



NVIDIA®

GPU Teaching Kit

Accelerated Computing



UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

## Lecture 17: Introduction to CUDA C

CUDA C vs. CUDA Libs vs. OpenACC  
Memory Allocation and Data Movement API Functions  
Data Parallelism and Threads  
Introduction to CUDA Toolkit



NVIDIA®

GPU Teaching Kit

Accelerated Computing



UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

## Introduction to CUDA C

CUDA C vs. CUDA Libs vs. OpenACC

# 3 Ways to Accelerate Applications

## Applications

Libraries

Easy to use  
Most Performance

Compiler  
Directives  
OpenACC/OpenMP

Easy to use  
Portable code

Programming  
Languages

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

MAGMA  
ICL-UT

C U S P

Numerical & Math  
RAND, Statistics



ArrayFire



Data Struct. & AI  
Sort, Scan, Zero Sum



Visual Processing  
Image & Video



5

NVIDIA

ILLINOIS

## Compiler Directives OpenACC and OpenMP: 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

6

NVIDIA

ILLINOIS

# 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

7



## GPU Programming Languages

Numerical analytics ➤

MATLAB Mathematica, LabVIEW

Fortran ➤

CUDA Fortran

C ➤

CUDA C

C++ ➤

CUDA C++

Python ➤

PyCUDA, Copperhead, Numba

F# ➤

Alea.cuBase

8



# CUDA - C

## Applications

Libraries

Compiler  
Directives  
OpenACC/OpenMP

Programming  
Languages

Easy to use  
Most Performance

Easy to use  
Portable code

**Most Performance**  
**Most Flexibility**



**NVIDIA**<sup>®</sup>

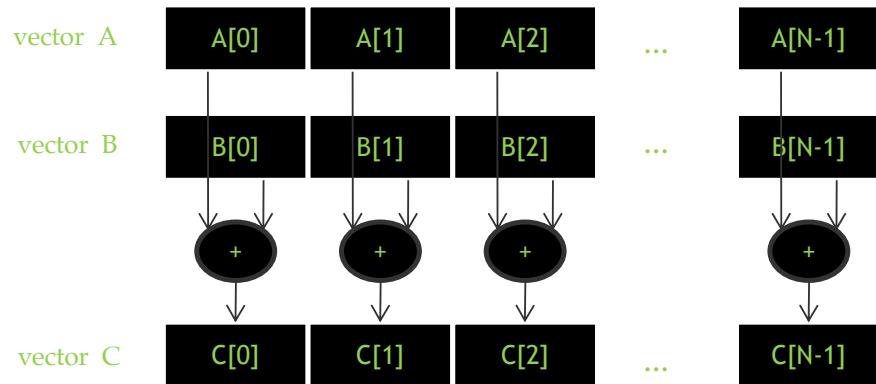
GPU Teaching Kit  
Accelerated Computing



## Introduction to CUDA C

Memory Allocation and Data Movement API Functions

# Data Parallelism - Vector Addition Example

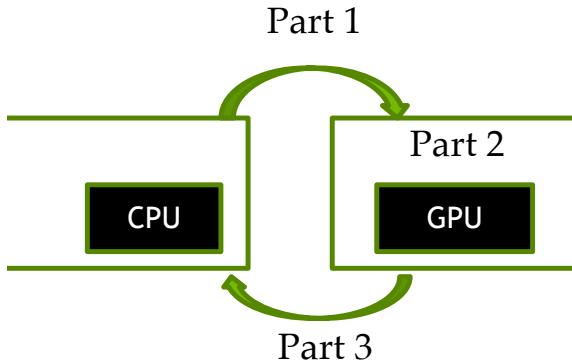


## Vector Addition – Traditional C Code

```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```

# Heterogeneous Computing vecAdd CUDA Host Code

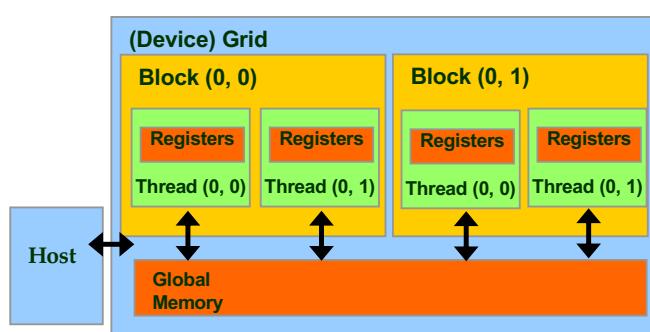


```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;
    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to device memory

    // Part 2
    // Kernel launch code – the device performs the actual vector addition

    // Part 3
    // copy C from the device memory
    // Free device vectors
}
```

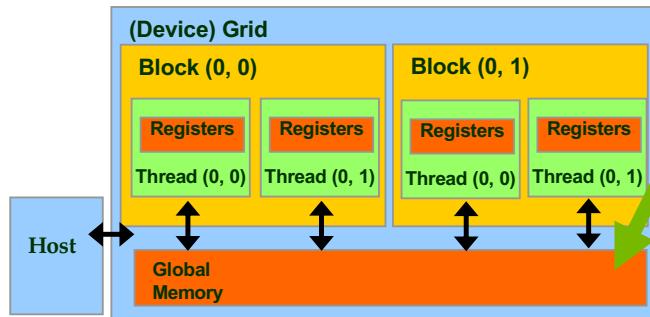
## Partial Overview of CUDA Memories



- Device code can:
  - R/W per-thread **registers**
  - R/W all-shared **global memory**
- Host code can
  - Transfer data to/from per grid **global memory**

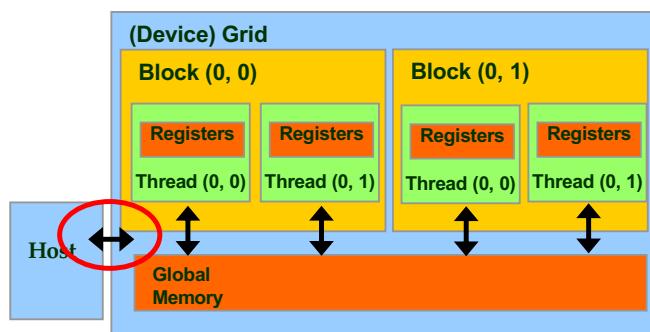
We will cover more memory types and more sophisticated memory models later.

# CUDA Device Memory Management API functions



- `cudaMalloc()`
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object in terms of bytes
- `cudaFree()`
  - Frees object from device global memory
  - One parameter
    - **Pointer** to freed object

# Host-Device Data Transfer API functions



- `cudaMemcpy()`
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
- Transfer to device is asynchronous

# Vector Addition Host Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code – to be shown later

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

17

NVIDIA

ILLINOIS

## In Practice, Check for API Errors in Host Code

```
cudaError_t err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__,
    __LINE__);
    exit(EXIT_FAILURE);
}
```

18

NVIDIA

ILLINOIS



NVIDIA®

GPU Teaching Kit

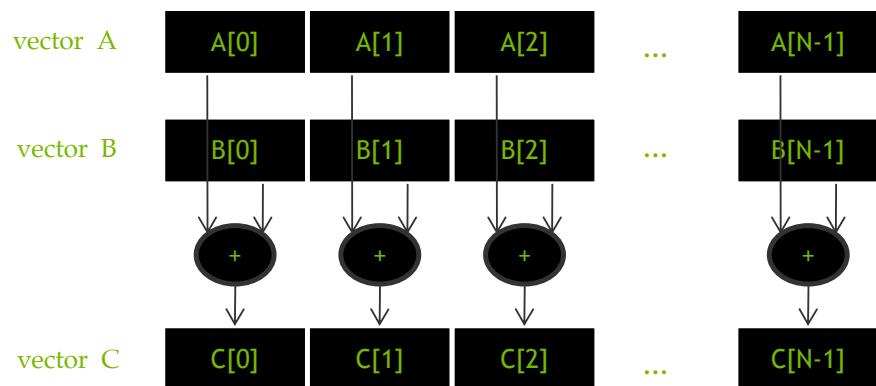


UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

# Introduction to CUDA C

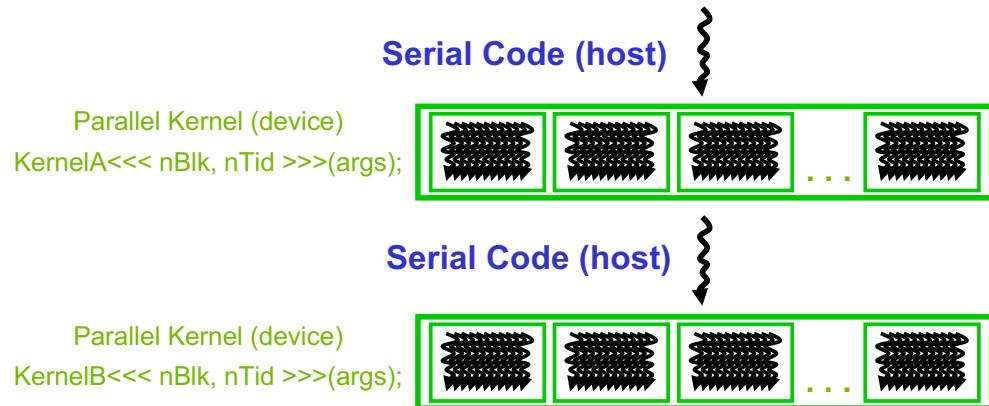
Threads and Kernel Functions

## Data Parallelism - Vector Addition Example

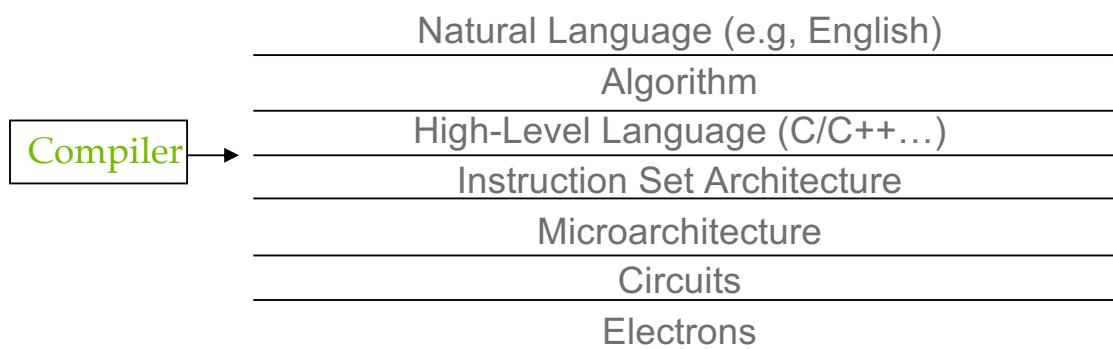


# CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
  - Serial parts in **host** C code
  - Parallel parts in **device** SPMD kernel code



## From Natural Language to Electrons

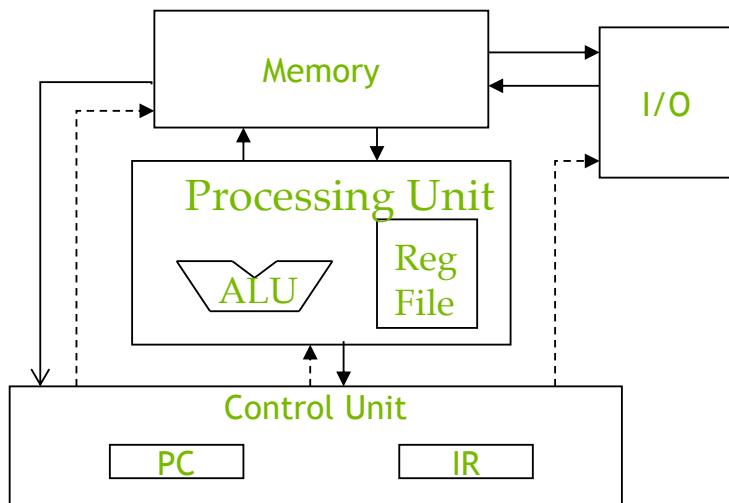


# A program at the ISA level

- A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.
  - Both CPUs and GPUs are designed based on (different) instruction sets
- Program instructions operate on data stored in memory and/or registers.

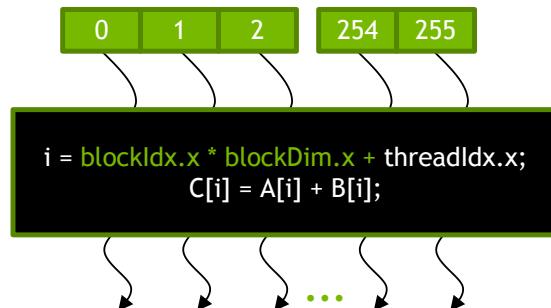
# A Thread as a Von-Neumann Processor

A thread is a “virtualized” or  
“abstracted”  
Von-Neumann Processor



# Arrays of Parallel Threads

- A CUDA kernel is executed by a **grid** (array) of threads
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions

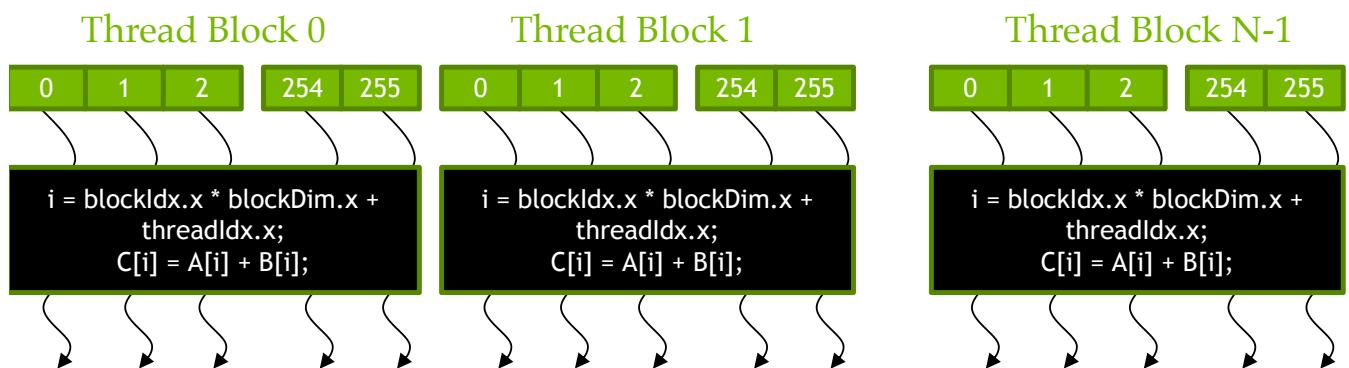


25

NVIDIA

ILLINOIS

## Thread Blocks: Scalable Cooperation



- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks do not interact

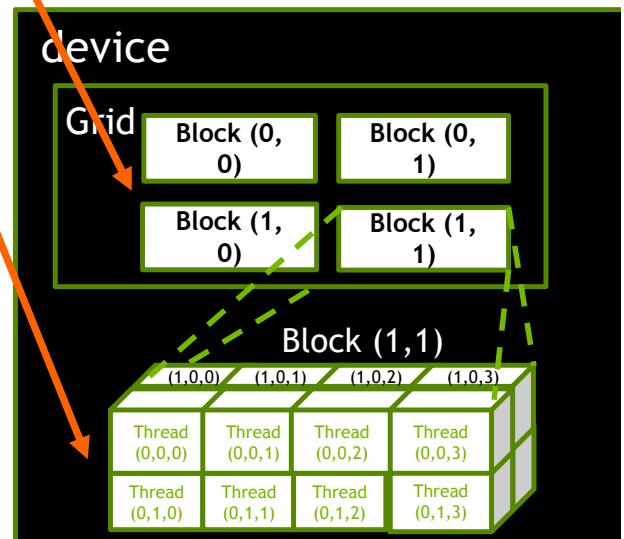
26

NVIDIA

ILLINOIS

# blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...



NVIDIA®

GPU Teaching Kit

Accelerated Computing



## Introduction to CUDA C

Introduction to the CUDA Toolkit

# 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



<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

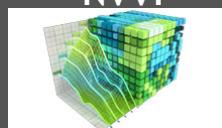
<http://docs.nvidia.com/cuda/cuda-gdb>

# Developer Tools - Profilers

NSIGHT



NVVP

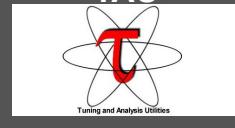


NVPROF

Time(ns)	Time	Calls	Avg	Min	Max	Name
49.888	866.0ms	58473	1.717ns	1.594ns	2.055ns	void th
100.000	1.000ms	1	1.000ns	1.000ns	1.000ns	main
25.378	448.05ns	235662	3.741ns	1.536ns	2.369ns	void th
1.450	1.450ns	1	1.450ns	1.450ns	1.450ns	__cudaThreadFence
17.078	256.00ns	1	1.488ns	1.488ns	1.488ns	__cudaLaunchKernel
2.988	51.879ns	200	259.09ns	246.97ns	261.81ns	kernelA
1.150	1.150ns	500	2.30ns	2.28ns	2.32ns	kernelB
0.938	16.159ns	200	89.99ns	71.84ns	98.75ns	kernelC
0.730	7.30ns	400	18.75ns	17.50ns	20.00ns	kernelD
0.699	12.079ns	200	60.37ns	59.08ns	62.36ns	kernelE
0.636	16.999ns	200	54.96ns	52.08ns	58.20ns	kernelF
0.572	11.329ns	200	56.65ns	54.02ns	59.28ns	kernelG
0.128	2.1340ns	1	2.1340ns	2.1340ns	2.1340ns	void th

NVIDIA Provided

TAU



VampirTrace



3rd Party

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

# NVPROF

## Command Line Profiler

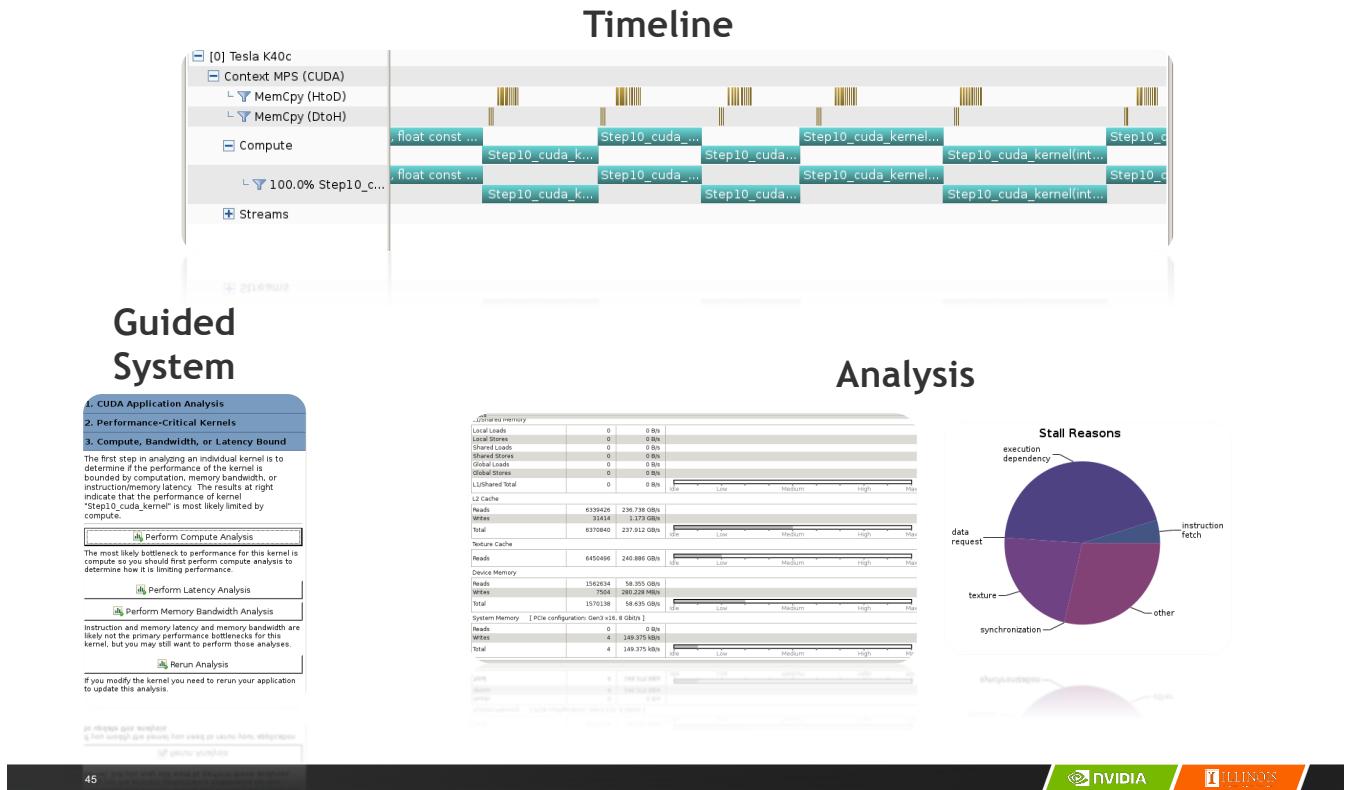
- Compute time in each kernel
- Compute memory transfer time
- Collect metrics and events
- Support complex process hierarchy's
- Collect profiles for NVIDIA Visual Profiler
- No need to recompile

## 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)



## 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

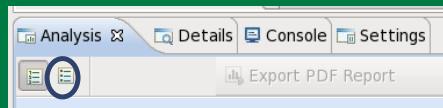
Control + mouse drag in timeline to zoom in

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.

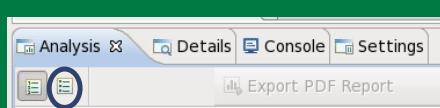
## Example 4: NVVP

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

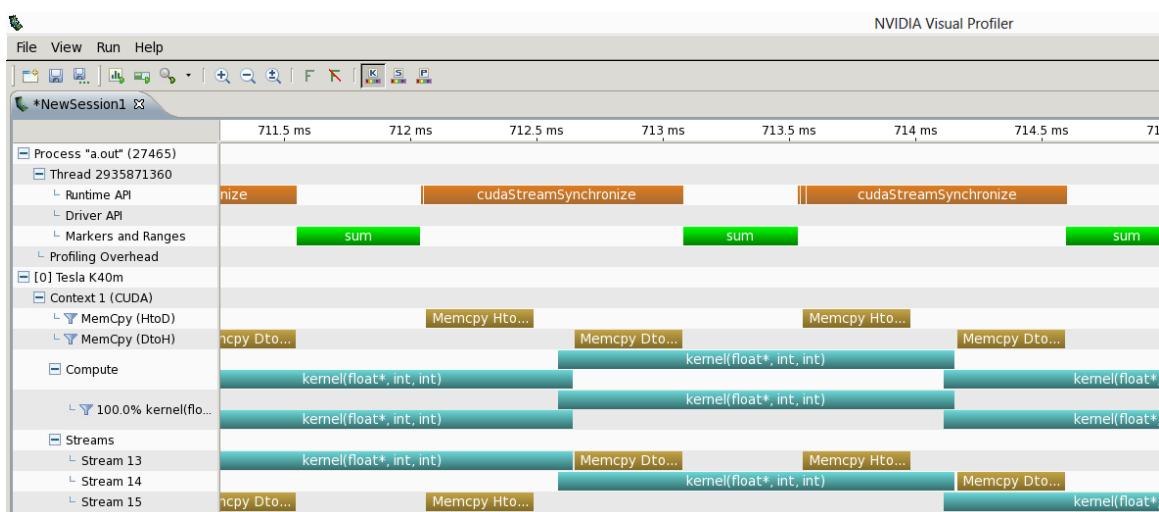
<http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/>

49

NVIDIA

ILLINOIS

## NVTX Profile



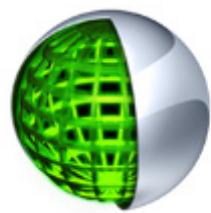
50

NVIDIA

ILLINOIS

# 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

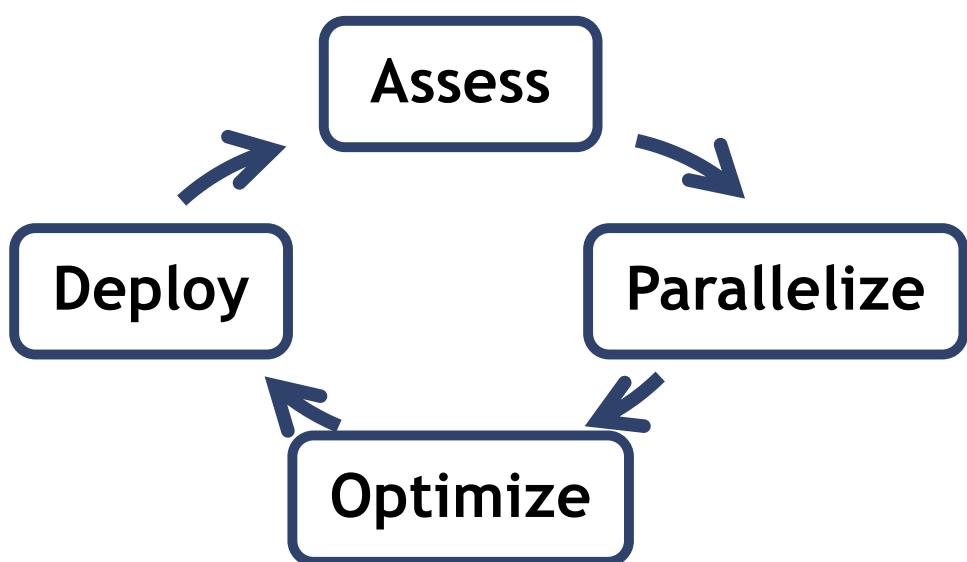
### Instructions:

1. Run nsight  
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

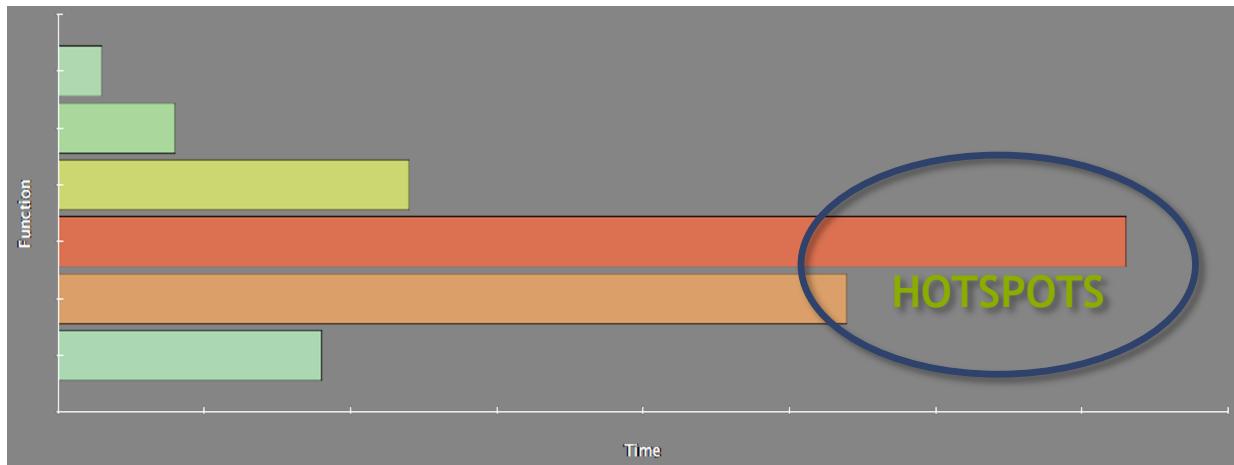
# Profiler Summary

- 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

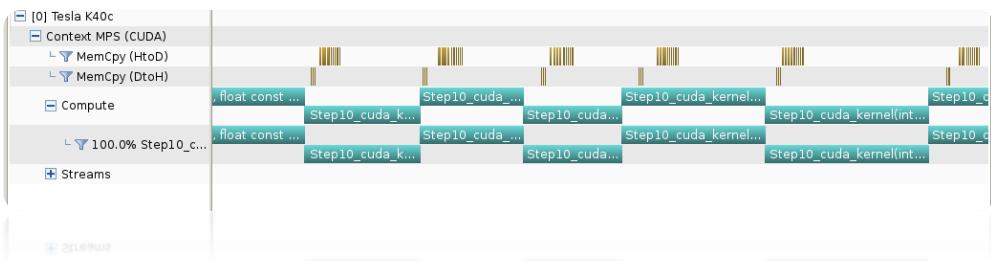
Libraries

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**

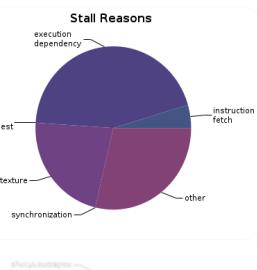
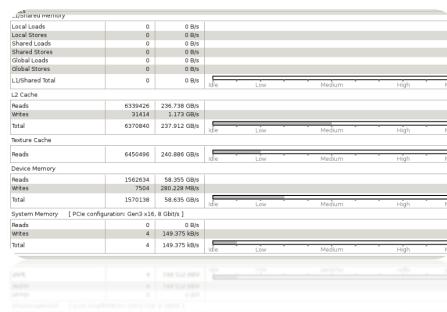
**Perform Memory Bandwidth Analysis**

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

**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

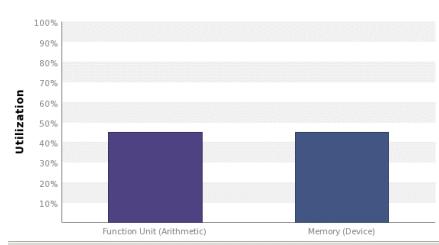
Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

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/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	770.0671
End	770.3241
Duration	256.7141
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
		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