

Introduction to CUDA Programming

Jian Tao

jtao@tamu.edu

Fall 2017 HPRC Short Course

09/28/2017



Relevant Short Course and Workshop

Intermediate CUDA Programming

https://hprc.tamu.edu/training/intermediate_cuda.html

2:30pm - 3:55pm, Tuesday, October 31 2017

Bring-Your-Own-Code Workshop

<https://coehpc.engr.tamu.edu/byoc/>

Offered regularly

GPU as an Accelerator



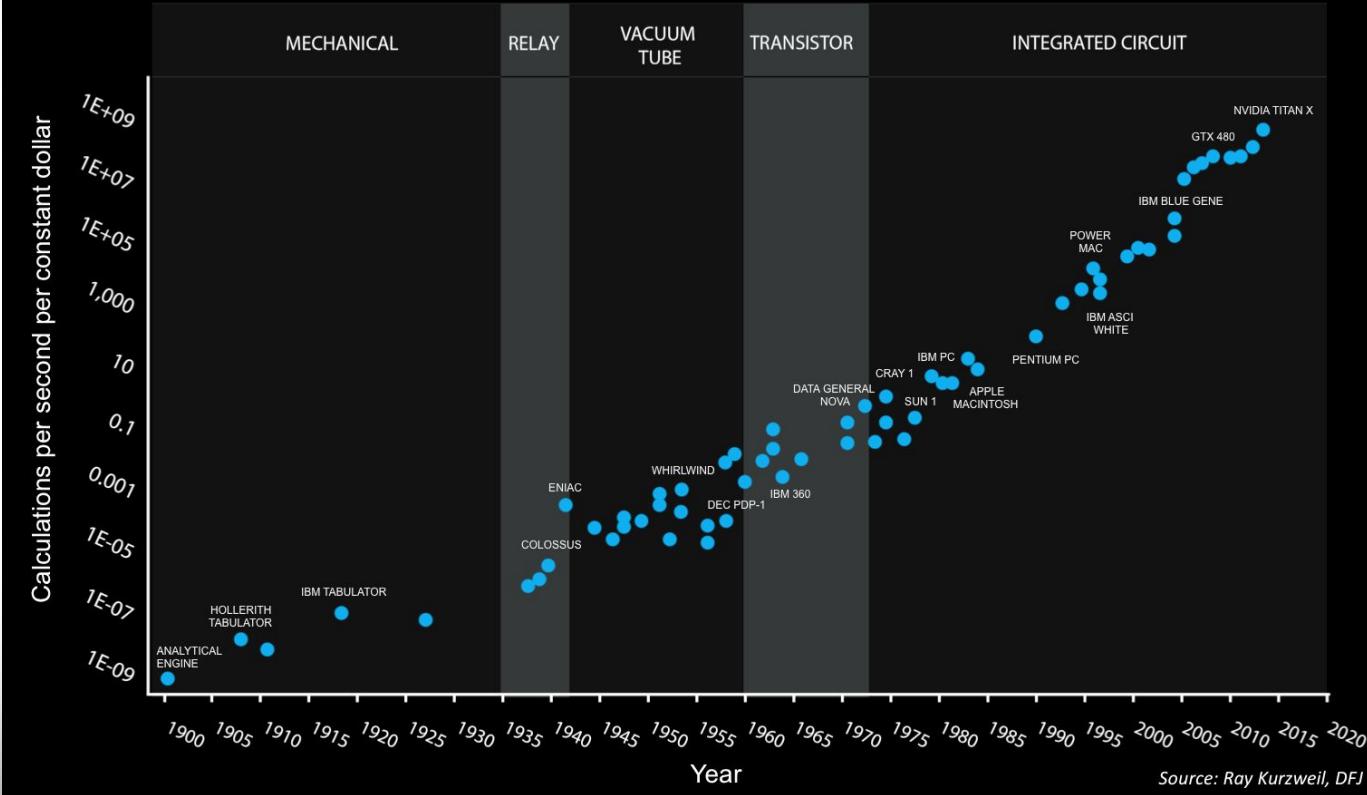
CPU



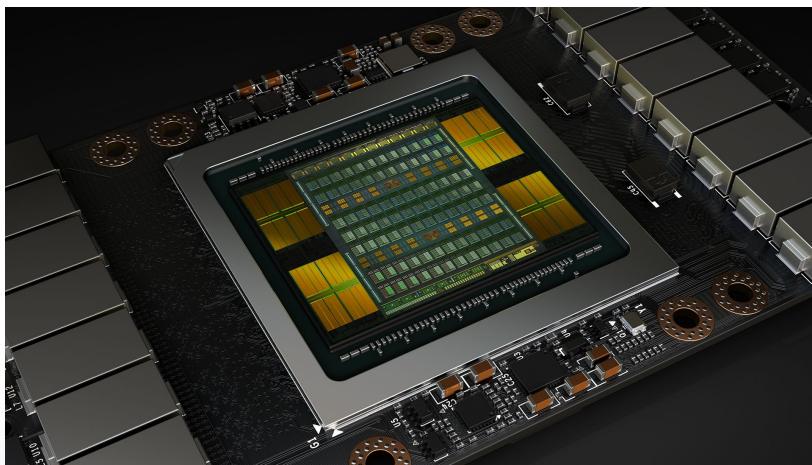
GPU Accelerator



120 Years of Moore's Law



NVIDIA Tesla V100 with 21.1 Billion Transistors



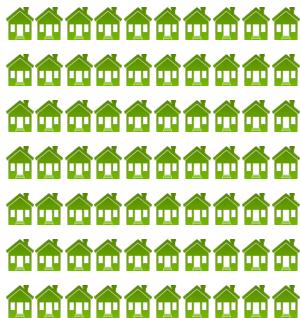
Why Computing Perf/Watt Matters?

2.3 PFlops



7.0
Megawatts

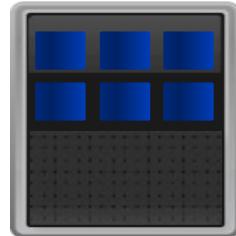
7000 homes



7.0
Megawatts

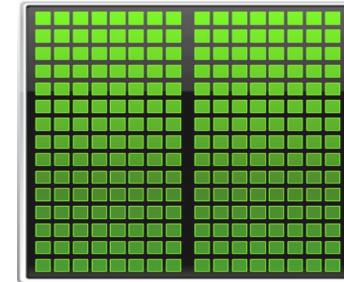
CPU

Optimized for
Serial Tasks



GPU Accelerator

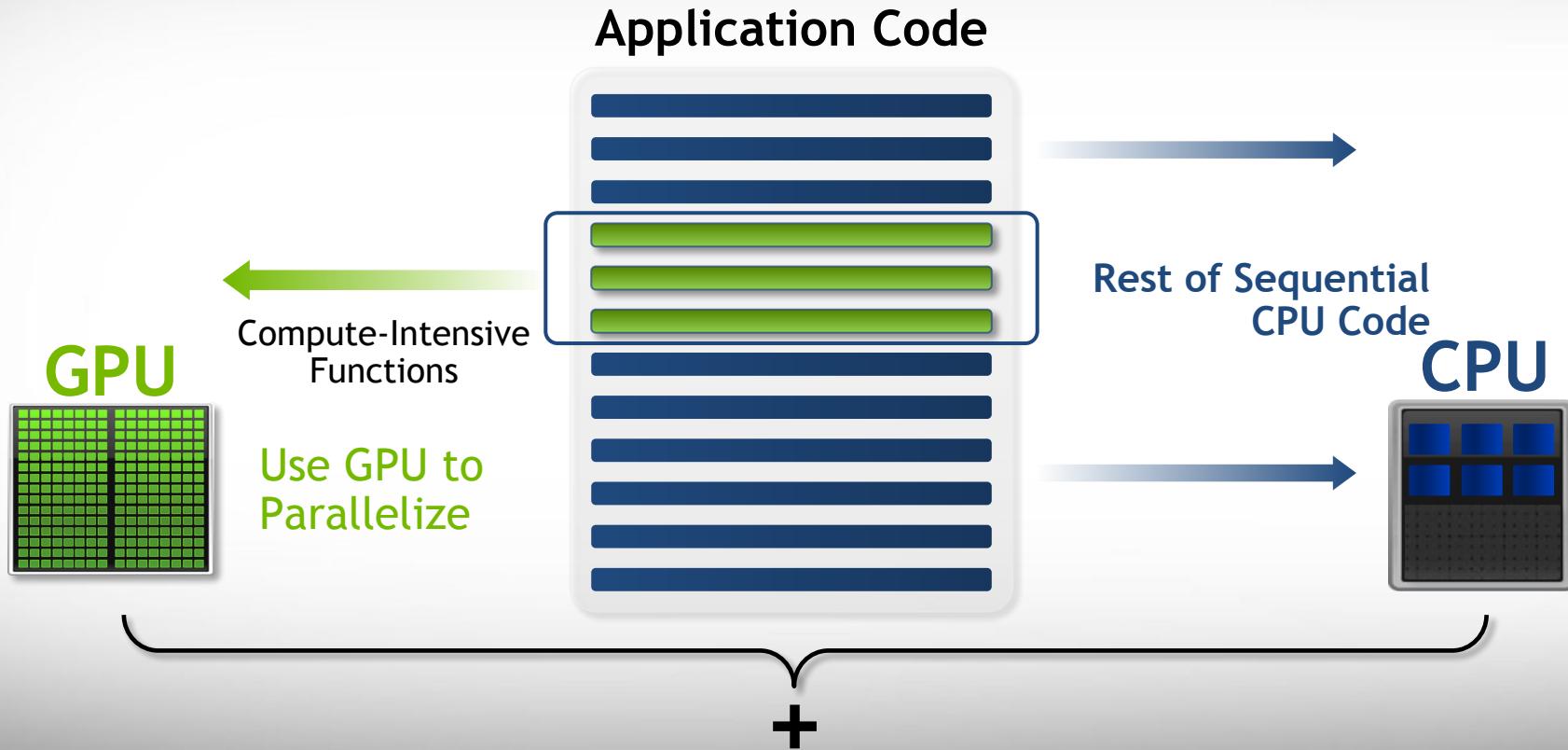
Optimized for Many
Parallel Tasks



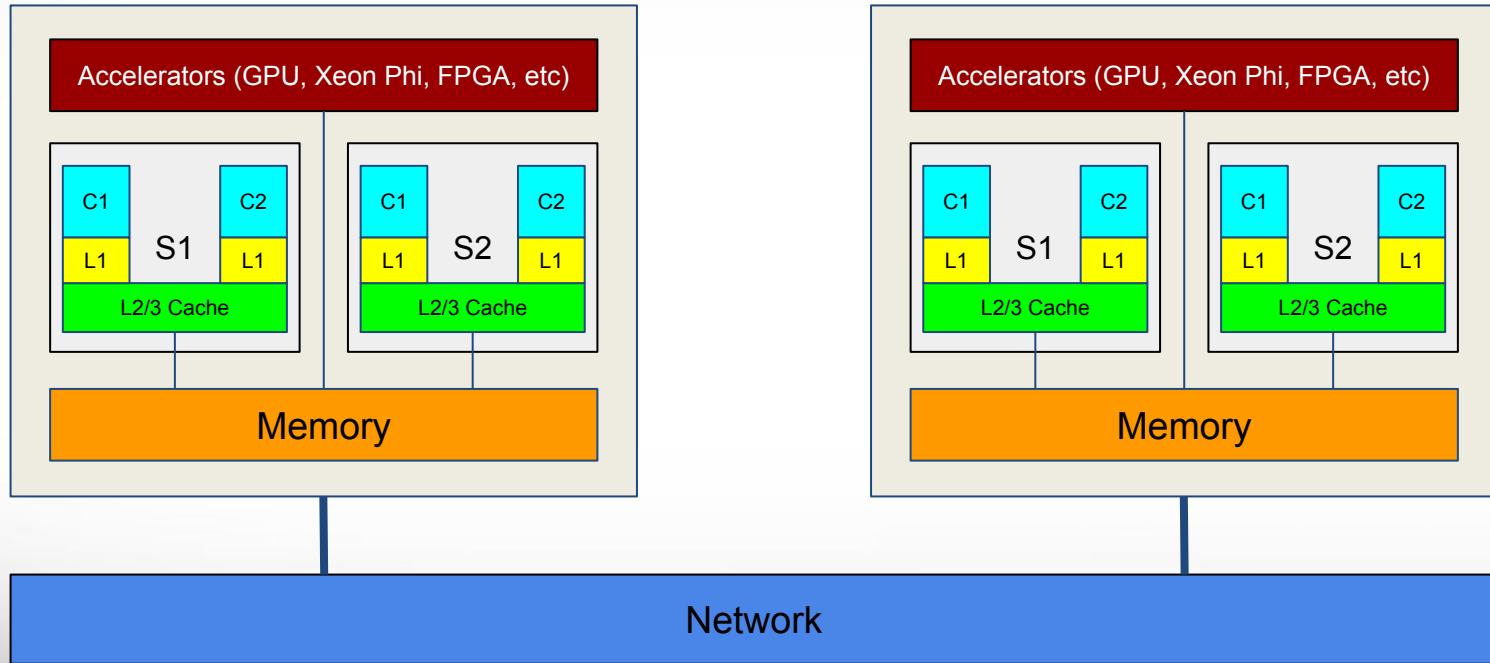
Traditional CPUs are
not economically feasible

GPU-accelerated computing
started a new era

Add GPUs: Accelerate Science Applications



HPC - Distributed Heterogeneous System



Programming Models: MPI + (CUDA, OpenCL, OpenMP, OpenACC, etc.)

Amdahl's Law



$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

CUDA Parallel Computing Platform

<https://developer.nvidia.com/cuda-toolkit>

Programming
Approaches

Libraries

“Drop-in” Acceleration

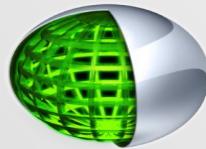
OpenACC
Directives

Easily Accelerate Apps

Programming
Languages

Maximum Flexibility

Development
Environment



Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

CUDA-GDB debugger
NVIDIA Visual Profiler

Open Compiler
Tool Chain



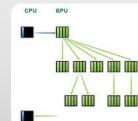
Enables compiling new languages to CUDA platform, and
CUDA languages to other architectures

Hardware
Capabilities

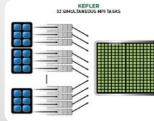
SMX



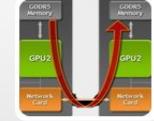
Dynamic Parallelism



HyperQ



GPUDirect



3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

Programming
Languages

“Drop-in”
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

Programming
Languages

“Drop-in”
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility

Libraries: Easy, High-Quality Acceleration

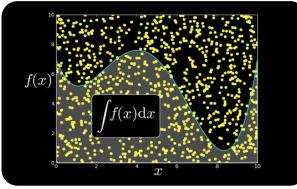
- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions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

Some GPU-accelerated Libraries

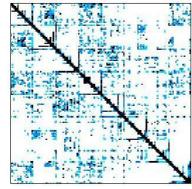
<https://developer.nvidia.com/gpu-accelerated-libraries>



NVIDIA cuBLAS



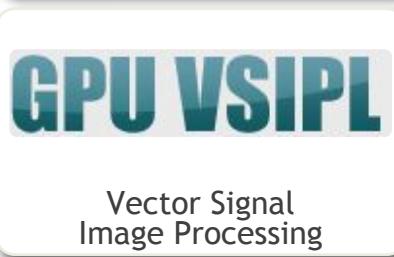
NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP



Vector Signal
Image Processing

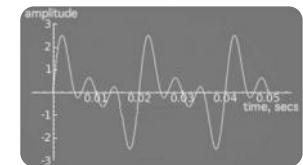


GPU Accelerated
Linear Algebra



ICL UT

Matrix Algebra on GPU
and Multicore
 open source
initiative



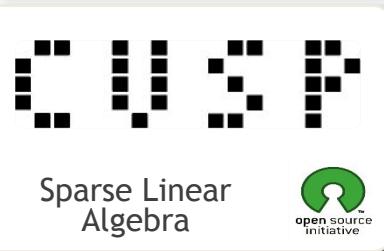
NVIDIA cuFFT



IMSL Library



ArrayFire Matrix
Computations



Sparse Linear
Algebra
 open source
initiative



C++ STL Features
for CUDA



CUDA-accelerated Application with Libraries

- **Step 1:** Substitute library calls with equivalent CUDA library calls

`saxpy (...)` ➤ `cublasSaxpy (...)`

- **Step 2:** Manage data locality

- with CUDA: `cudaMalloc()`, `cudaMemcpy()`, etc.
- with CUBLAS: `cublasAlloc()`, `cublasSetVector()`, etc.

- **Step 3:** Rebuild and link the CUDA-accelerated library

```
$nvcc myobj.o -l cublas
```

Explore the CUDA (Libraries) Ecosystem

- CUDA Tools and Ecosystem described in detail on NVIDIA Developer Zone.

NVIDIA ACCELERATED COMPUTING Downloads Training Ecosystem Forums

Tools & Ecosystem

Home > ComputeWorks

 Accelerated Solutions
GPUs are accelerating many applications across numerous industries.
[Learn more >](#)

 Numerical Analysis Tools
GPU acceleration for applications with high arithmetic density.
[Learn more >](#)

 GPU-Accelerated Libraries
Application accelerating can be as easy as calling a library function.
[Learn more >](#)

 Language and APIs
GPU acceleration can be accessed from most popular programming languages.
[Learn more >](#)

 Performance Analysis Tools
Find the best solutions for analyzing your application's performance profile.
[Learn more >](#)

 Debugging Solutions
Powerful tools can help debug complex parallel applications in intuitive ways.
[Learn more >](#)

 Key Technologies
Learn more about parallel computing technologies and architectures.
[Learn more >](#)

 Accelerated Web Services
Micro services with visual and intelligent capabilities using deep learning.
[Learn more >](#)

 Cluster Management
Managing your cluster and job scheduling can be simple and intuitive.
[Learn more >](#)

<https://developer.nvidia.com/tools-ecosystem>



Texas A&M University

m

High Performance Research Computing – <https://hprc.tamu.edu>

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

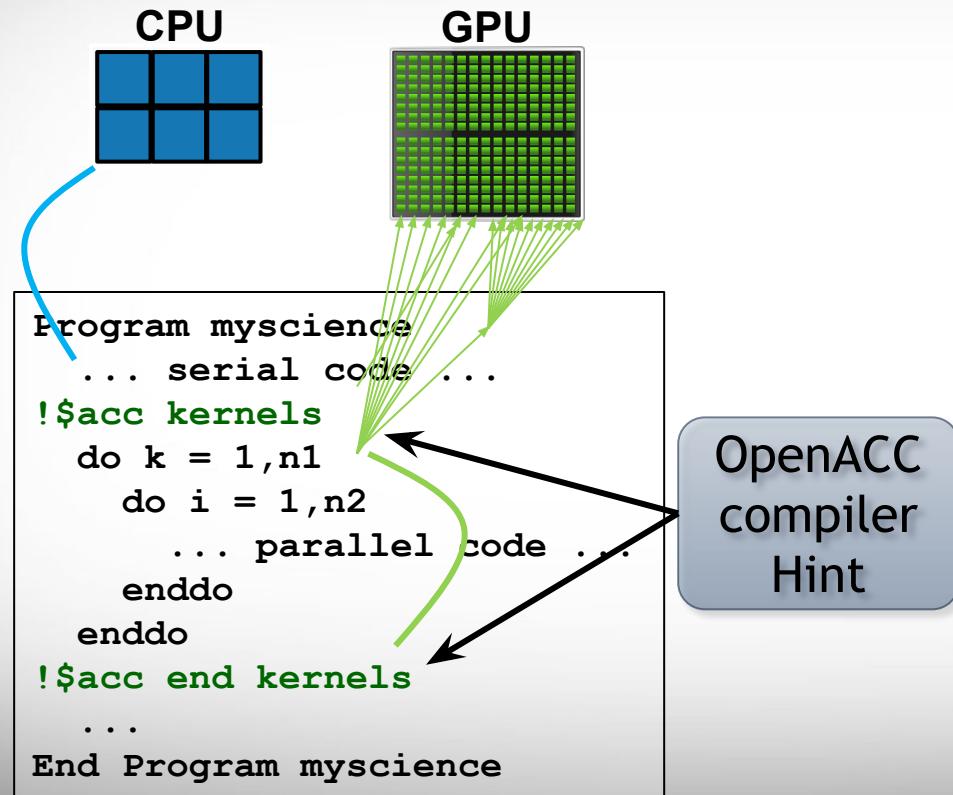
Programming
Languages

“Drop-in”
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility

OpenACC Directives



Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

OpenACC



The Standard for GPU Directives

- **Easy:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU

Directives: Easy & Powerful

Real-Time Object
Detection

Global Manufacturer of
Navigation Systems



5x in 40 Hours

Valuation of Stock Portfolios
using Monte Carlo

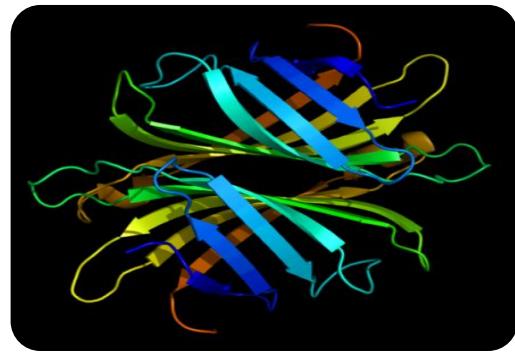
Global Technology Consulting
Company



2x in 4 Hours

Interaction of Solvents and
Biomolecules

University of Texas at San Antonio



5x in 8 Hours

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

Programming
Languages

“Drop-in”
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility

GPU Programming Languages

Numerical analytics ➤ MATLAB, Mathematica, LabVIEW

Fortran ➤ OpenACC, CUDA Fortran

C ➤ OpenACC, CUDA C, OpenCL

C++ ➤ Thrust, CUDA C++, OpenCL

Python ➤ PyCUDA, PyOpenCL, Copperhead

Java / F# ➤ JCuda / Alea GPU

Rapid Parallel C++ Development

- Resembles C++ STL
- High-level interface
 - Enhances developer productivity
 - Enables performance portability between GPUs and multicore CPUs
- Flexible
 - CUDA, OpenMP, and TBB backends
 - Extensible and customizable
 - Integrates with existing software
- Open source



```
// generate 32M random numbers on host
thrust::host_vector<int> h_vec(32 << 20);
thrust::generate(h_vec.begin(),
                 h_vec.end(),
                 rand);

// transfer data to device (GPU)
thrust::device_vector<int> d_vec = h_vec;

// sort data on device
thrust::sort(d_vec.begin(), d_vec.end());

// transfer data back to host
thrust::copy(d_vec.begin(),
             d_vec.end(),
             h_vec.begin());
```

<https://thrust.github.io/>

Learn More

These languages are supported on all CUDA-capable GPUs.

You might already have a CUDA-capable GPU in your laptop or desktop PC!

CUDA C/C++

<http://developer.nvidia.com/cuda-toolkit>

Thrust C++ Template Library

<http://developer.nvidia.com/thrust>

CUDA Fortran

<https://developer.nvidia.com/cuda-fortran>

PyCUDA (Python)

<https://developer.nvidia.com/pycuda>

Alea GPU

<http://www.aleagpu.com>

MATLAB

<http://www.mathworks.com/discovery/matlab-gpu.html>

Mathematica

<http://www.wolfram.com/mathematica/new-in-8/cuda-and-opencl-support/>

CUDA C/C++ BASICS



What is CUDA?

- CUDA Architecture
 - Used to mean “Compute Unified Device Architecture”
 - Expose GPU parallelism for general-purpose computing
 - Retain performance
- CUDA C/C++
 - Based on industry-standard C/C++
 - Small set of extensions to enable heterogeneous programming
 - Straightforward APIs to manage devices, memory etc.

A Brief History of CUDA

- Researchers used OpenGL APIs for general purpose computing on GPUs before CUDA.
- In 2007, NVIDIA released first generation of Tesla GPU for general computing together their proprietary CUDA development framework.
- Current stable version of CUDA is 8.0 (as of Sept. 2017).
- CUDA 9 Release Candidate is available.

Heterogeneous Computing

- Terminology:
 - *Host* The CPU and its memory (host memory)
 - *Device* The GPU and its memory (device memory)



Host



Device

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

_global_ void stencil_1d(int *in, int *out) {
    _shared_ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int index = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[index] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[index - RADIUS] = in[gindex - RADIUS];
        temp[index + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[index + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out; // host copies of a, b, c
    int _d_in, *_d_out; // device copies of a, c
    in = (int *)malloc(sizeof(int) * N * 2 * RADIUS);
    out = (int *)malloc(sizeof(int) * N * 2 * RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **)&_d_in, size);
    cudaMalloc((void **)&_d_out, size);

    // Copy to device
    cudaMemcpy(_d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(_d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d(<>N * BLOCK_SIZE, BLOCK_SIZE>>>(_d_in + RADIUS, _d_out + RADIUS));

    // Copy result back to host
    cudaMemcpy(out, _d_out, size, cudaMemcpyDeviceToHost);

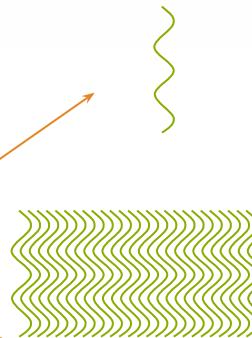
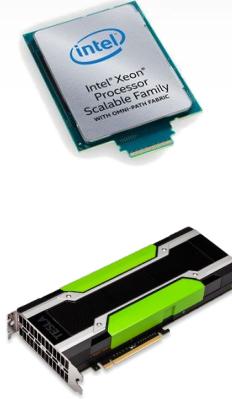
    // Cleanup
    free(in); free(out);
    cudaFree(_d_in); cudaFree(_d_out);
    return 0;
}
```

parallel function

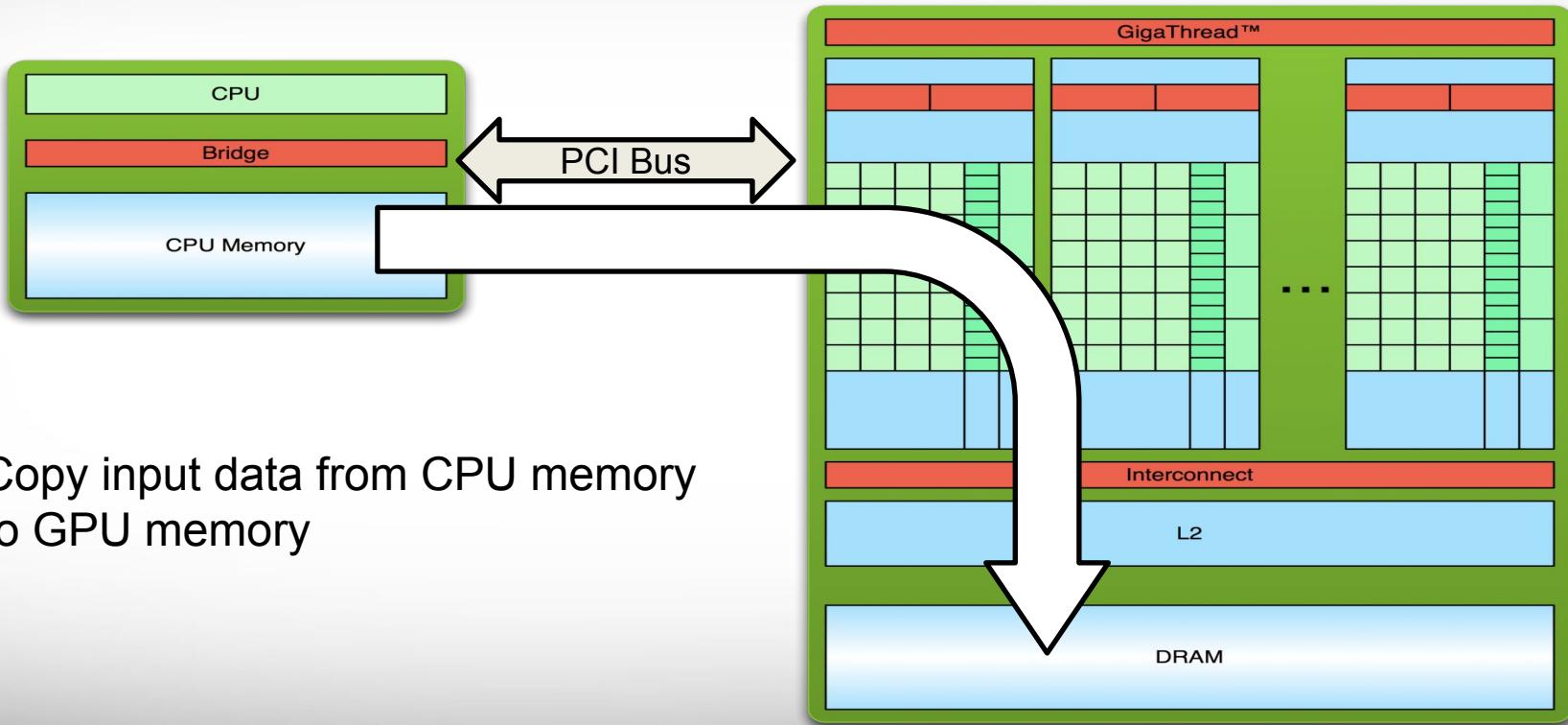
serial code

parallel code

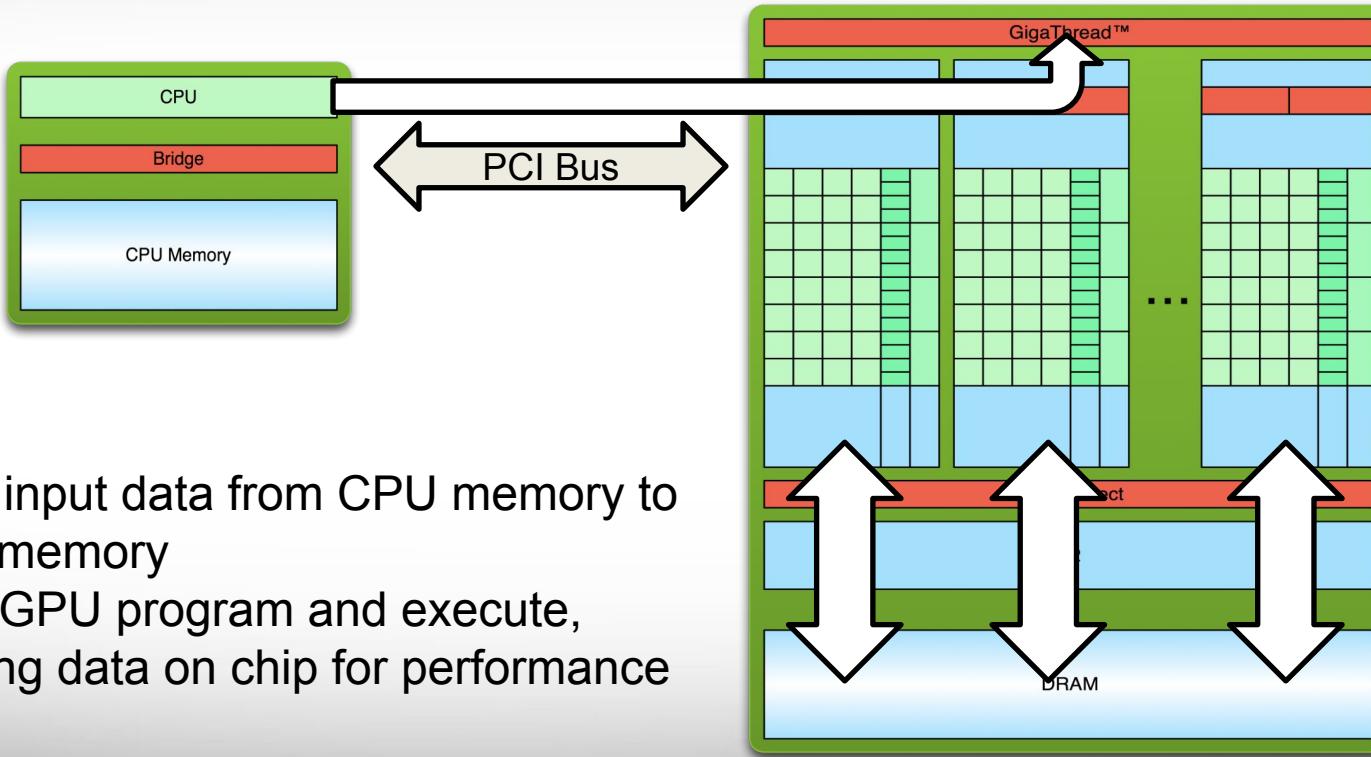
serial code



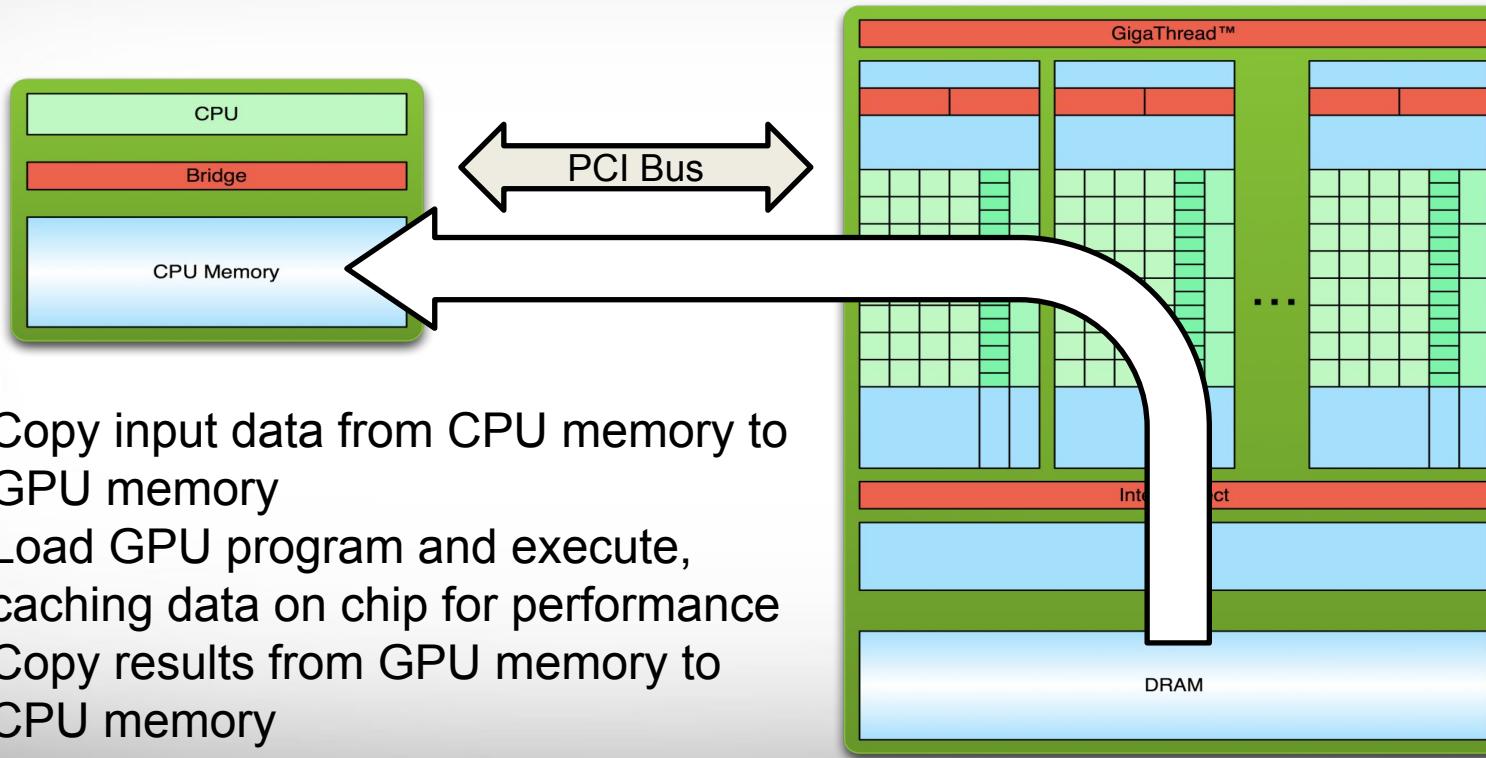
Simple Processing Flow



Simple Processing Flow



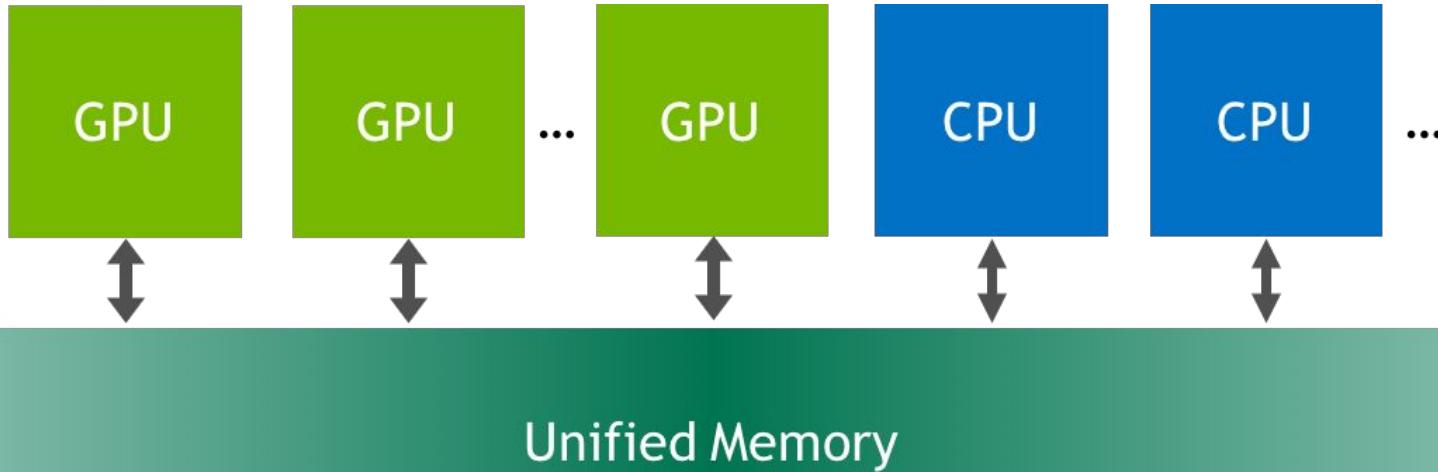
Simple Processing Flow



Unified Memory

Software: CUDA 6.0 in 2014

Hardware: Pascal GPU in 2016



Hello World!

```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

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

Hello World! with Device Code

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

- Two new syntactic elements...

Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
 - Runs on the device
 - Is called from host code
- nvcc separates source code into host and device components
 - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
 - Host functions (e.g. `main()`) processed by standard host compiler
 - `gcc`, `icc`, etc.

Hello World! with Device Code

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
 - Also called a “kernel launch”
 - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

Hello World! with Device Code

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

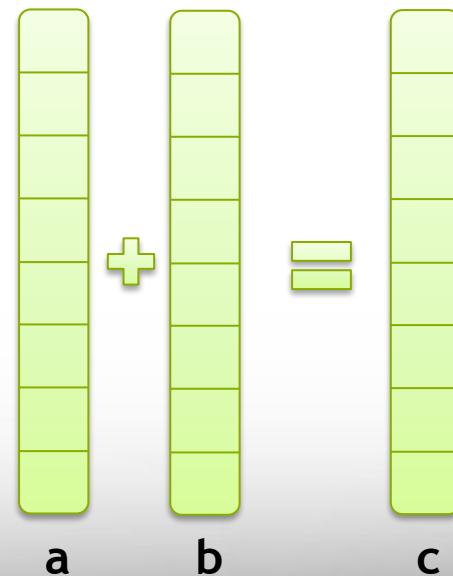
Output:

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

- `mykernel()` does nothing!

Parallel Programming in CUDA C/C++

- But wait... GPU computing is about massive parallelism!
- We need a more interesting example...
- We'll start by adding two integers and build up to vector addition



Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
 - `add()` will execute on the device
 - `add()` will be called from the host

Addition on the Device

- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU.

Memory Management

- Host and device memory are separate entities
 - *Device* pointers point to GPU memory
 - May be passed to/from host code
 - May *not* be dereferenced in host code
 - *Host* pointers point to CPU memory
 - May be passed to/from device code
 - May *not* be dereferenced in device code
- Simple CUDA API for handling device memory
 - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
 - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`



Addition on the Device: add()

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- Let's take a look at `main()`...

Addition on the Device: main()

```
int main(void) {
    int a, b, c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Setup input values
    a = 2;
    b = 7;
```

Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Moving to Parallel

- GPU computing is about massive parallelism
 - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();
```



```
add<<< N, 1 >>>();
```

- Instead of executing add () once, execute N times in parallel

Vector Addition on the Device

- With **add()** running in parallel we can do vector addition
- Terminology: each parallel invocation of **add()** is referred to as a **block**
 - The set of blocks is referred to as a **grid**
 - Each invocation can refer to its block index using **blockIdx.x**
- By using **blockIdx.x** to index into the array, each block handles a different index

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- On the device, each block can execute in parallel:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

Vector Addition on the Device: add()

- Returning to our parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- Let's take a look at `main()`...

Vector Addition on the Device: main()

```
#define N 512
int main(void) {
    int *a *b *c           // host copies of a, b, c
    int *d_a, *d_b, *d_c;   // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and set up input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Profiling with nvprof

```
$nvprof add_parallel
```

```
==28491== Profiling result:
```

Time(%)	Time	Calls	Avg	Min	Max	Name
43.45%	4.3520us	1	4.3520us	4.3520us	4.3520us	add(int*, int*, int*)
30.35%	3.0400us	2	1.5200us	1.3120us	1.7280us	[CUDA memcpy HtoD]
26.20%	2.6240us	1	2.6240us	2.6240us	2.6240us	[CUDA memcpy DtoH]

```
==28491== API calls:
```

Time(%)	Time	Calls	Avg	Min	Max	Name
99.34%	231.73ms	3	77.242ms	6.1990us	231.71ms	cudaMalloc
0.33%	766.63us	182	4.2120us	171ns	143.74us	cuDeviceGetAttribute
0.15%	357.72us	2	178.86us	173.06us	184.67us	cuDeviceTotalMem
0.08%	175.05us	3	58.351us	6.6470us	147.94us	cudaFree
0.03%	75.722us	1	75.722us	75.722us	75.722us	cudaLaunch
0.03%	74.091us	3	24.697us	10.865us	35.014us	cudaMemcpy
0.03%	65.073us	2	32.536us	30.391us	34.682us	cuDeviceGetName
0.00%	4.6390us	3	1.5460us	221ns	3.9590us	cudaSetupArgument
0.00%	4.4490us	3	1.4830us	434ns	3.3590us	cuDeviceGetCount
0.00%	2.7070us	6	451ns	196ns	777ns	cuDeviceGet
0.00%	1.9940us	1	1.9940us	1.9940us	1.9940us	cudaConfigureCall

Review (1 of 2)

- Difference between *host* and *device*
 - *Host* CPU
 - *Device* GPU
- Using `__global__` to declare a function as device code
 - Executes on the device
 - Called from the host
- Passing parameters from host code to a device function

Review (2 of 2)

- Basic device memory management
 - `cudaMalloc()`
 - `cudaMemcpy()`
 - `cudaFree()`
- Launching parallel kernels
 - Launch **N** copies of `add()` with `add<<<N,1>>>(...)`.
 - Use `blockIdx.x` to access block index.
 - Use `nvprof` for collecting & viewing profiling data.

More Resources

You can learn more about the details at

- CUDA Programming Guide (docs.nvidia.com/cuda)
- CUDA Zone – tools, training, etc. (developer.nvidia.com/cuda-zone)
- Download CUDA Toolkit & SDK (www.nvidia.com/getcuda)
- Nsight IDE (Eclipse or Visual Studio) (www.nvidia.com/nsight)

Intermediate CUDA Programming Short Course

- GPU memory management and unified memory
- Parallel kernels in CUDA C
- Parallel communication and synchronization
- Running a CUDA code on Ada
- Profiling and performance evaluation

Acknowledgements

- Educational materials from NVIDIA via its Academic Programs.
- Supports from Texas A&M Engineering Experiment Station (TEES) and High Performance Research Computing (HPRC).

Running CUDA Code on Ada

<https://github.com/jtao/coehpc>

```
# load CUDA module
$ml CUDA/8.0.44-GCC-5.4.0-2.26

# copy sample code to your scratch space
$cd $SCRATCH
$cp -r /scratch/training/CUDA .

# compile CUDA code
$cd CUDA
$nvcc hello_world_host.cu -o hello_world

# edit job script & submit your first GPU job
$bsub < cuda_run.sh
```

