



GPU Teaching Kit

Accelerated Computing



Lecture 11 – Introduction to CUDA C

Introduction to the CUDA Toolkit

Part I
Developer Resources

Where CUDA C fits in the big picture?

3 Ways to Accelerate Applications

Applications

Libraries

Compiler
Directives

Programming
Languages

Easy to use
Most Performance

Easy to use
Portable code

Most Performance
Most Flexibility

3

NVIDIA

ILLINOIS

Libraries: Easy, High-Quality Acceleration

- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications

4

NVIDIA

ILLINOIS

GPU Accelerated Libraries

Linear Algebra
FFT, BLAS,
SPARSE, Matrix



cULA|tools



C U S P

Numerical & Math
RAND, Statistics



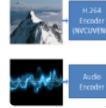
ArrayFire



Data Struct. & AI
Sort, Scan, Zero Sum



Visual Processing
Image & Video



NVIDIA
Video
Encode



5

NVIDIA

ILLINOIS

Vector Addition in Thrust

```
thrust::device_vector<float> deviceInput1(inputLength);
thrust::device_vector<float> deviceInput2(inputLength);
thrust::device_vector<float> deviceOutput(inputLength);

thrust::copy(hostInput1, hostInput1 + inputLength, deviceInput1.begin());
thrust::copy(hostInput2, hostInput2 + inputLength, deviceInput2.begin());

thrust::transform(deviceInput1.begin(), deviceInput1.end(), deviceInput2.begin(),
                 deviceOutput.begin(), thrust::plus<float>());
```

6

NVIDIA

ILLINOIS

Compiler Directives: Easy, Portable Acceleration

- **Ease of use:** Compiler takes care of details of parallelism management and data movement
- **Portable:** The code is generic, not specific to any type of hardware and can be deployed into multiple languages
- **Uncertain:** Performance of code can vary across compiler versions

7



OpenACC

- Compiler directives for C, C++, and FORTRAN

```
#pragma acc parallel loop
copyin(input1[0:inputLength],input2[0:inputLength]),
copyout(output[0:inputLength])
for(i = 0; i < inputLength; ++i) {
    output[i] = input1[i] + input2[i];
}
```

8



Programming Languages: Most Performance and Flexible Acceleration

- **Performance:** Programmer has best control of parallelism and data movement
- **Flexible:** The computation does not need to fit into a limited set of library patterns or directive types
- **Verbose:** The programmer often needs to express more details

9



GPU Programming Languages

Numerical analytics ➤

MATLAB Mathematica, LabVIEW

Fortran ➤

CUDA Fortran

C ➤

CUDA C

C++ ➤

CUDA C++

Python ➤

PyCUDA, Copperhead, Numba

F# ➤

Alea.cuBase

10



CUDA - C

Applications

Libraries

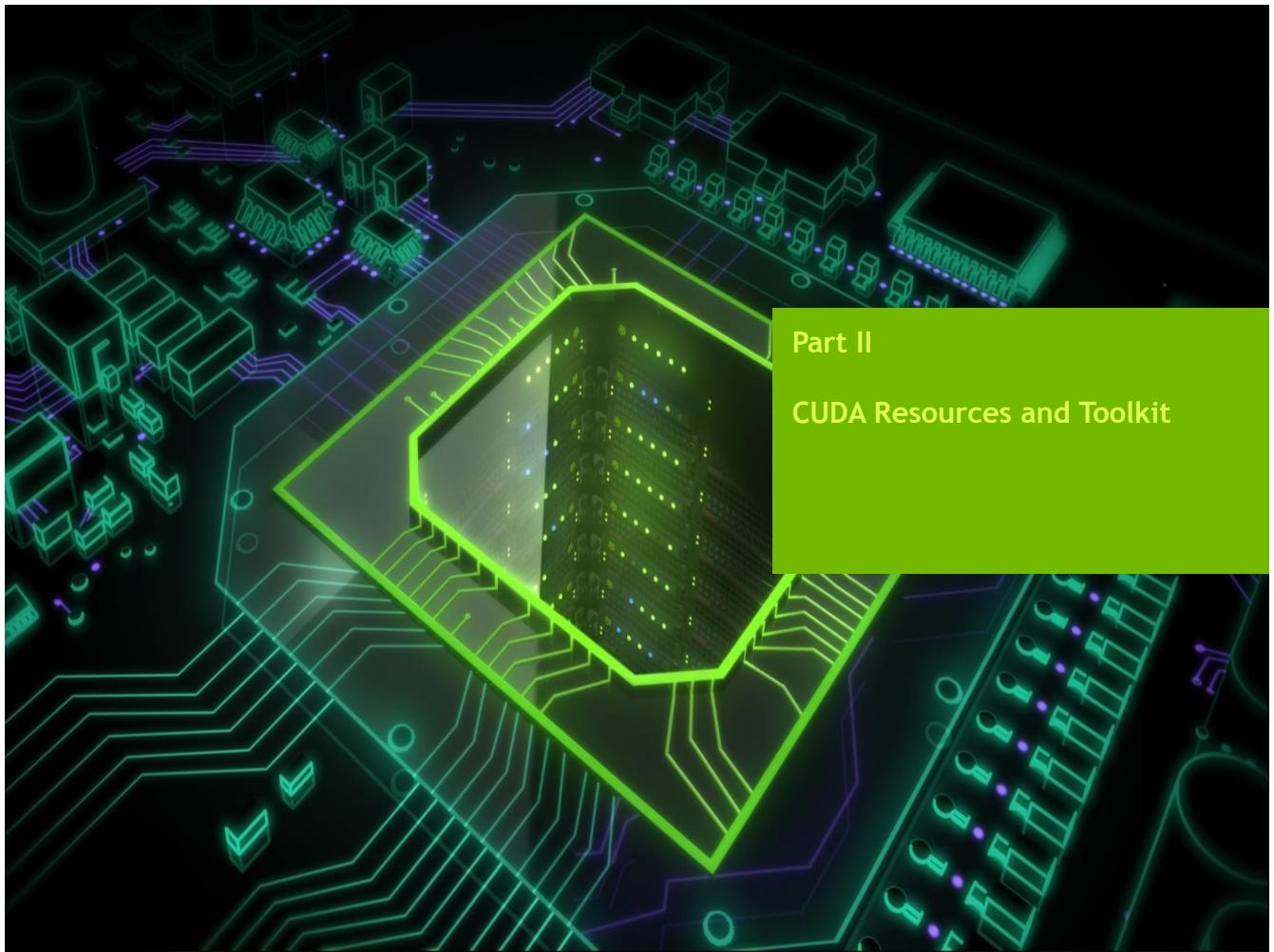
Easy to use
Most Performance

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

```
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

```
__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

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

Output:

```
$ 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

<https://developer.nvidia.com/debugging-solutions>

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-memcheck>

CUDA-MEMCHECK

- Try cuda-memcheck

Instructions:

1. Build & Run a source code
2. Run with cuda-memcheck
%> cuda-memcheck ./a.out
3. Add nvcc flags “-Xcompiler -rdynamic -lineinfo”
4. Rebuild & Run with cuda-memcheck
5. Fix any illegal write that you may have

<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 the exercise 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

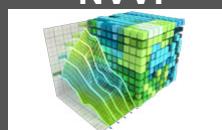
<http://docs.nvidia.com/cuda/cuda-gdb>

Developer Tools - Profilers

NSIGHT



NVVP

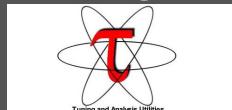


NVPROF

```
--2000ms Profiling results
Time(s) Time Calls Avg Min Max Name
49.888 866.0ms 58473 1.717us 1.594ns 2.810us void th
(0.000 100.0ms 256 1.717us 1.594ns 2.810us void th
23.339 440.0ms 252662 1.747us 1.536ns 2.369us void th
L:\src\test\detail\device\cudamemcheck\func-thruth\detall\fill_Fat
17.875 39.0ms 1 4.838us 1.483ns 1.483ns void th
2.908 51.819ms 200 259.09us 240.97us 264.03us kerMake
1.150 1.598ms 200 55.37us 53.37us 57.37us kerGpu
0.938 16.198ms 200 89.991us 71.846us 98.751us kerGpu
0.722 1.098ms 200 44.001us 41.001us 47.001us kerMem
0.699 12.071ms 200 68.376us 59.680us 82.384us kerMem
0.636 16.993ms 200 55.963us 52.690us 68.268us kerMem
0.523 1.221ms 200 1.221us 1.221us 1.221us kerMem
0.128 2.1342ms 1 2.1342us 2.1342us 2.1342us void th
```

NVIDIA Provided

TAU



3rd Party

VampirTrace



<https://developer.nvidia.com/performance-analysis-tools>

NVPROF

Command Line Profiler

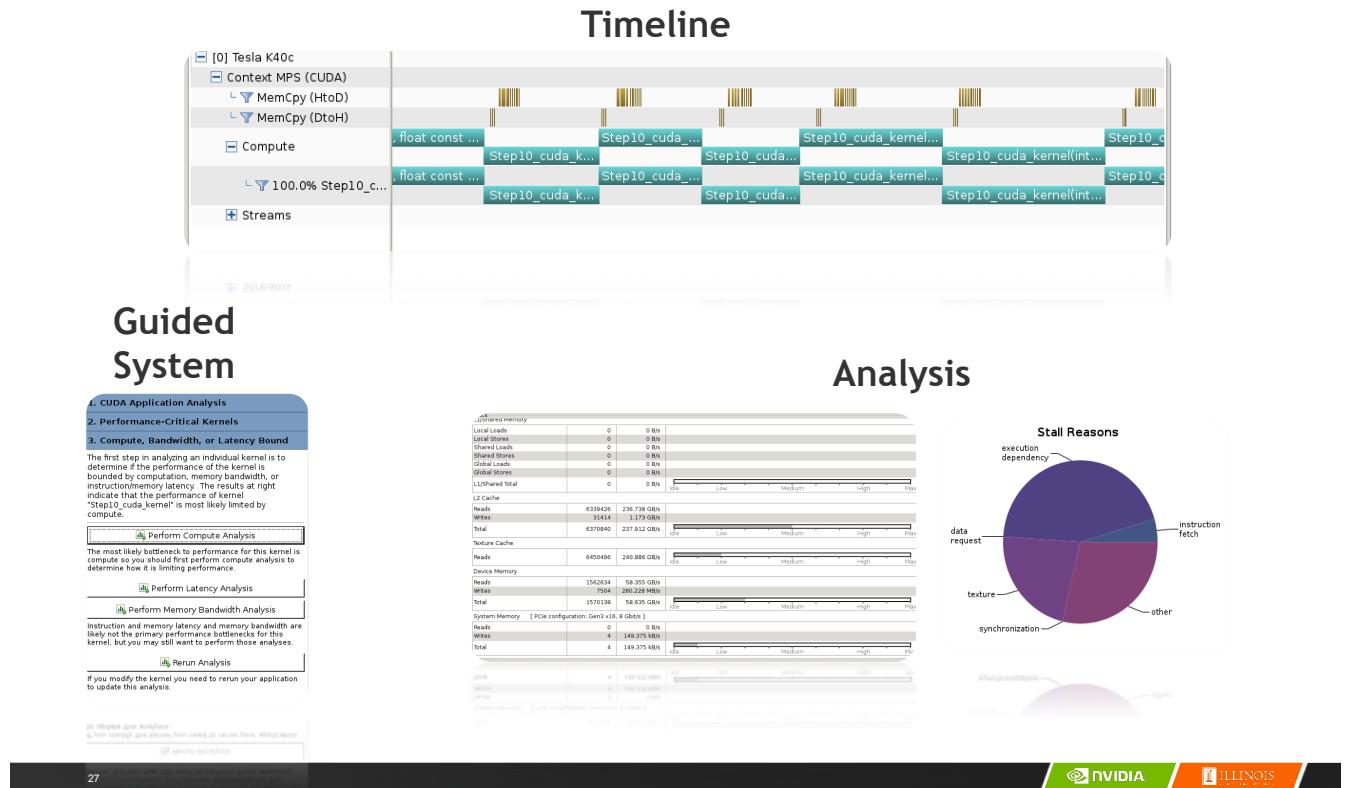
- Compute time in each kernel
- Compute memory transfer time
- Collect metrics and events
- Support complex process hierarchy's
- Collect profiles for NVIDIA Visual Profiler
- No need to recompile

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)



Example 4: NVVP

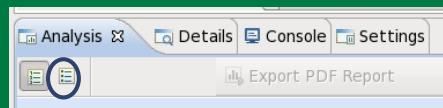
Instructions:

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 2nd 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.

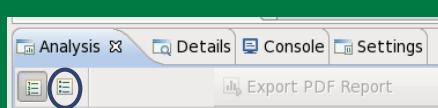
Example 4: NVVP

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

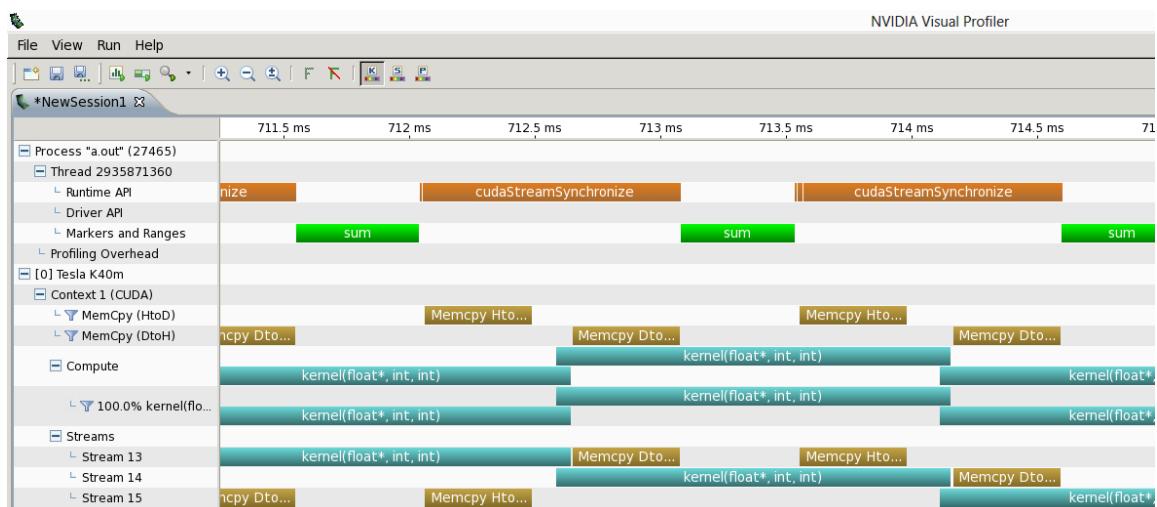
NVTX

- 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

<http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/>

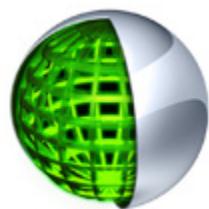


NVTX Profile



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

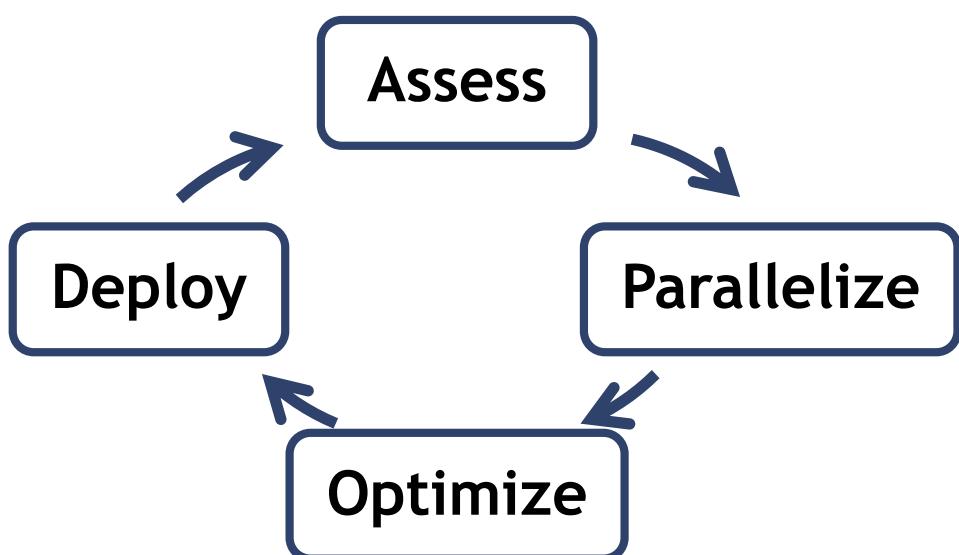
Instructions:

1. Run nsight and *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

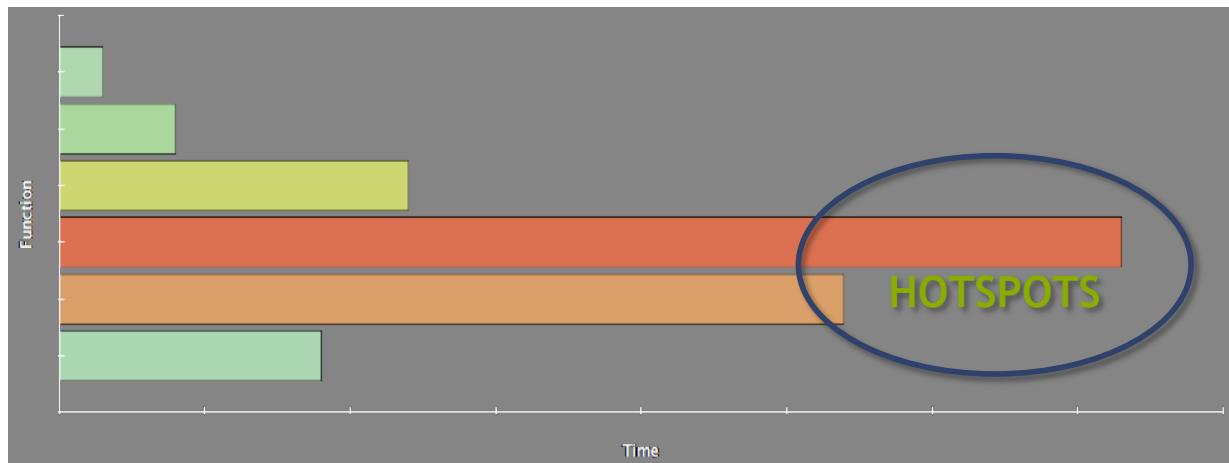
Profiler Summary

- 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



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

Parallelize

Applications

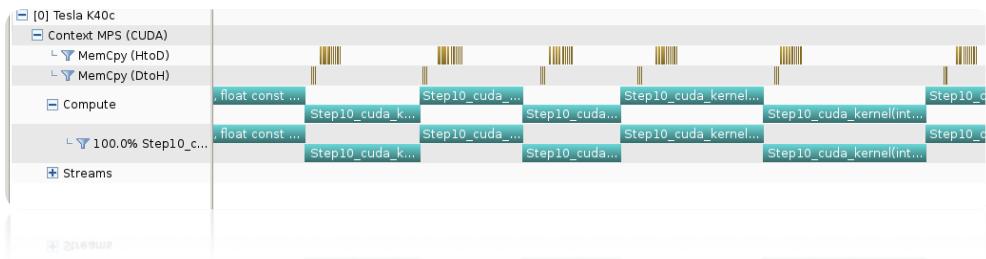
Libraries

Compiler
Directives

Programming
Languages

Optimize

Timeline



Guided System

1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is limited by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10_cuda_kernel" is most likely limited by compute.

4. Perform Compute Analysis

The most likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.

5. Perform Latency Analysis

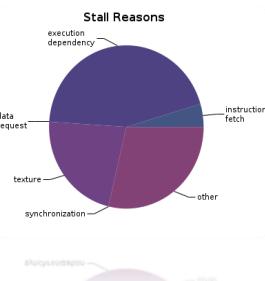
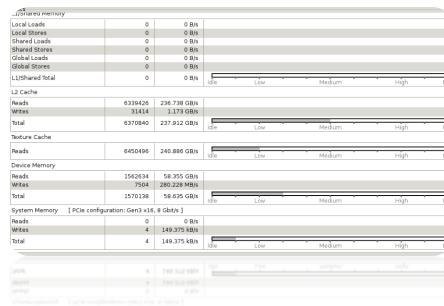
6. Perform Memory Bandwidth Analysis

Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel. But you may still need to perform these analyses.

7. Return Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Analysis



Bottleneck Analysis

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

129 GB/s → 84 GB/s



gpuTranspose_kernel(int, int, float const *, float*)	
Start	547.303 ms (
End	547.716 ms (
Duration	413.872 µs
Grid Size	[64, 64, 1]
Block Size	[32, 32, 1]
Registers/Thread	10
Shared Memory/Block	4 kB
▼ Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	▲ 5.9%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficiency	97.1%
▼ Occupancy	
Achieved	86.7%
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Requested	48 kB
Shared Memory Executed	48 kB

Shared Memory Alignment and Access Pattern

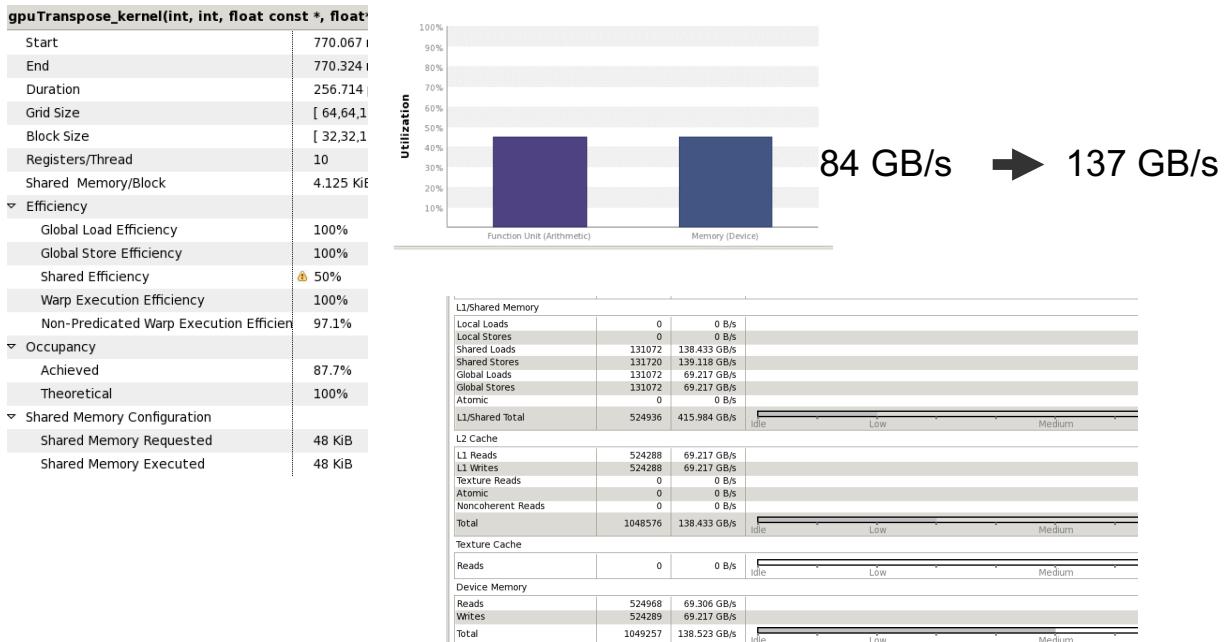
Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

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.

▼ Line / File main.cu - /home/juitjens/code/CudaHandsOn/Example19

49 Shared Load Transactions/Access = 16, Ideal Transactions/Access = 1 [2097152 transactions for 131072 total executions]

Performance Analysis



41



NVIDIA®

GPU Teaching Kit



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](#).