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	Compiler Directives	Programming Languages
Easy to use Most Performance	Easy to use Portable code	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

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
```

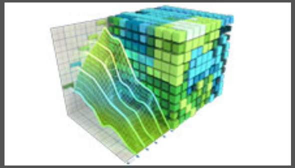
3. Fix Bug

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

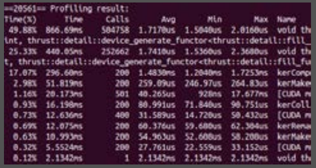
Developer Tools - Profilers

NSIGHT

NVVP



NVPROF



```
==20561== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.88%  866.69ms    504758  1.7170us  1.5040us  2.0160us  void th
int, thrust::detail::device_generate_functor<thrust::detail::fill_
 25.33%  440.05ms    252662  1.7410us  1.5360us  2.3680us  void th
t, thrust::detail::device_generate_functor<thrust::detail::fill_fu
 17.07%  296.60ms       200  1.4830ms  1.2040ms  1.7253ms  kerCompe
  2.98%  51.819ms       200  259.09us  246.97us  264.83us  kerMake
  1.16%  20.173ms       501  40.265us     928ns  17.677ms  [CUDA m
  0.93%  16.198ms       200  80.991us  71.840us  90.751us  kerColl
  0.73%  12.636ms       400  31.589us  14.720us  50.432us  [CUDA m
  0.69%  12.075ms       200  60.376us  59.680us  62.304us  kerRena
  0.63%  10.993ms       200  54.963us  52.608us  58.208us  kerMake
  0.32%  5.5524ms       200  27.761us  22.559us  33.152us  [CUDA m
  0.12%  2.1342ms         1  2.1342ms  2.1342ms  2.1342ms  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

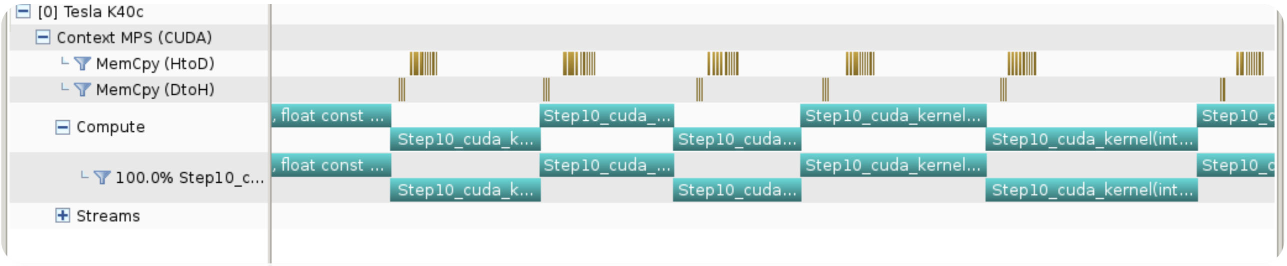
   %> nvprof --metrics gld_efficiency,gst_efficiency ./a.out
5. Store a timeline to load in NVVP

   %> nvprof –o profile.timeline ./a.out
6. Store analysis metrics to load in NVVP

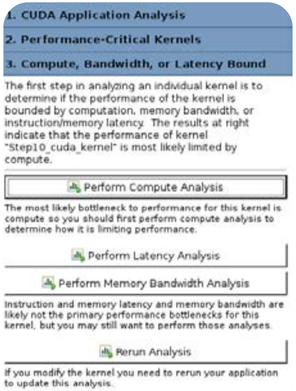
   %> nvprof –o profile.metrics --analysis-metrics ./a.out

NVIDIA’s Visual Profiler (NVVP)

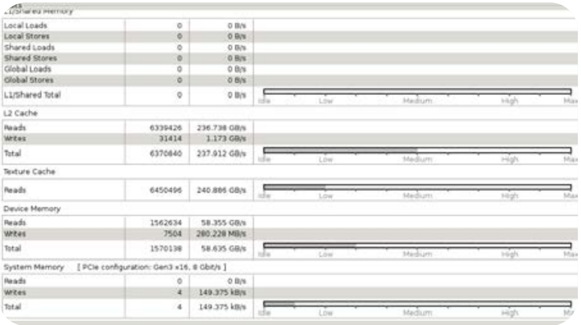
Timeline



Guided System



Analysis



L1/Shared Memory

Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	0	0 B/s
Shared Stores	0	0 B/s
Global Loads	0	0 B/s
Global Stores	0	0 B/s
L1/Shared Total	0	0 B/s

L2 Cache

Reads	6339426	236.738 GB/s
Writes	31414	1.173 GB/s
Total	6370840	237.912 GB/s

Texture Cache

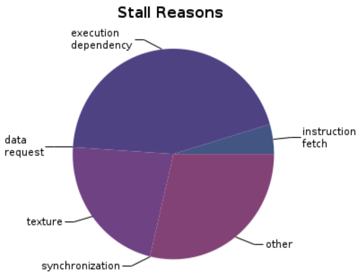
Reads	6450496	240.886 GB/s

Device Memory

Reads	1562634	58.355 GB/s
Writes	7504	280.228 MB/s
Total	1570138	58.635 GB/s

System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]

Reads	0	0 B/s
Writes	4	149.375 kB/s
Total	4	149.375 kB/s



Example 4: NVVP

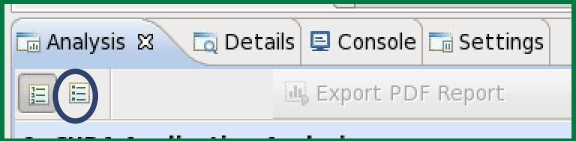
Instructions:

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:

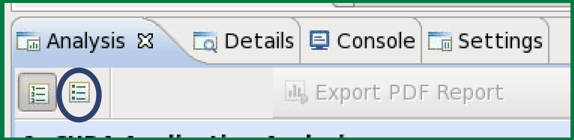
If kernel order is non-deterministic you can only load the timeline or the metrics but not both.

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



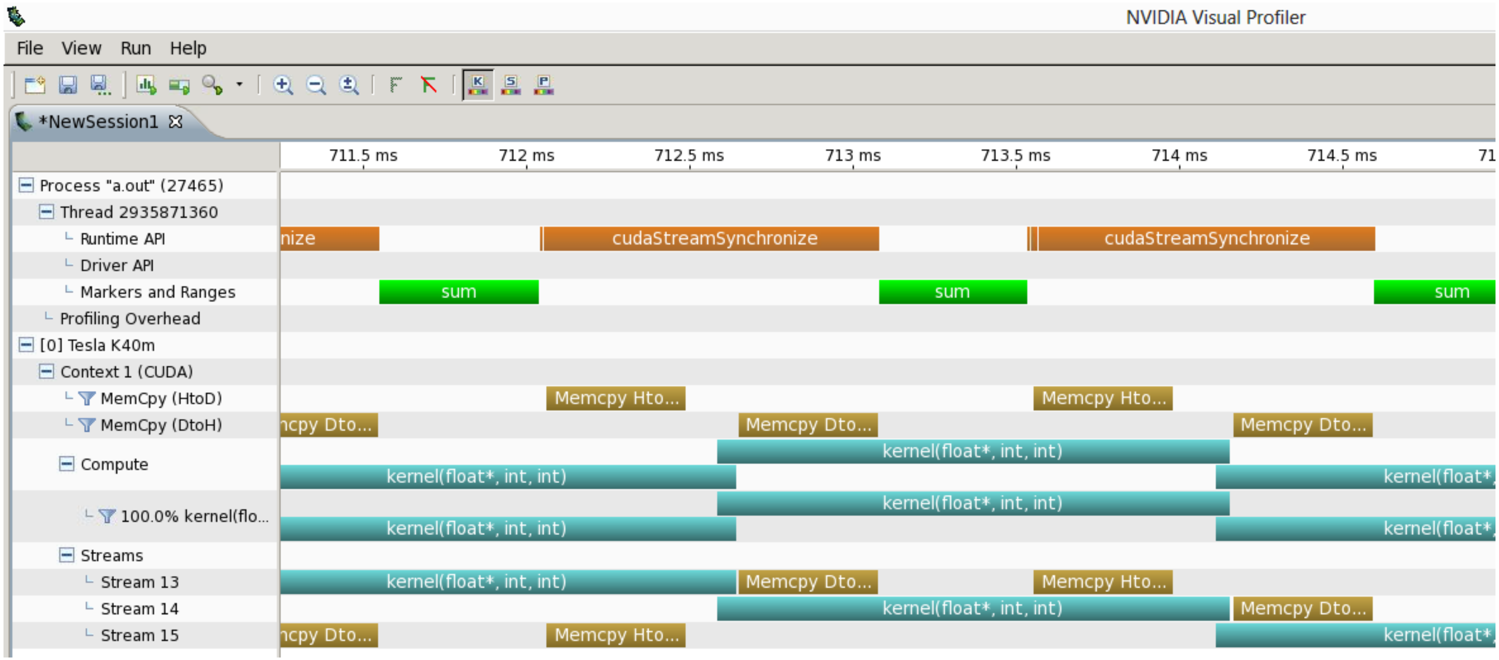
2. Click on a kernel
   Select Unguided Analysis
   Click Analyze All

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 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

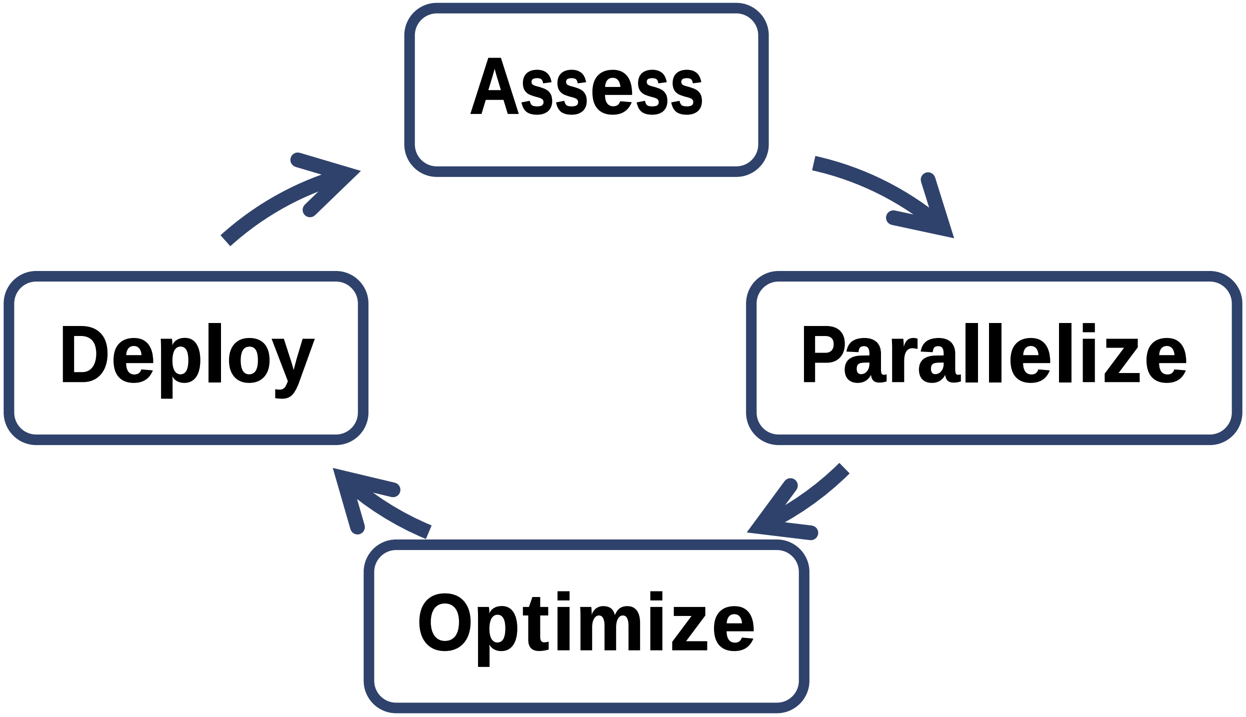
Instructions:

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

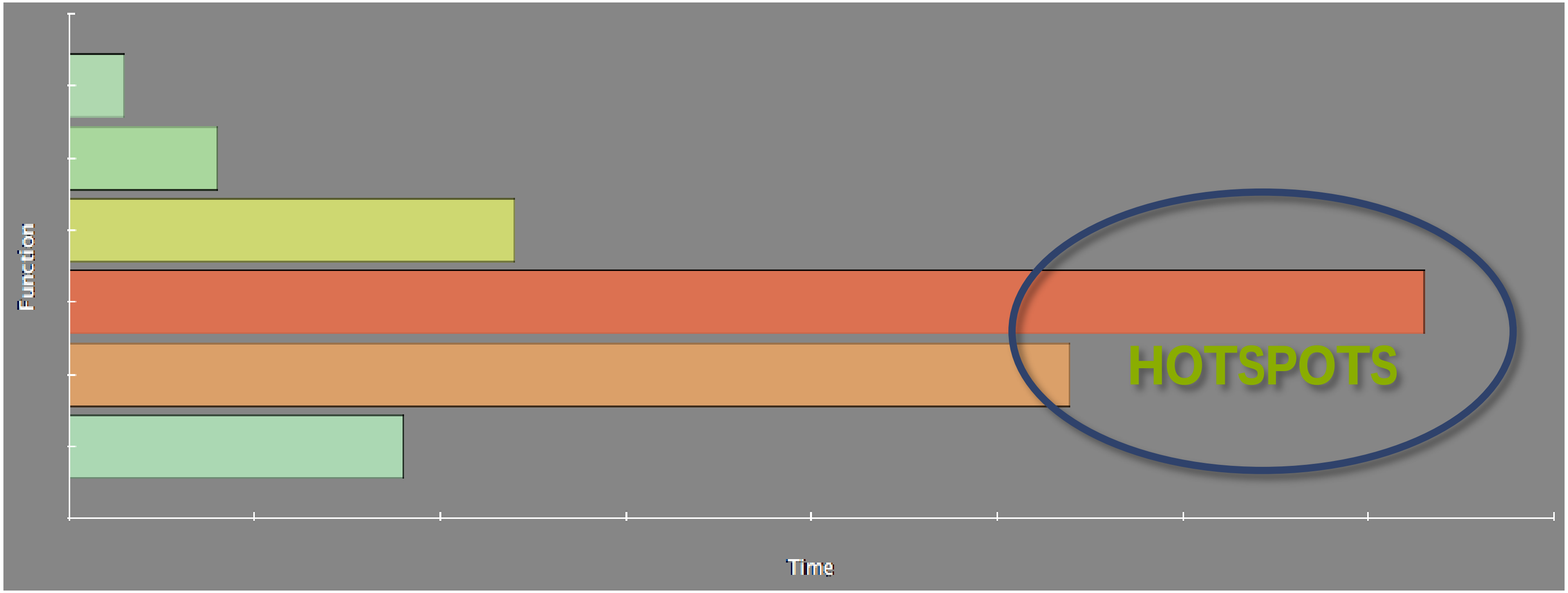
Profiler Summary

- Many profile tools are available
- NVIDIA Provided
  - NVPROF: Command Line
  - NVVP: Visual profiler
  - NSIGHT: IDE (Visual Studio and Eclipse)
- 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

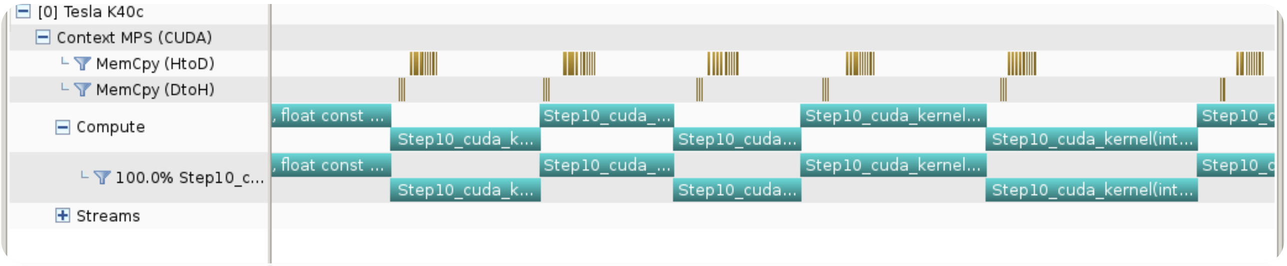
Libraries

Compiler Directives

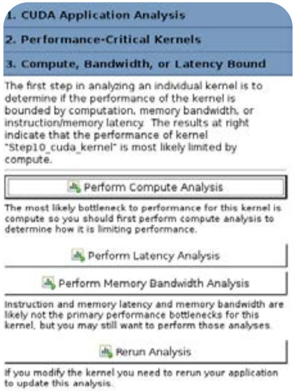
Programming Languages

Optimize

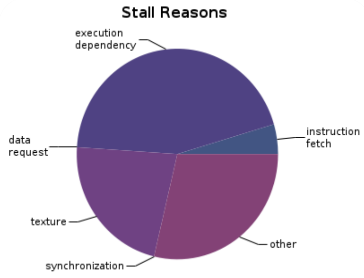
Timeline



Guided System



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	
L1/Shared Total	2490368	1,520.977 GB/s	Idle Low Medium

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 Efficien	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**

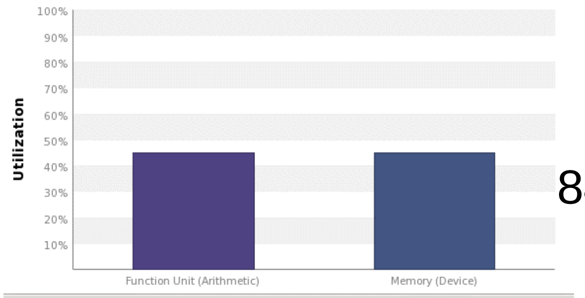
Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

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 ı
End	770.324 ı
Duration	256.714 |
Grid Size	[64,64,1
Block Size	[32,32,1
Registers/Thread	10
Shared Memory/Block	4.125 KiE
▽ Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	⚠ 50%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficien	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	Idle Low Medium
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	Idle Low Medium
Texture Cache			
Reads	0	0 B/s	Idle Low Medium
Device Memory			
Reads	524968	69.306 GB/s	
Writes	524289	69.217 GB/s	
Total	1049257	138.523 GB/s	Idle Low Medium

GPU Teaching Kit