



GPU Teaching Kit

Accelerated Computing



# 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

```
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;  
}
```

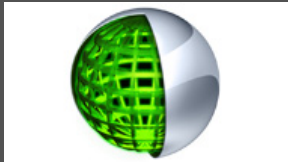
Output:

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

- `mykernel` (does nothing, somewhat anticlimactic!)

# Developer Tools - Debuggers

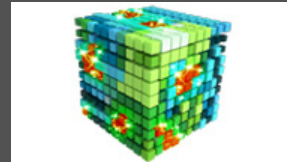
**NSIGHT**



**CUDA-GDB**



**CUDA MEMCHECK**



**NVIDIA Provided**

**allinea**  
DDT

**TotalView®**

**3<sup>rd</sup> 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>

## 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:

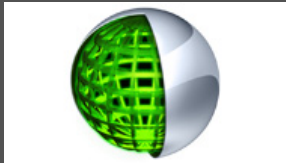
```
(cuda-gdb) b main           //set break point at main
(cuda-gdb) r                 //run application
(cuda-gdb) l                 //print line context
(cuda-gdb) b foo             //break at kernel foo
(cuda-gdb) c                 //continue
(cuda-gdb) cuda thread       //print current thread
(cuda-gdb) 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

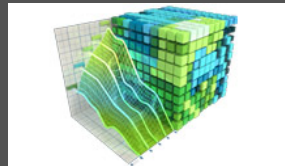
<http://docs.nvidia.com/cuda/cuda-gdb>

# Developer Tools - Profilers

NSIGHT



NVVP

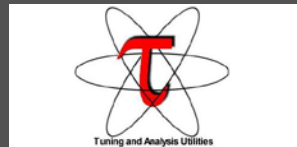


NVPROF

```
--20561== Profiling result:
Time(s)    Time    Calls    Avg    Min    Max    Name
49.88%    866.6ms    504758    1.727ms    1.560ms    2.815ms    void th
unt_thrust::detail::device_generate_function::thrust::detail::fill
25.33%    449.0ms    252662    1.741ms    1.536ms    2.308ms    void th
t_thrust::detail::device_generate_function::thrust::detail::fill_fo
17.87%    296.6ms    200    1.483ms    1.264ms    1.725ms    kerComp
2.98%    51.81ms    200    259.89us    246.97us    264.83us    kerTake
1.16%    20.17ms    501    40.26us    30us    17.477ms    [CUDA m
0.93%    16.19ms    200    80.99us    71.84us    90.75us    kerColl
0.73%    12.63ms    400    31.58us    14.72us    50.43us    [CUDA m
0.69%    12.07ms    200    60.37us    59.68us    62.36us    kerForm
0.63%    10.99ms    200    54.96us    52.68us    58.20us    kerTake
0.32%    5.552ms    200    27.76us    22.55us    33.15us    [CUDA m
0.12%    2.134ms    1    2.134ms    2.134ms    2.134ms    void th
```

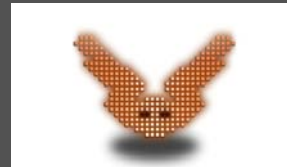
NVIDIA Provided

TAU



Tuning and Analysis Utilities

VampirTrace



3rd Party

<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

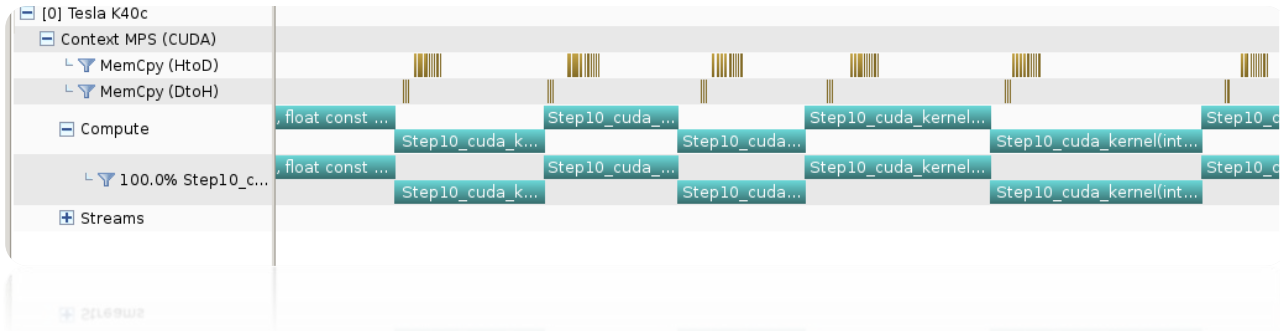
# Example 4: nvprof

## 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)

## 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 bounded 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.

**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.

**Perform Latency Analysis**

**Perform Memory Bandwidth Analysis**

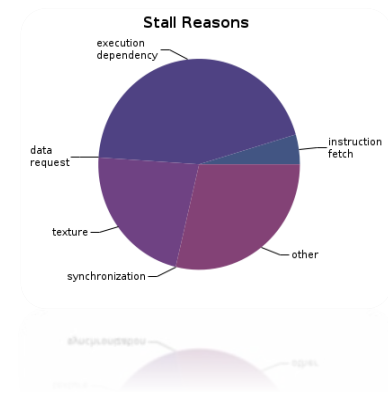
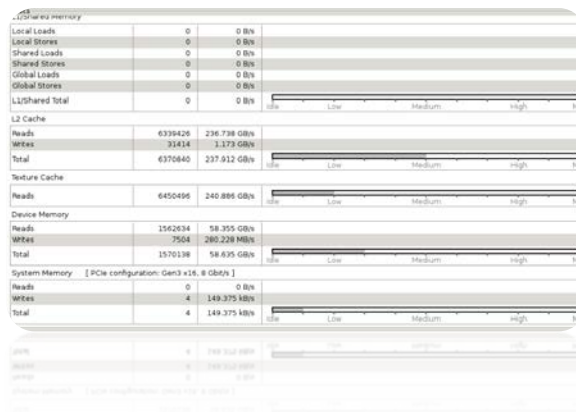
Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

**Rerun Analysis**

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

go to the next step

## 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<sup>nd</sup> Browse

Select profile.metrics

Click Finish

### 2. Explore Timeline

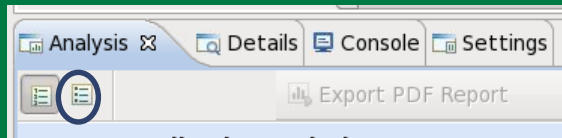
Control + mouse drag in timeline to zoom in

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.

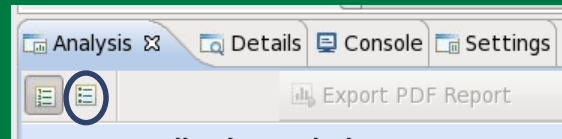
If you load just metrics the timeline looks odd but metrics are correct.

# 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

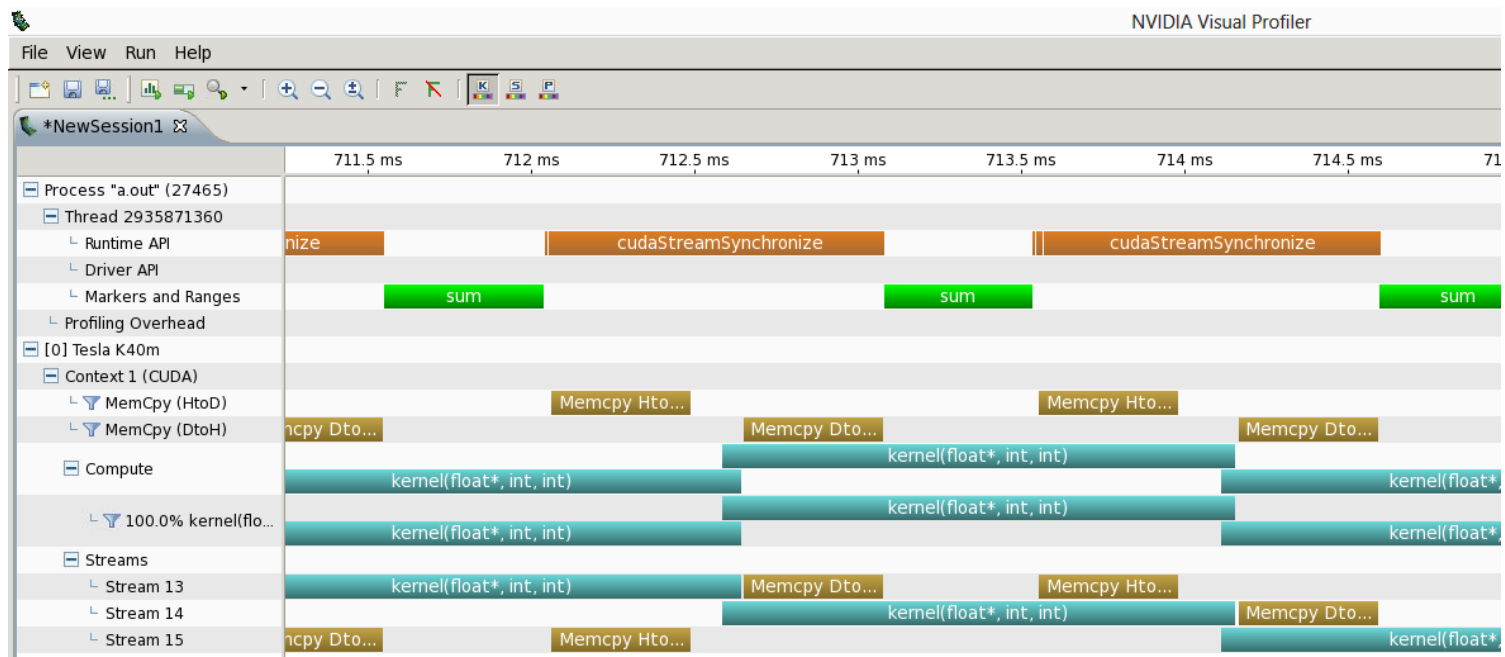


# NVTX

- 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/>

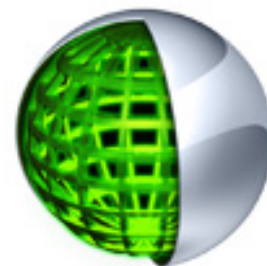
# NVTX Profile





# NSIGHT

- CUDA enabled Integrated Development Environment
  - Source code editor: syntax highlighting, code refactoring, etc
  - Build Manger
  - 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

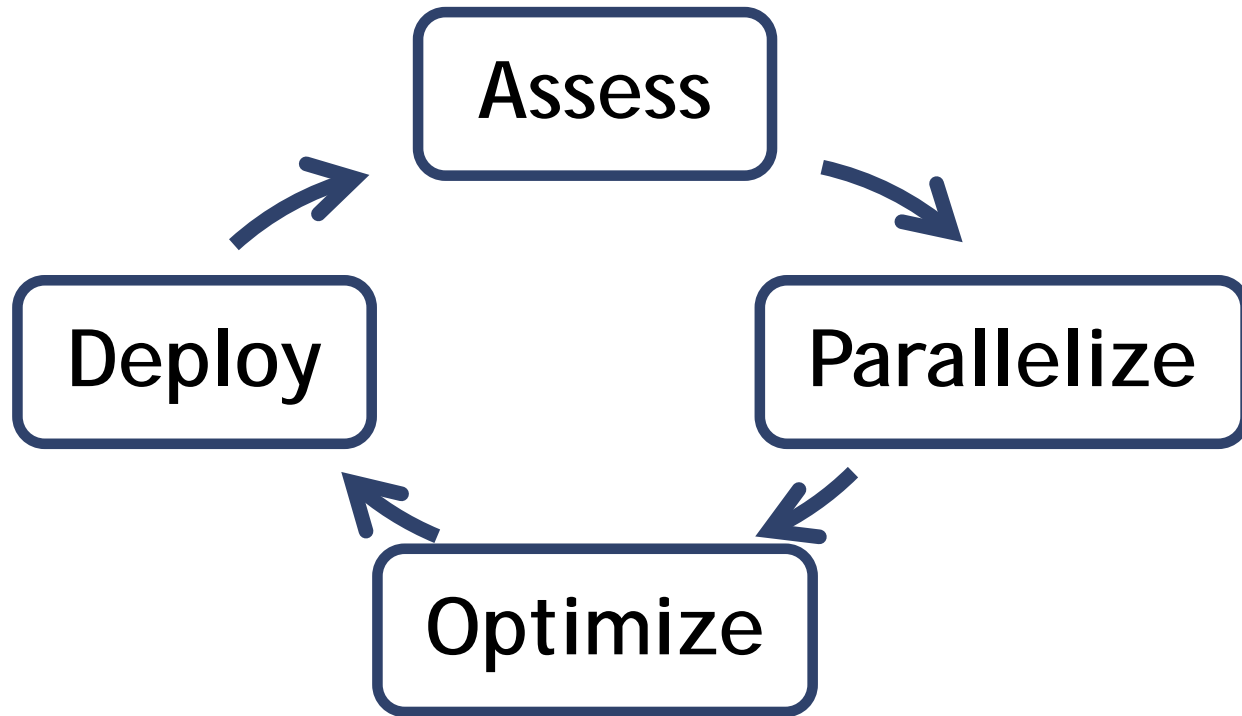
Instructions:

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

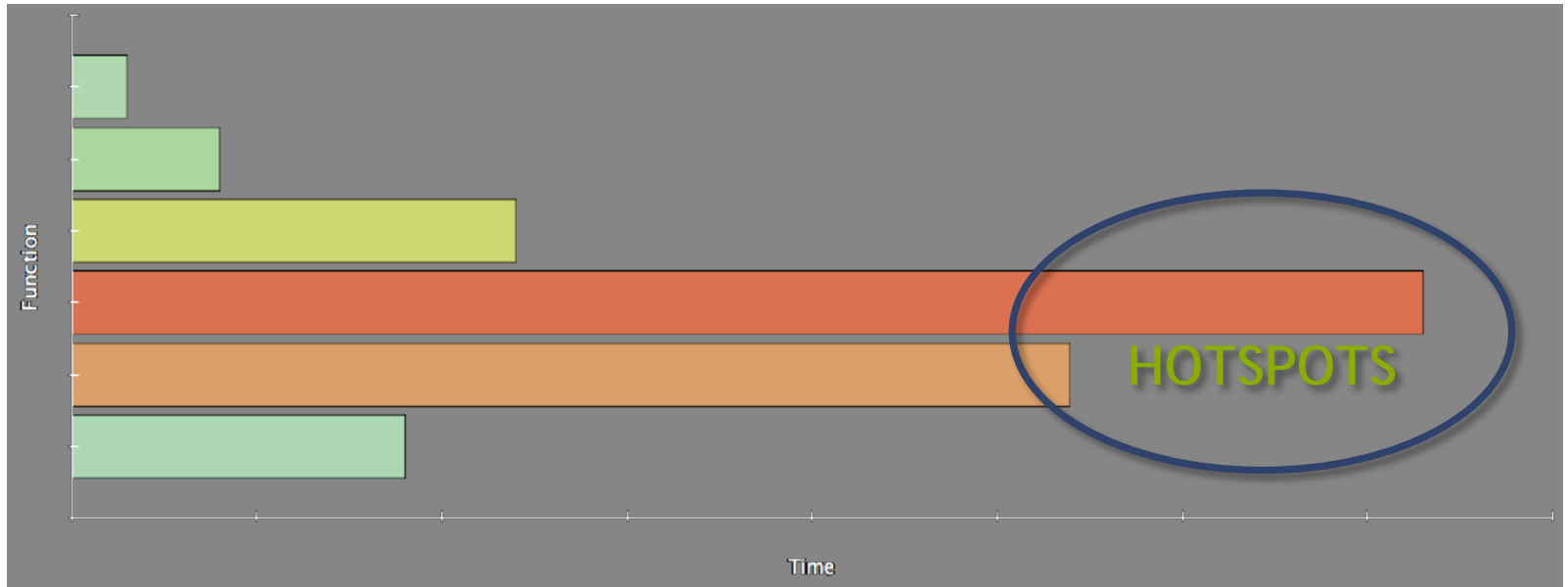
# Profiler Summary

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

# Optimization



# Assess



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

# Parallelize

Applications

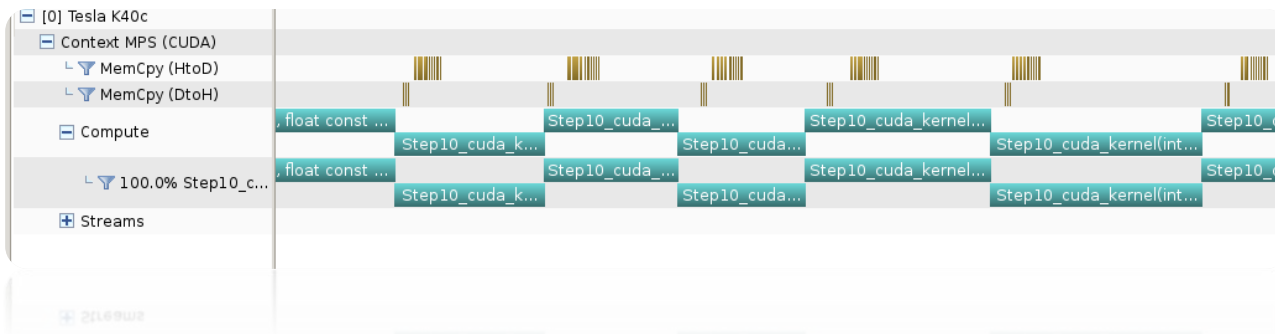
Libraries

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 bounded 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.

**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.

**Perform Latency Analysis**

**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 want to perform those analyses.

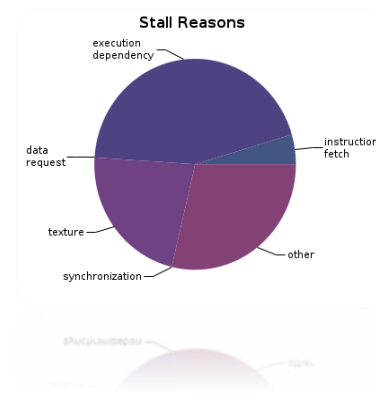
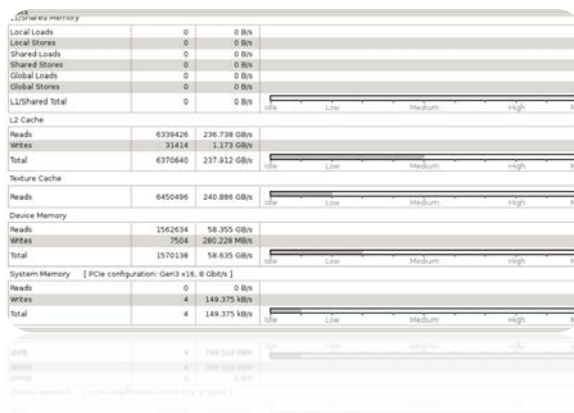
**Rerun Analysis**

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

go to page 400 Workphase  
is from workphase and passing from workphase to workphase from workphase

go to page 400 Workphase

## Analysis



# Bottleneck Analysis

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

129 GB/s ➡ 84 GB/s

## L1/Shared Memory

Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	2097152	1,351.979 GB/s
Shared Stores	131072	84.499 GB/s
Global Loads	131072	42.249 GB/s
Global Stores	131072	42.249 GB/s
Atomic	0	0 B/s
<b>L1/Shared Total</b>	<b>2490368</b>	<b>1,520.977 GB/s</b>



gpuTranspose_kernel(int, int, float const *, float*)	
Start	547.303 ms (5)
End	547.716 ms (5)
Duration	413.872 μs
Grid Size	[ 64,64,1 ]
Block Size	[ 32,32,1 ]
Registers/Thread	10
Shared Memory/Block	4 KiB
▼ 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 KiB
Shared Memory Executed	48 KiB

## ⚠ 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/jluitjens/code/CudaHandsOn/Example19

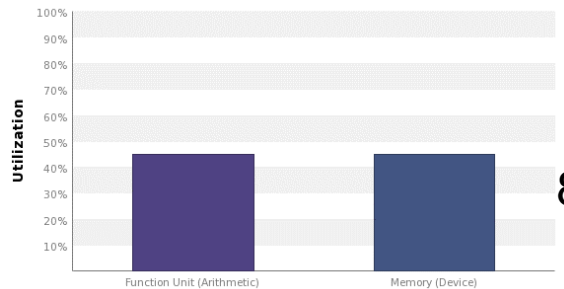
49 | Shared Load Transactions/Access = 16, Ideal Transactions/Access = 1 [ 2097152 transactions for 131072 total executions ]



# Performance Analysis

**gpuTranspose\_kernel(int, int, float const \*, float)**

Start	770.067
End	770.324
Duration	256.714
Grid Size	[ 64,64,1
Block Size	[ 32,32,1
Registers/Thread	10
Shared Memory/Block	4.125 KiB
▼ Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	⚠ 50%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficiency	97.1%
▼ Occupancy	
Achieved	87.7%
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB



84 GB/s ➔ 137 GB/s

L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	131072	138.433 GB/s	
Shared Stores	131720	139.118 GB/s	
Global Loads	131072	69.217 GB/s	
Global Stores	131072	69.217 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	524936	415.984 GB/s	
L2 Cache			
L1 Reads	524288	69.217 GB/s	
L1 Writes	524288	69.217 GB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Total	1048576	138.433 GB/s	
Texture Cache			
Reads	0	0 B/s	
Device Memory			
Reads	524968	69.306 GB/s	
Writes	524289	69.217 GB/s	
Total	1049257	138.523 GB/s	



## 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](https://creativecommons.org/licenses/by-nc/4.0/).