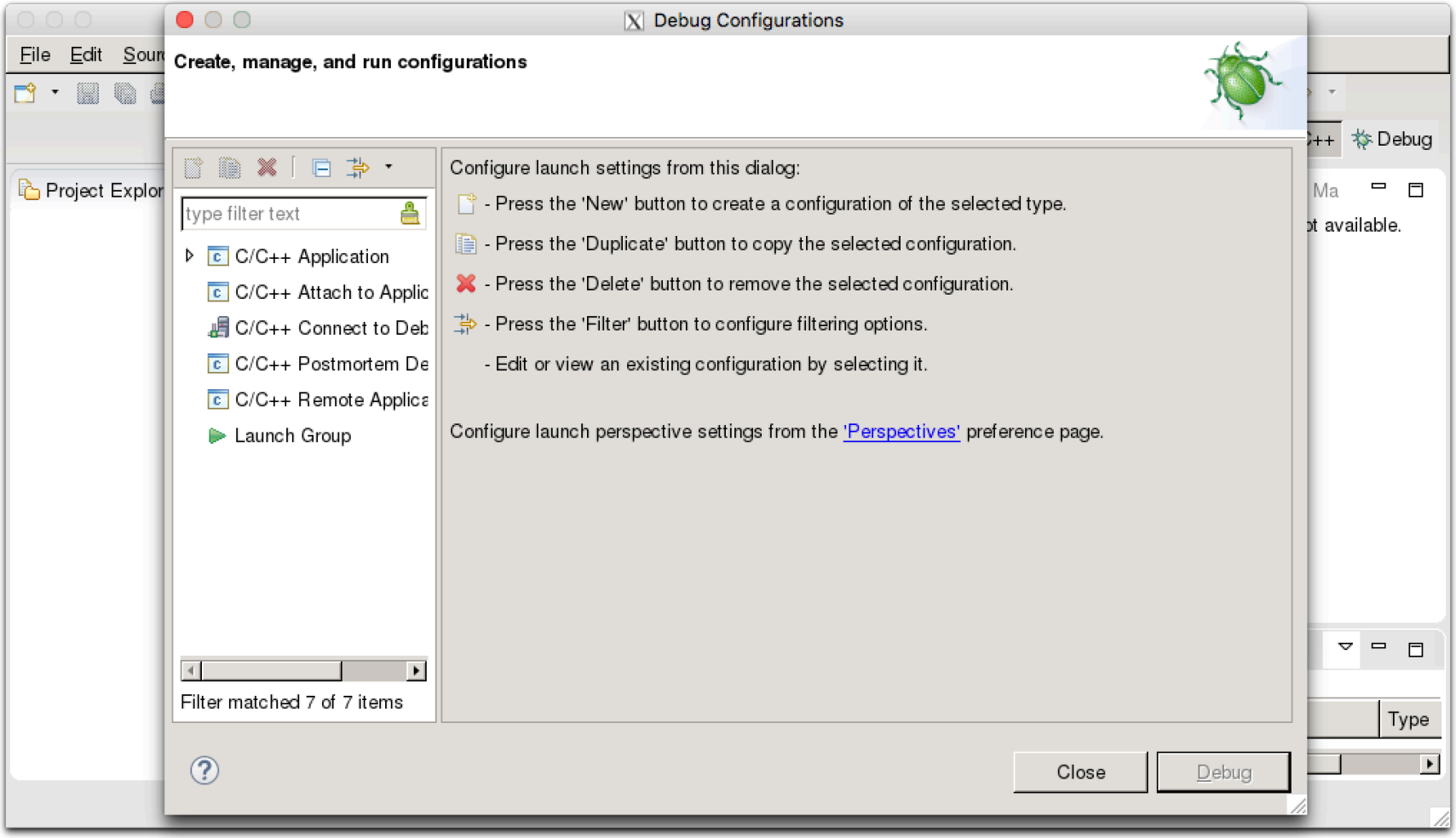
Configure debugging



Debug CUDA Program with Nsight EE

Setup

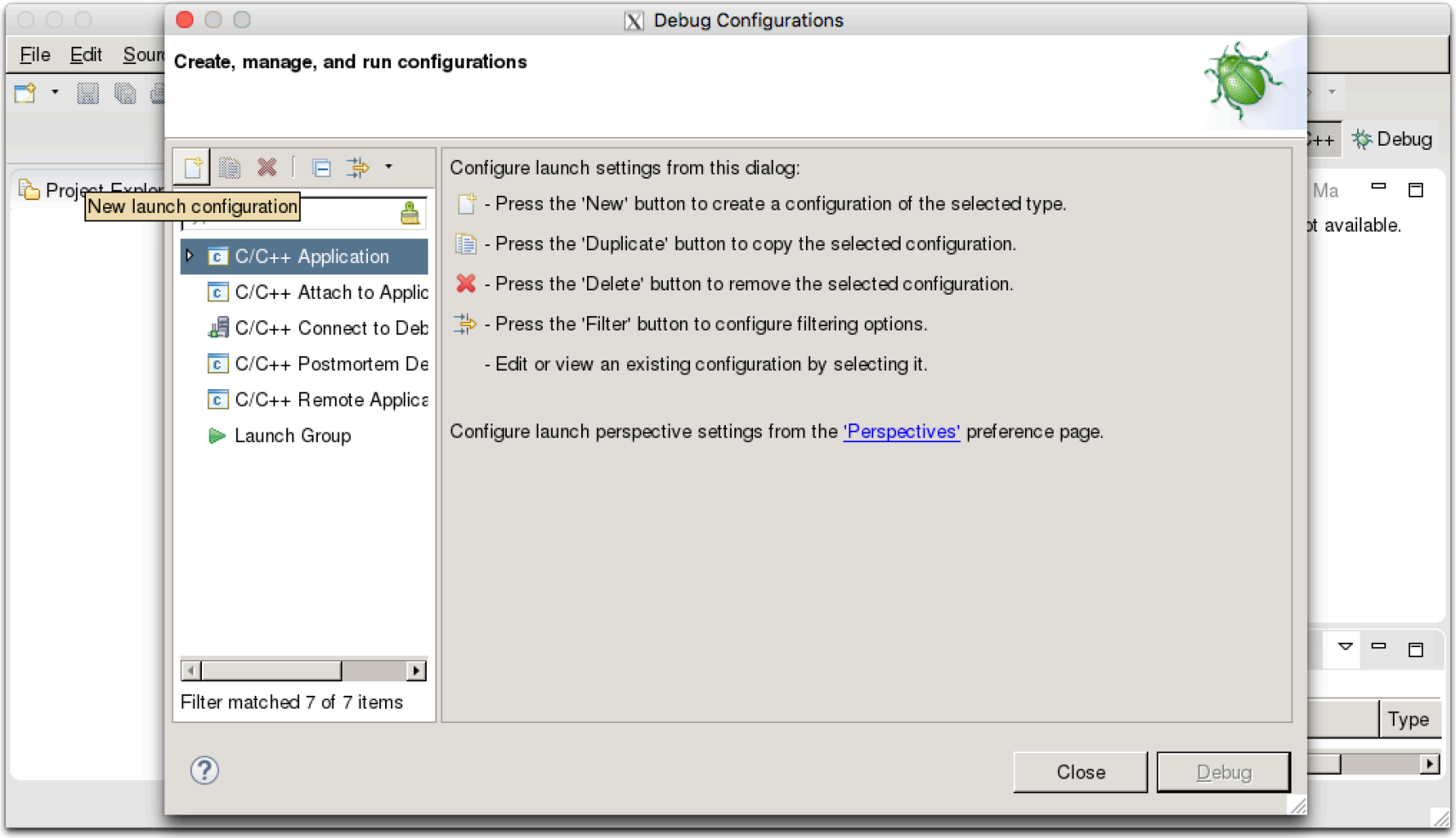
Choose *C/C++ Application*



Debug CUDA Program with Nsight EE

Setup

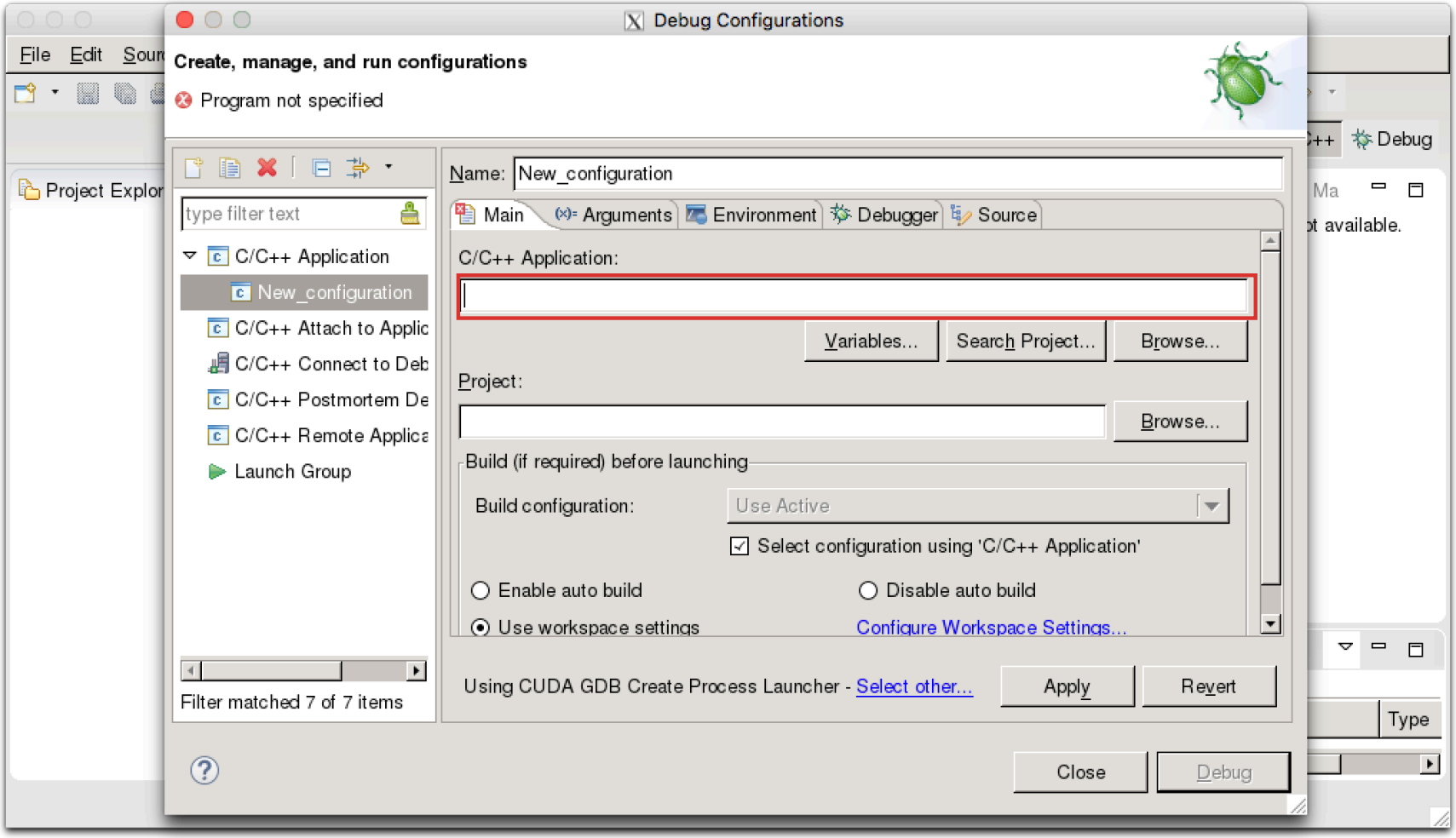
Create *New launch configuration*



Debug CUDA Program with Nsight EE

Setup

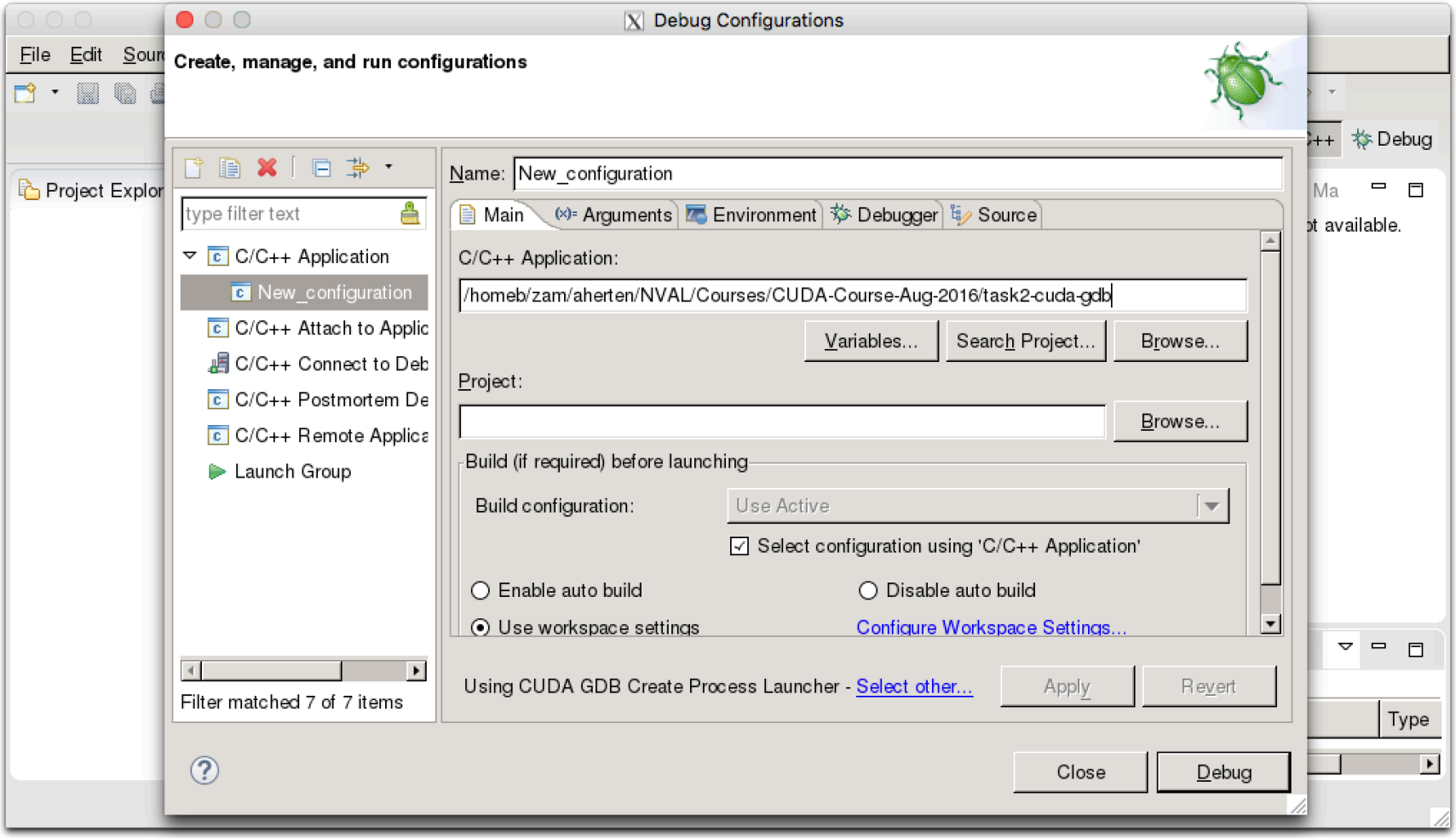
Insert path to executable



Debug CUDA Program with Nsight EE

Setup

Insert path to executable

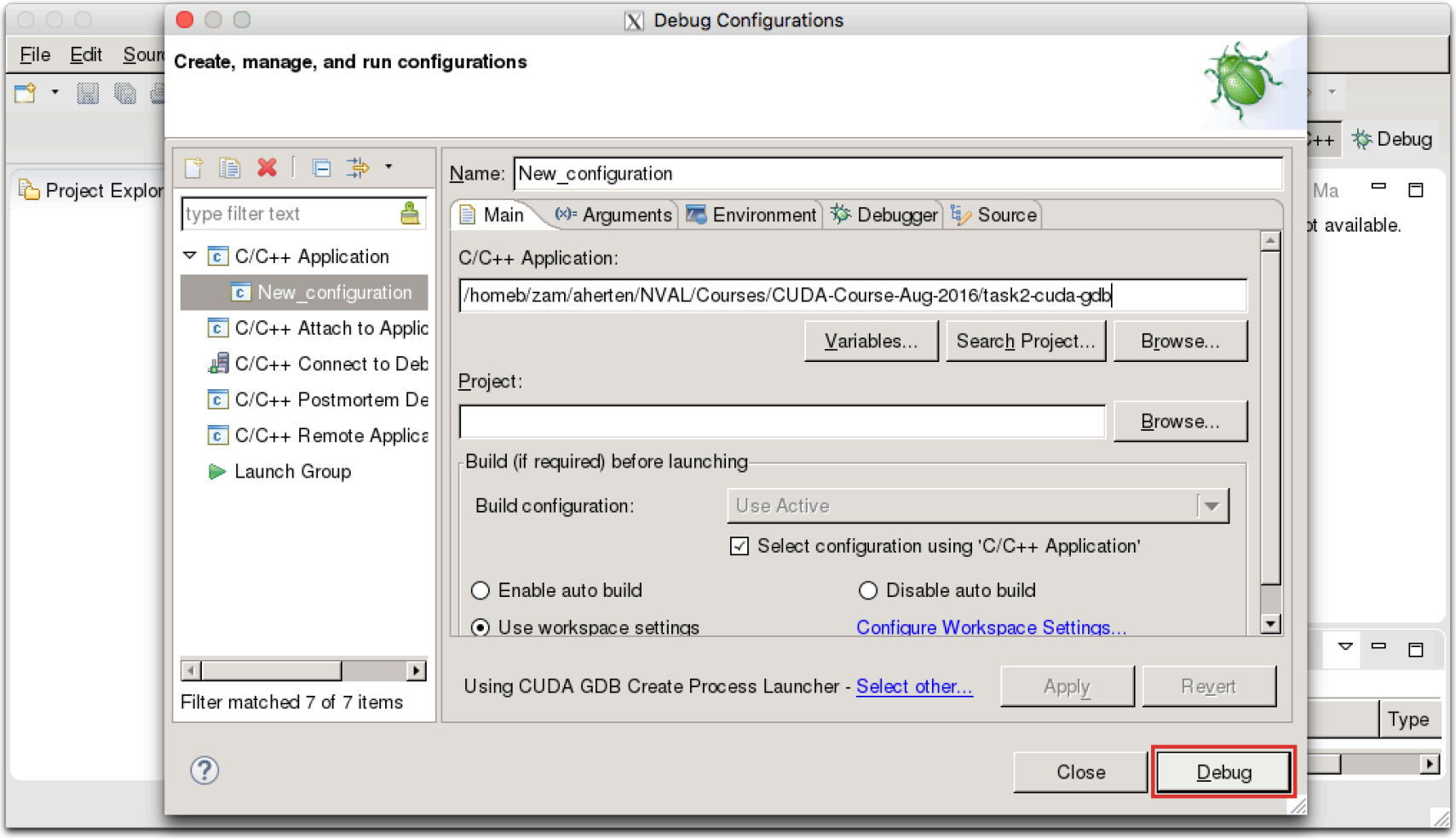


Debug CUDA Program with Nsight EE

Setup

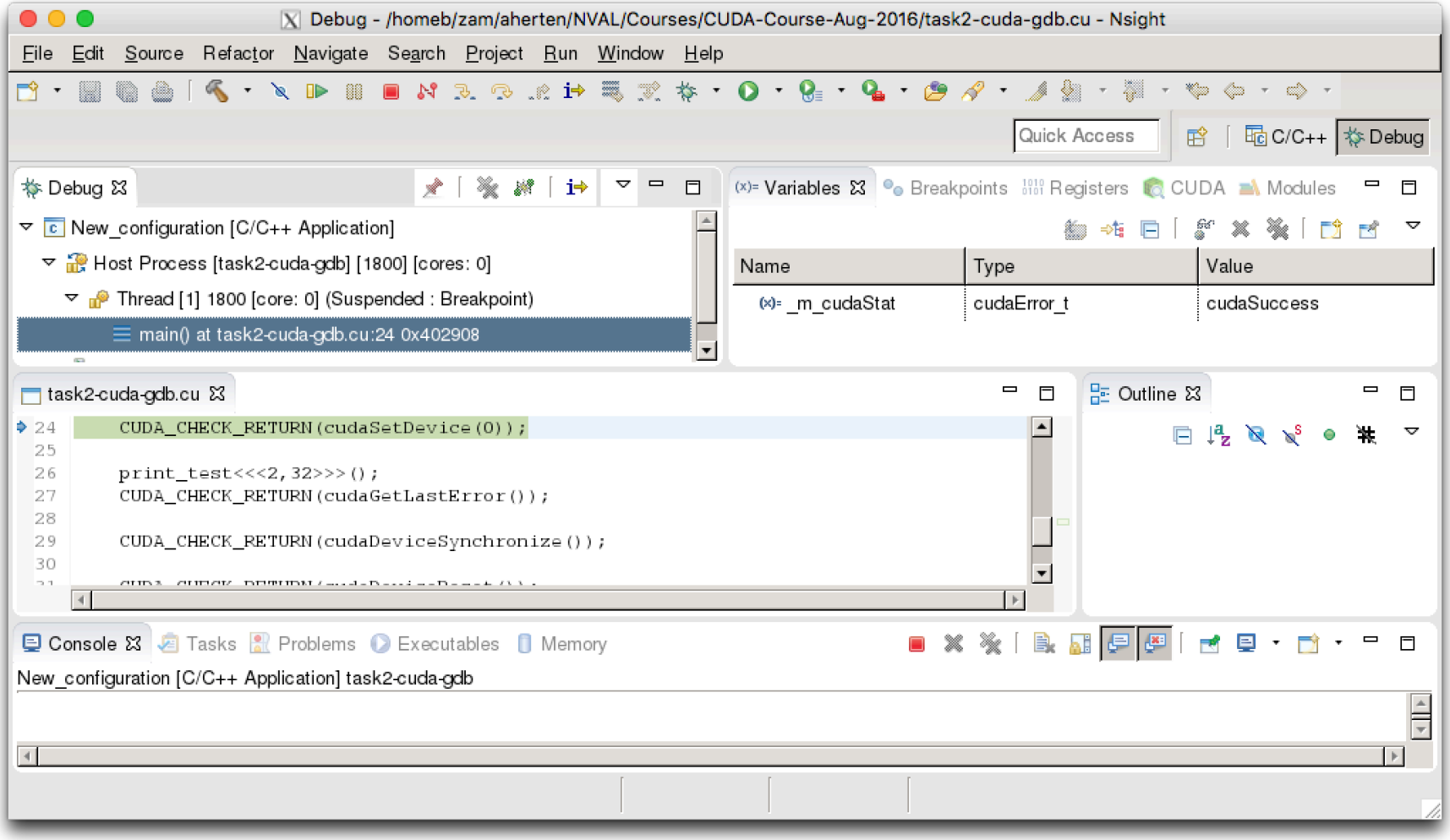
JÜLICH FORSCHUNGSZENTRUM

Click *Debug*



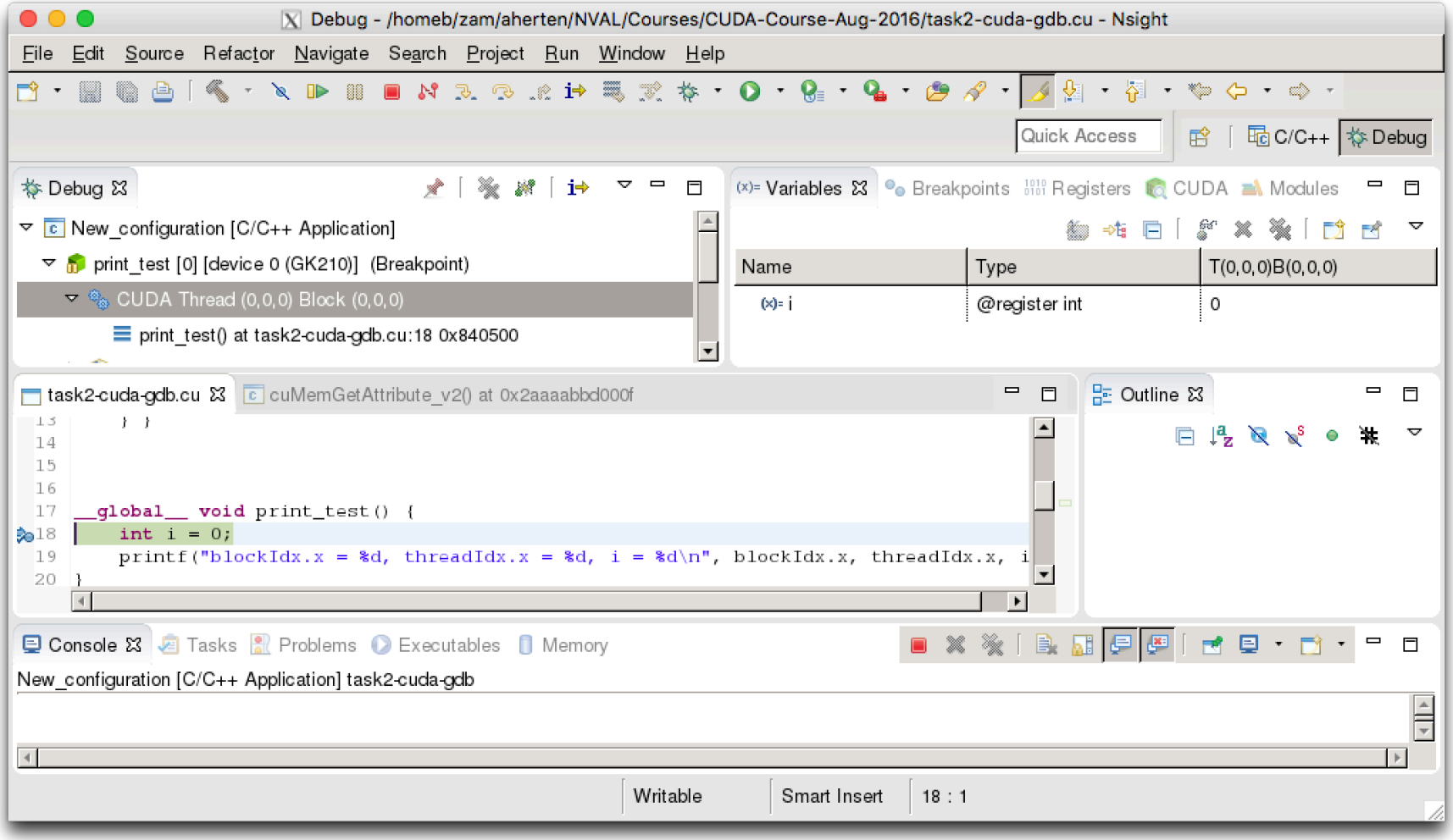
Debug CUDA Program with Nsight EE

Usage



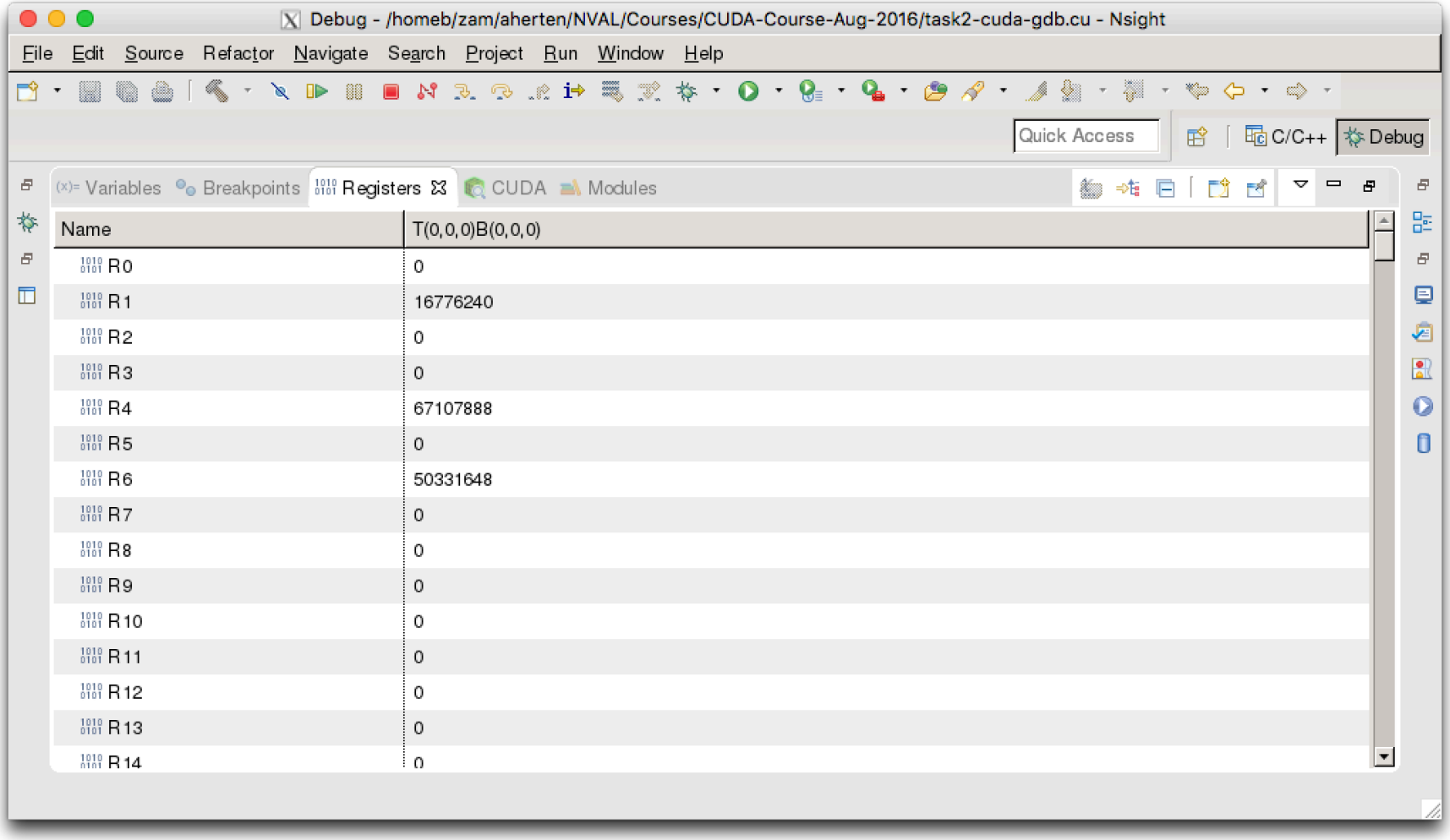
Debug CUDA Program with Nsight EE

Usage



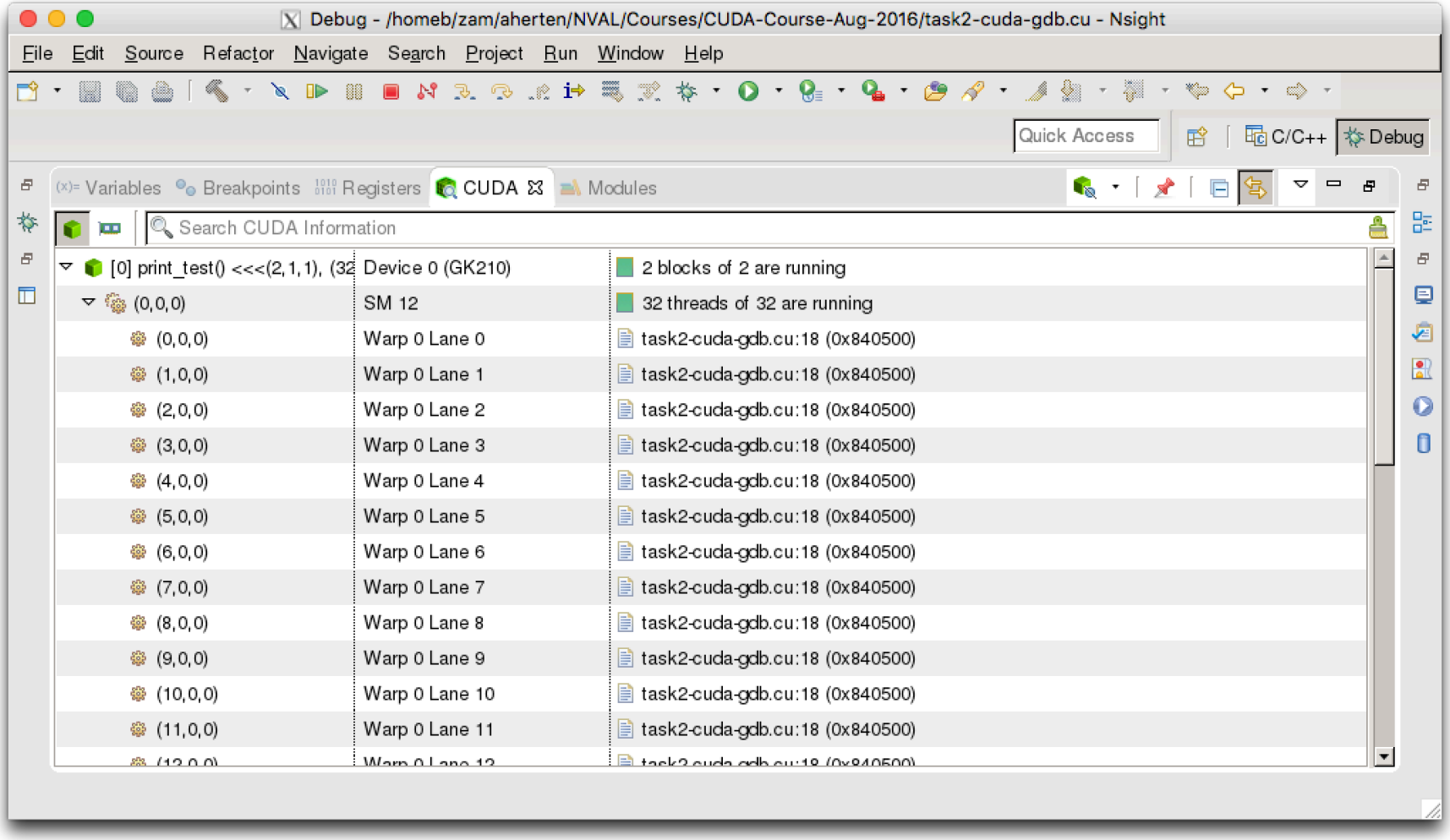
Debug CUDA Program with Nsight EE

Usage



Debug CUDA Program with Nsight EE

Usage



Task 1

Use `cuda-memcheck` to identify error

JÜLICH FORSCHUNGSZENTRUM

- Location of code: `CUDATools/Exercises/Tasks/`
- Steps
  - Build:
    `make task1-cuda-memcheck`
  - Run:
    `cuda-memcheck ./task1-cuda-memcheck`
  - Fix!
    Look into `task1-cuda-memcheck.cu` and fix error; `cuda-memcheck` should run without errors!

JURECA Getting Started

```
module load GCC/4.9.3-2.25 CUDA/7.5.18
salloc --reservation=gpu-kurs --partition=gpus --nodes=1 --time=1:30:00 --gres=mem128,gpu:4
srun cuda-memcheck ./task1-cuda-memcheck
```

Task 2

Use Nsight Eclipse Edition to debug program

JÜLICH FORSCHUNGSZENTRUM

- Location of code: `CUDATools/Exercises/Tasks/`
- Steps
  - Build program:
    `make task2-cuda-gdb`
  - 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.

JURECA Interactivity

```
module load GCC/4.9.3-2.25 CUDA/7.5.18
salloc --reservation=gpu-kurs --partition=gpus --nodes=1 --time=1:30:00 --gres=mem128,gpu:4
srun --forward-x --pty /bin/bash -i
nsight
```

Profiling

Motivation for Measuring Performance

JÜLICH FORSCHUNGSZENTRUM

- **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

- Easy access to hardware counters (PAPI, CUPTI)

nvprof

Command-line GPU profiler

JÜLICH FORSCHUNGSZENTRUM

- 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
    Handy together with `--analysis-metrics` (gather data for *analysis mode* of Visual Profiler)
  - `--print-gpu-trace`: Show trace of function calls
  - `--unified-memory-profiling per-process-device`: Print unified memory profiling information
    Prevent zero-copy fallback with `CUDA_MANAGED_FORCE_DEVICE_ALLOC=1`
    Attention with Multi-GPU setups!
  - `--help`: For all the rest…

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

nvprof

Example I

Start via `nvprof` *PROGRAM*

Tasks — ssh jureca -X — ssh jureca -X

..urse-2016-Aug | jureca

```
aherten@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%  4.0960us         1  4.0960us  4.0960us  4.0960us  scale(float, float*, float*, int)

==32741== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.15%  215.42ms         2  107.71ms  44.070us  215.37ms  cudaMallocManaged
  0.58%  1.2695ms       166  7.6470us     100ns  298.86us  cuDeviceGetAttribute
  0.09%  204.39us         2  102.19us  20.279us  184.11us  cudaFree
  0.06%  138.65us         2  69.327us  69.133us  69.521us  cuDeviceTotalMem
  0.06%  124.73us         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     355ns  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

Example II

JÜLICH FORSCHUNGSZENTRUM

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

Tasks — ssh jureca -X — ssh jureca -X

..urse-2016-Aug | jureca

```
aherten@jrl11:~/NVAL/Courses/CUDA-Course-Aug-2016$ srun nvprof --metrics inst_executed,inst_issued,issued_ipc,flop_count_sp,flop_cou
nt_dp --cpu-profiling on ./task3-scale_vector_um
==787== NVPROF is profiling process 787, command: ./task3-scale_vector_um
==787== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==787== Profiling application: ./task3-scale_vector_um
==787== Profiling result:
==787== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K80 (0)"
       Kernel: scale(float, float*, float*, int)
         1                            inst_executed                     Instructions Executed        1088        1088        1088
         1                              inst_issued                       Instructions Issued        1576        1576        1576
         1                               issued_ipc                                Issued IPC    0.094433    0.094433    0.094433
         1                            flop_count_sp  Floating Point Operations(Single Precisi        2048        2048        2048
         1                            flop_count_dp  Floating Point Operations(Double Precisi           0           0           0
Passed!

======== CPU profiling result (bottom up):
38.89% cudbgGetAPIVersion
| 38.89% start_thread
|   38.89% clone
30.56% cuDevicePrimaryCtxRetain
| 30.56% cudart::contextStateManager::initPrimaryContext(cudart::device*)
```

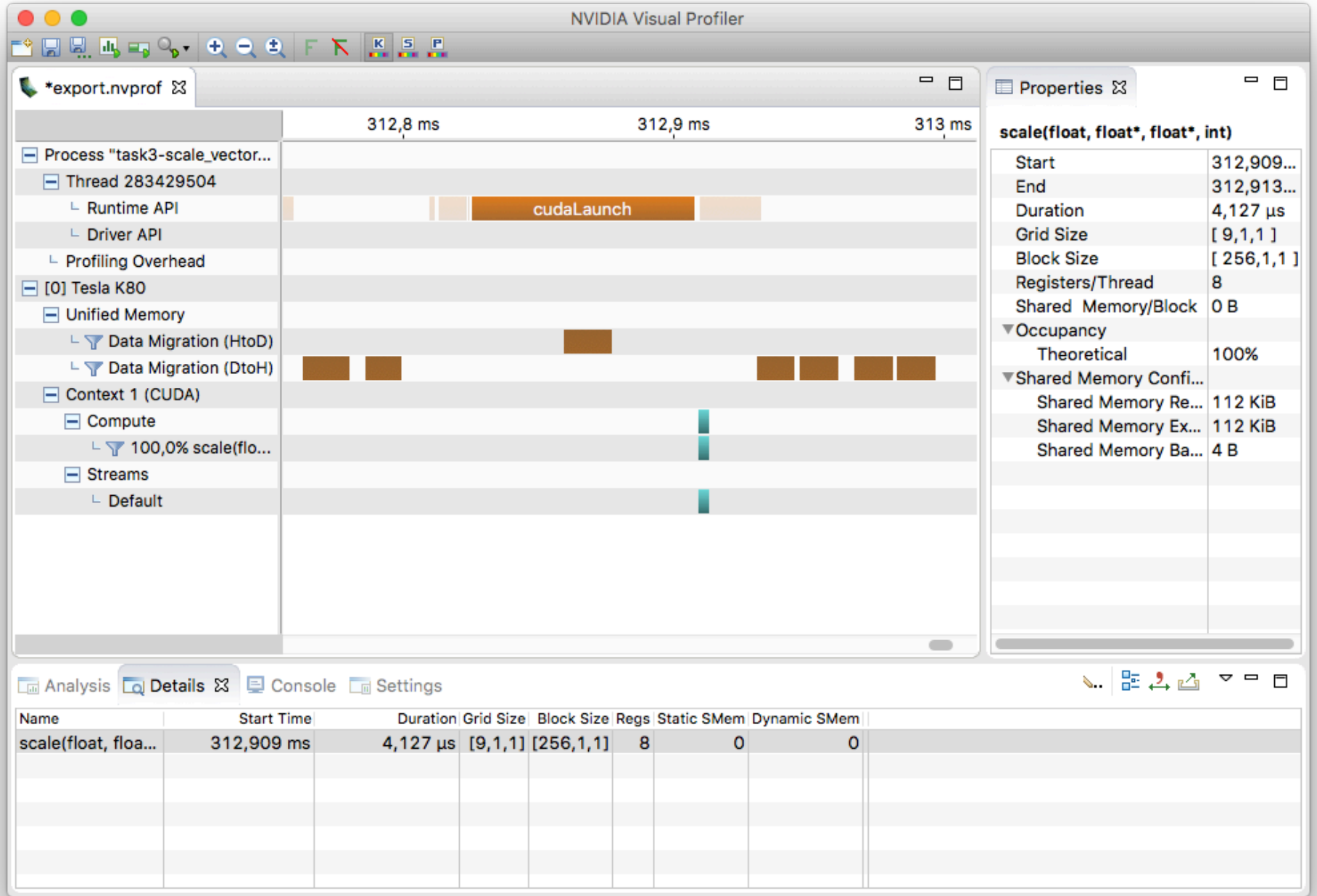
Visual Profiler

The insight provider

JÜLICH FORSCHUNGSZENTRUM

- 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`


NVIDIA Visual Profiler
*export.nvprof
312,8 ms
312,9 ms
313 ms
Process "task3-scale_vector...
Thread 283429504
Runtime API
Driver API
Profiling Overhead
[0] Tesla K80
Unified Memory
Data Migration (HtoD)
Data Migration (DtoH)
Context 1 (CUDA)
Compute
100,0% scale(flo...
Streams
Default
cudaLaunch
Properties
scale(float, float*, float*, int)
Start 312,909...
End 312,913...
Duration 4,127 µs
Grid Size [9,1,1]
Block Size [256,1,1]
Registers/Thread 8
Shared Memory/Block 0 B
Occupancy
Theoretical 100%
Shared Memory Confi...
Shared Memory Re... 112 KiB
Shared Memory Ex... 112 KiB
Shared Memory Ba... 4 B
Analysis
Details
Console
Settings
Name Start Time Duration Grid Size Block Size Regs Static SMem Dynamic SMem
scale(float, floa... 312,909 ms 4,127 µs [9,1,1] [256,1,1] 8 0 0

Task 3

Analyze and profile `scale_vector_um`

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
   - Import to Visual Profiler
   - 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 (`nvvp`)
   - Start, profile, and run guided analysis in Visual Profiler

JURECA Reminder

```
module load GCC/4.9.3-2.25 CUDA/7.5.18
salloc --reservation=gpu-kurs --partition=gpus --nodes=1 --time=1:30:00 --gres=mem128,gpu:2


  srun nvprof ./scale_vector_um                    srun --forward-x --pty /bin/bash -i
                                                   nvvp
```

Summary

JÜLICH FORSCHUNGSZENTRUM

- NVIDIA has handy tools to gain insight into application
- Debugging:
  - `cuda-memcheck`
  - `cuda-gdb`
  - Nsight Eclipse Edition
- Profiling:
  - nvprof
  - Visual Profiler
- All come with the CUDA Toolkit

Thank you!

Slides based on Jiri Kraus' *CUDA Tools* presentation