GP-GPU PROGRAMMING WITH CUDA

Larry Brown Ph.D.
larryb@nvidia.com
AGENDA

1. NVIDIA Intro
2. GP-GPU Intro
3. Accelerating Code with CUDA
4. Optimizations
5. Key Applications
NVIDIA - INVENTOR OF THE GPU

NVIDIA Invented the GPU in 1999, with over 1 Billion shipped to date.

Initially a dedicated a graphics processor for PCs, the GPU’s computational power and energy efficiency led to it quickly being adopted for professional and scientific simulation and visualization.

In 2007, NVIDIA launched the CUDA® programming platform, and opened up the general purpose parallel processing capabilities of the GPU.
USA - TWO FLAGSHIP SUPERCOMPUTERS

SUMMIT
150-300 PFLOPS Peak Performance
IBM POWER9 CPU + NVIDIA Volta GPU
NVLink High Speed Interconnect
>40 TFLOPS per Node, >3,400 Nodes
2017

SIERRA
> 100 PFLOPS Peak Performance

Powered by the NVIDIA GPU
END OF TRADITIONAL FREQUENCY SCALING

The Free Lunch is Over: A Fundamental Turn towards Concurrency in Software
Herb Sutter

The breakdown is in Dennard scaling ...
Moores law marches on.
GP-GPU Programming
HETEROGENEOUS COMPUTING
10X PERFORMANCE 5X ENERGY EFFICIENCY

CPU
Optimized for Serial Tasks

GPU Accelerator
Optimized for Parallel Tasks
LOW LATENCY OR HIGH THROUGHPUT?

**CPU**
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

**GPU**
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation
GPGPU ACCELERATION - THE BASIC IDEA

Application Code

Compute-Intensive Functions

Rest of Sequential CPU Code

Accelerator

CPU
GPGPU IMPACT - REALTIME LINE-OF-SIGHT

Task
Line-of-sight at 60 fps

GPUs
100x faster than CPUs

Result
Instant analysis for mission preparation

http://www.brainshark.com/nvidia/demo-video-lucidlightspeed
http://www.luciad.com/
GPGPU IMPACT - FIND OBJECTS IN IMAGES

Task: Automatic objects search in multiple video/image formats

GPUs: 20x faster than CPUs delivering 20,000 FPS

Result: Live and forensic search with ranking and alerts

One hour of video searched in 5 seconds

http://www.nervvetech.com/
GPGPU IMPACT - GRAPH ANALYTICS

- Comms & Social networks
- Cyber pattern recognition
- Shortest path finding

PageRank : 19x Speedup

Time per iteration (s)

Intel Xeon E5-2690 v2

1 GPU vs 60 Nodes
- 280x vs optimized Spark
- 1440x vs Spark
- 3420x vs Hadoop
GPGPU IMPACT - DEEP LEARNING

Image Recognition Challenge

1.2M training images • 1000 object categories

Hosted by

GPU Entries

Classification Error Rates
HOW DO I USE GPUS?
CUDA PARALLEL DEVELOPMENT PLATFORM

developer.nvidia.com

Programming Approaches

- Libraries
  - “Drop-in” Acceleration

- 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

- LLVM Compiler Infrastructure
  - Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

Hardware Capabilities

- SMX
- Dynamic Parallelism
- HyperQ
- GPUDirect

“Drop-in” Acceleration
Easily Accelerate Apps
Maximum Flexibility
Nsight IDE
CUDA-GDB debugger
NVIDIA Visual Profiler
Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

developer.nvidia.com
3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries
“Drop-in” Acceleration

OpenACC Directives
Easily Accelerate Applications

Programming Languages
Maximum Flexibility
ACCELERATING YOUR APPLICATIONS

Simple & Portable Parallel Software Development

Applications

Libraries
“Drop-in” Acceleration

Directives
Annotate code with compiler hints

Languages
Modern language features (unified memory, for_each, lambda)
GPU Accelerated Libraries

- NVIDIA cuBLAS
- NVIDIA cuRAND
- NVIDIA cuSPARSE
- NVIDIA NPP
- GPU VSIPL: Vector Signal Image Processing
- CULA tools: GPU Accelerated Linear Algebra
- MAGMA: Matrix Algebra on GPU and Multicore
- NVIDIA cuFFT
- Rogue Wave Software: IMSL Library
- ArrayFire Matrix Computations
- CUSP: Sparse Linear Algebra
- Thrust: C++ STL Features for CUDA
“DROP-IN” LIBRARY EXAMPLE: NVBLAS

Automatic Speedup for “R” application

> LD_PRELOAD=/usr/local/cuda/lib64/libnvblas.so R

> A <- matrix(rnorm(4096*4096), nrow=4096, ncol=4096)
> B <- matrix(rnorm(4096*4096), nrow=4096, ncol=4096)
> system.time(C <- A %*% B)

user  system elapsed
0.348  0.142   0.289

Matrix-Matrix Multiplication in R

Use in any app that uses standard BLAS3

R, Octave, Scilab, etc.

NO CODE CHANGE REQUIRED
5X-10X SPEEDUP USING NVIDIA LIBRARIES

BLAS | LAPACK | SPARSE | FFT | Math | Deep Learning | Image Processing

- Filter
- Statistics
- Geometry Transformations
- JPEG
- Morphological
- Linear Transform
- Color Processing
- Color Conversion
- Threshold And Compar
- Alpha Composition
- Logical
- Arithmetic
- Data Exchange And Initialization

Graph 1: Sparse Matrix x Dense Vector (SpMV)

Graph 2: Benchmark Results

Graph 3: Performance Comparison

Graph 4: GPU Performance Over Time
WHAT IS A DROP-IN LIBRARY?

1. Uses standard or near-standard interface

2. No code change required
   • Recompile might not even be required

3. Data lives in CPU memory at the beginning and end of library routines.
   • Completely invisible to the program
ACCELERATING YOUR APPLICATIONS

Simple & Portable Parallel Software Development

Applications

- Libraries
  - “Drop-in” Acceleration

- Directives
  - Annotate code with compiler hints

- Languages
  - Modern language features (unified memory, for_each, lambda)
OPENACC DIRECTIVES

Manage
Data
Movement

#pragma acc data copyin(a,b) copyout(c)
{
...

#pragma acc parallel
{
#pragma acc loop gang vector
    for (i = 0; i < n; ++i) {
        z[i] = x[i] + y[i];
        ...
    }
}
...

Initiate
Parallel
Execution

Optimize
Loop
Mappings

• Incremental
• Single source
• Interoperable
• Simple compiler hints
• CPU, GPU, MIC
• Compiler parallelizes code
OpenACC
Simple | Powerful | Portable

Fueling the Next Wave of Scientific Discoveries in HPC

main()
{
    <serial code>
    #pragma acc kernels
    // automatically runs on GPU
    {
        <parallel code>
    }
}

University of Illinois
PowerGrid - MRI Reconstruction

70x Speed-Up
2 Days of Effort

RIKEN Japan
NICAM - Climate Modeling

7-8x Speed-Up
5% of Code Modified

8000+
Developers
using OpenACC

http://www.cray.com/sites/default/files/resources/OpenACC_213462.12_OpenACC_Cosmo_CS_FNL.pdf
http://www.openacc.org/content/experiences-porting-molecular-dynamics-code-gpus-cray-xk7
OpenACC makes GPU computing approachable for domain scientists. Initial OpenACC implementation required only minor effort, and more importantly, no modifications of our existing CPU implementation.
INTRODUCING THE NEW OPENACC TOOLKIT

Free Toolkit Offers Simple & Powerful Path to Accelerated Computing

- **PGI Compiler**
  Free OpenACC compiler for academia

- **NVProf Profiler**
  Easily find where to add compiler directives

- **GPU Wizard**
  Identify which GPU libraries can jumpstart code

- **Code Samples**
  Learn from examples of real-world algorithms

- **Documentation**
  Quick start guide, Best practices, Forums

http://developer.nvidia.com/openacc
OPENACC DELIVERS PERFORMANCE PORTABILITY


Speedup vs 1x CPU Core

Intel OpenMP on 2x Haswell CPUs (32 Cores) - SPEC® Estimate*
PGI OpenACC on 2x Haswell CPUs (32 Cores)
PGI OpenACC on Tesla K80

Supermicro SYS-2028GR-TFT, Intel Xeon E5-2698 v3, 32 cores, NVIDIA Tesla K80, 256GB of System Memory
PGI 15.7 Beta OpenACC Multicore and K80 results from SPEC ACCEL™ measured June 2015.
PGI 15.0.90 OpenMP results use Cloverleaf reference application and SPEC OMP2012 using workloads from SPEC ACCEL™
SPEC and the benchmark names SPEC ACCEL™ and SPEC OMP® are registered trademarks of the Standard Performance Evaluation Corporation.

Intel OpenMP 15.7 Beta
PGI OpenACC 15.7 Beta

www.spec.org/accel
www.spec.org/omp2012
uk-mac.github.io/CloverLeaf/
https://mantevo.org/download/
ACCELERATING YOUR APPLICATIONS

Simple & Portable Parallel Software Development

Applications

Libraries

“Drop-in” Acceleration

Directives

Annotate code with compiler hints

Languages

Modern language features (unified memory, for_each, lambda)
VISION: MAINSTREAM PARALLEL PROGRAMMING

- Enable more programmers to write parallel software
- Give programmers the choice of language to use
- Embrace and evolve standards in key languages
COMPILE PYTHON FOR PARALLEL ARCHITECTURES

Anaconda Accelerate from Continuum Analytics

- NumbaPro array-oriented compiler for Python & NumPy
- Compile for CPUs or GPUs (uses LLVM + NVIDIA Compiler SDK)

Fast Development + Fast Execution: Ideal Combination

Free Academic License

http://continuum.io
Nsight Editor
- Automated CPU to GPU code refactoring
- Semantic highlighting
- Integrated code samples & documentation
- Cross-compilation for Linux ARM and POWER targets

Nsight Debugger
- Simultaneously debug CPU and GPU code
- Inspect variables across CUDA threads
- Use breakpoints & single-step debugging
- Integrated CUDA memory checker

Nsight Profiler
- Quickly identifies performance issues
- Guided expert analysis
- Source line correlation
- PLANNED: Unified CPU and GPU profiling
C++ PARALLEL ALGORITHMS LIBRARY

std::vector<int> vec = ...

// previous standard sequential loop
std::for_each(vec.begin(), vec.end(), f);

// explicitly sequential loop
std::for_each(std::seq, vec.begin(), vec.end(), f);

// permitting parallel execution
std::for_each(std::par, vec.begin(), vec.end(), f);

* Complete set of parallel primitives: for_each, sort, reduce, scan, etc.

* ISO C++ committee voted unanimously to accept as official tech. specification working draft

Prototype: https://github.com/n3554/n3554
### CUDA C/C++

#### Standard C Code

```c
void saxpy_serial(int n,
    float a,
    float *x,
    float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

#### Parallel C Code

```c
__global__
void saxpy_parallel(int n,
    float a,
    float *x,
    float *y)
{
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);
```

UPDATING LEGACY CODE

1. Libraries
2. Directives
3. CUDA C
Accelerating Code with CUDA
CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- std::syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
HELLO WORLD!

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
HETEROGENEOUS COMPUTING

Terminology:

- **Host**: The CPU and its memory (host memory)
- **Device**: The GPU and its memory (device memory)
# HETEROGENEOUS COMPUTING

```cpp
#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 lindex = threadIdx.x + RADIUS;

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

    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + 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, b, c

    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in  = (int*)malloc(size);
    fill_ints(in, N + 2*RADIUS);
    out = (int*)malloc(size);
    fill_ints(out, 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;
}
```

---

**device code**

**parallel function**

**serial function**

**serial code**

**host code**

**device function call**

**serial code**
1. Copy input data from CPU memory to GPU memory
SIMPLE PROCESSING FLOW

1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it
1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it
3. Copy results from GPU memory to CPU memory
```c
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

- `nvcc` separates source code into host and device components

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

```c
__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
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

```c
__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, somewhat anticlimactic!
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
A simple kernel to add two integers

```c
__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
Note that we use pointers for the variables

```c
__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()`
Returning to our `add()` kernel

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

Let’s take a look at `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;
REVIEW

- 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
RUNNING IN PARALLEL

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
MOVING TO PARALLEL

GPU computing is about massive parallelism

So how do we run code in parallel on the device?

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

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

This first index, is called the BlockIdx...you will see why soon.
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`

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

By using `blockIdx.x` to index into the array, each block handles a different element of the array
VECTOR ADDITION ON THE DEVICE

```c
__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**
  
  ```
  ```

- **Block 3**
  
  ```
  ```
VECTOR ADDITION ON THE DEVICE: ADD()

Returning to our parallelized add() kernel

```c
__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()...
#define N 512

```c
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 setup 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()**

```c
// Copy inputs to device
cudamempy(d_a, a, size, cudaMemcpyHostToDevice);
cudamempy(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
cudamempy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudafree(d_a); cudafree(d_b); cudafree(d_c);
return 0;
```
REVIEW

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
INTRODUCING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
CUDA THREADS

- Terminology: a block can be split into parallel threads

- Let’s change add() to use parallel threads instead of parallel blocks

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

- We use threadIdx.x instead of blockIdx.x

- Need to make one change in main()...
VECTOR ADDITION USING THREADS: 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 setup input values
    a = (int *)malloc(size); random ints(a, N);
    b = (int *)malloc(size); random ints(b, N);
    c = (int *)malloc(size);
VECTOR ADDITION USING THREADS: MAIN()

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

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(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;
```
COMBINING THREADS AND BLOCKS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
We’ve seen parallel vector addition using:
- Several blocks with one thread each
- One block with several threads

Let’s adapt vector addition to use both blocks and threads

Why? We’ll come to that...

First let’s discuss data indexing...
INDEXING ARRAYS WITH BLOCKS AND THREADS

- No longer as simple as using `blockIdx.x` and `threadIdx.x`.

- Consider indexing an array with one element per thread (8 threads/block).

With $M$ threads per block, a unique index for each thread is given by:

```c
int index = blockIdx.x * M + threadIdx.x;
```
INDEXING ARRAYS: EXAMPLE

Which thread will operate on the red element?

```
int index = blockIdx.x * M + threadIdx.x;
= 2 * 8 + 5;
= 21;
```
VECTOR ADDITION WITH BLOCKS AND THREADS

- Use the built-in variable blockDim.x for threads per block

  ```c
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  ```

- Combined version of `add()` to use parallel threads and parallel blocks

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

- What changes need to be made in `main()`?
#define N (2048*2048)
#define THREADS_PER_BLOCK 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 setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
ADDATION WITH BLOCKS AND THREADS:

MAIN()

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

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

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

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
HANDLING ARBITRARY VECTOR SIZES

- Typical problems are not even multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

  ```c
  __global__ void add(int *a, int *b, int *c, int n) {
      int index = threadIdx.x + blockIdx.x * blockDim.x;
      if (index < n)
          c[index] = a[index] + b[index];
  }
  ```

- Update the kernel launch:

  ```c
  add<<<(N/M + 1),M>>>(d_a, d_b, d_c, N);
  ```
WHY BOTHER WITH THREADS?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

- Unlike parallel blocks, threads have mechanisms to efficiently:
  - Communicate
  - Synchronize

- To look closer, we need a new example...
## COOPERATING THREADS

### CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- `_syncthreads()`
- Asynchronous operation
- Handling errors
- Managing devices
1D STENCIL

Consider applying a 1D stencil to a 1D array of elements

- Each output element is the sum of input elements within a radius

If radius is 3, then each output element is the sum of 7 input elements:
IMPLEMENTING WITHIN A BLOCK

- Each thread processes one output element
  - `blockDim.x` elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times
Terminology: within a block, threads share data via shared memory

Extremely fast on-chip memory
- By opposition to device memory, referred to as global memory
- Like a user-managed cache

Declare using __shared__, allocated per block

Data is not visible to threads in other blocks
IMPLEMENTING WITH SHARED MEMORY

- Cache data in shared memory
  - Read \((\text{blockDim}.x + 2 \times \text{radius})\) input elements from global memory to shared memory
  - Compute \(\text{blockDim}.x\) output elements
  - Write \(\text{blockDim}.x\) output elements to global memory

- Each block needs a halo of \(\text{radius}\) elements at each boundary

```plaintext
in
  +-------+-------+
  |  blue |  gray |
  +-------+-------+
  halo on left

out
  +-------+-------+
  |  gray |  blue |
  +-------+-------+
  halo on right
```

\(\text{blockDim}.x\) output elements
__global__ void stencil_1d(int *in, int *out) {

    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

    int gindex = threadIdx.x + blockIdx.x * blockDim.x;

    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];

    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
}
// Apply the stencil

int result = 0;

for (int offset = -RADIUS; offset <= RADIUS; offset++)
    result += temp[lindex + offset];

// Store the result

out[gindex] = result;
DATA RACE!

The stencil example will not work...

Suppose thread 15 reads the halo before thread 0 has fetched it...

temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
SYNCTHREADS()

void __syncthreads();

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block


STENCIL KERNEL

```c
__global__ void stencil_1d(int *in, int *out) {

    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

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

// Store the result
out[gindex] = result;
Launching parallel threads

- Launch $N$ blocks with $M$ threads per block with kernel $<<<N,M>>>(...)$;
- Use `blockIdx.x` to access block index within grid
- Use `threadIdx.x` to access thread index within block

Assign elements to threads:

```c
int index = blockIdx.x * blockDim.x + threadIdx.x;
```
Use __shared__ to declare a variable/array in shared memory

- Data is shared between threads in a block
- Not visible to threads in other blocks

Use __syncthreads() as a barrier

- Use to prevent data hazards
Optimizing CUDA Kernels
1. NVIDIA’s Visual Profiler (NVVP)
2. NVPROF
3. Nsight Eclipse Edition
NVIDIA’S VISUAL PROFILER

Timeline

Guided System

Analysis

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_cude_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 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.

Perform Latency Analysis

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

Command line profiler

Collect profiles for NVVP

%> nvprof --analysis-metrics -o profile.out ./HACCmk

Collect for MPI processes

%> mpirun -np 2 nvprof --analysis-metrics -o profile.%p.out ./HACCmk

Collect profiles for complex process hierarchies

--profile-child-processes, --profile-all-processes

Collect key events and metrics

%> nvprof --metrics flops_sp ./HACCmk

--query-metrics --query-events
NVIDIA® NSIGHT™ ECLIPSE EDITION

**Editor**
- Automated CPU to GPU code refactoring
- Semantic highlighting of CUDA code
- Code completion & inline help
- Integrated CUDA samples

**Debugger**
- Simultaneously debug of CPU and GPU
- Inspect variables across CUDA threads
- Use breakpoints & single-step debugging

**Profiler**
- Quickly identifies performance issues
- Automated analysis
- Source line correlation

Available for Linux and Mac OS
BEFORE OPTIMIZING YOUR KERNELS

- Always use NVVP to determine if the kernel is the limiter
- Kernels may not always be the limiter

- Memory Allocation
- Data Transfer
- Synchronization
- Host
KEY KERNEL OPTIMIZATIONS

Latency
- Expressing Concurrency
- Instruction Level Parallelism

Bandwidth
- Coalesced Memory Access
- Texture Memory
- Shared memory

Compute
- Reducing Divergence
void transpose(float in[][[]], float out[][[]], int N) {
    for(int j=0; j < N; j++)
        for(int i=0; i < N; i++)
            out[j][i] = in[i][j];
}

- Commonly used in applications
  - BLAS and FFT
- Stresses memory systems
  - Strided reads or writes
2D TO 1D INDEXING

```c
void transpose(float in[], float out[], int N)
{
    for(int j=0; j < N; j++)
        for(int i=0; i < N; i++)
            out[j*N+i] = in[i*N+j];
}
```

- This indexing is often used in numerical codes
- We will use this indexing during this presentation
void transpose(float in[], float out[], int N) {
    #pragma omp parallel for
    for(int j=0; j < N; j++)
        #pragma omp parallel for
        for(int i=0; i < N; i++)
            out[j*N+i] = in[i*N+j];
}

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU+OMP</td>
<td>4.9 GB/s</td>
</tr>
</tbody>
</table>

2X Intel Xeon E5-2650 (16 threads)
Parallelize outer loop
void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
    int i, j;
    for (i=0; i<rows; i++)
        for (j=0; j<cols; j++)
            out[ j * rows + i ] = in [ i * cols + j ];
}

CPU KERNEL
___global___ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
    int i, j;

    i = blockIdx.x * blockDim.x + threadIdx.x;

    for ( j=0; j<cols; j++)
        out [ j * rows + i ] = in [ i * cols + j ];
}
RESULTS

- Initial implementation slightly slower than dual sockets
- Peak 288 GB/s
- Low percent of peak
  - Why?

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Throughput</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU+OMP</td>
<td>13.8 GB/s</td>
<td>--</td>
</tr>
<tr>
<td>CUDA-1D</td>
<td>9.6 GB/s</td>
<td>0.7x</td>
</tr>
</tbody>
</table>

GPU: K40
INITIAL PROFILE

- Always look at occupancy first!
- Each block is scheduled on an SM
  - There are 15 SMs on K40
  - Only 4 blocks!
- Bottleneck
  - Grid size
  - Most of the GPU is idle
- Solution
  - Express more parallelism
KEY KERNEL OPTIMIZATIONS

Latency
- Expressing Concurrency
- Instruction Level Parallelism

Bandwidth
- Coalesced Memory Access
- Texture Memory
- Shared memory

Compute
- Reducing Divergence
OPTIMIZATION: EXPRESS MORE PARALLELISM

- The CPU version parallelizes over rows and columns.
- Let's do the same on the GPU.
  - Replace columns loop with an index calculation.
  - Change launch configuration to 2D.
    - blockSize = 32x32
1D SOLUTION

```c
__global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
    int i, j;
    i = blockIdx.x * blockDim.x + threadIdx.x;
    for (j=0; j<cols; j++)
        out[ j * rows + i ] = in[ i * cols + j ];
}
```
```c
_global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
    int i, j;
    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;
    out[ j * rows + i ] = in[ i * cols + j ];
}
```
RESULTS

- We are now at a 3x speedup over 2 sockets
- But how are we doing overall?
  - Peak for K40 is 288 GB/s
  - ~14% of peak
- Why is bandwidth utilization low?
  - Back to NVVP

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Throughput</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU+OMP</td>
<td>13.9 GB/s</td>
<td>--</td>
</tr>
<tr>
<td>GPU-1D</td>
<td>9.7 GB/s</td>
<td>0.7x</td>
</tr>
<tr>
<td>GPU-2D</td>
<td>41.6 GB/s</td>
<td>3.0x</td>
</tr>
</tbody>
</table>
• Occupancy is now much better
• All SMs have work
• Global load efficiency is low
• Bottleneck
  • Uncoalesced loads
KEY KERNEL OPTIMIZATIONS

Latency

Expressing Concurrency
Instruction Level Parallelism

Bandwidth

Coalesced Memory Access
Texture Memory
Shared memory

Compute

Reducing Divergence
USE NVVP TO FIND COALEScing PROBLEMS

Compile with -lineinfo

```c
__global__ void gpuTranspose_kernel(int rows, int cols)
{
    int i; int j;
    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;
    out[i*cols + j] = in[j*cols + i];
}
```

Uncoalesced Global Memory

Global memory loads and stores have poor access patterns, leading to inefficient use of global memory bandwidth. Select from the table below to see the source code which generates the inefficient global loads and stores.

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File: main.c</td>
<td>Global Store L2 Transactions/Access = 32.0 [1048576 L2 transactions for 32768 total executions]</td>
</tr>
</tbody>
</table>
WHAT IS AN UNCOALESCED GLOBAL LOAD?

- Global memory access happens in transactions of 32 or 128 bytes

  - **Coalesced** access:
    - A group of 32 contiguous threads ("warp") accessing adjacent words
    - Few transactions and high utilization

  - **Uncoalesced** access:
    - A warp of 32 threads accessing scattered words
    - Many transactions and low utilization
KEY KERNEL OPTIMIZATIONS

Latency
- Expressing Concurrency
- Instruction Level Parallelism

Bandwidth
- Coalesced Memory Access
- Texture Memory
- Shared memory

Compute
- Reducing Divergence
Either the read or the write will be uncoalesced

Two options

1. Use texture for loads
2. Modify indexing to eliminate uncoalesced accesses
   - Fastest changing dimension should be threadIdx.x
   - Need to stage through shared memory
TEXTURE CACHE

- Texture has an on-chip cache (Think L1)
- Two ways to use it (SM_35 or greater)
  - Implicit
    - Add const __restrict__
  - Explicit
    - Use the __ldg(ptr) intrinsic
_global__ void
gpuTranspose_kernel(int rows, int cols,
    float const * __restrict__ in, float *out)
{
    int i, j;
    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;
    out [ j * rows + i ] = in [ i * cols + j ];
}
Now at 48% of peak with very little effort
About a 10x speedup over 2 sockets

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Throughput</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU+OMP</td>
<td>13.9 GB/s</td>
<td>--</td>
</tr>
<tr>
<td>GPU-1D</td>
<td>9.7 GB/s</td>
<td>0.7x</td>
</tr>
<tr>
<td>GPU-2D</td>
<td>41.6 GB/s</td>
<td>3.0x</td>
</tr>
<tr>
<td>GPU-Tex</td>
<td>137.4 GB/s</td>
<td>9.9x</td>
</tr>
</tbody>
</table>
CAN WE DO BETTER?

- Not compute bound
- Not memory bound
- Assume latency bound

What if we did the shared memory approach?
KEY KERNEL OPTIMIZATIONS

Latency
- Expressing Concurrency
- Instruction Level Parallelism

Bandwidth
- Coalesced Memory Access
- Texture Memory
- Shared memory

Compute
- Reducing Divergence
**SHARED MEMORY**

- Accessible by all threads in a block
- Fast compared to global memory
  - Low access latency
  - High bandwidth
- Common uses:
  - Software managed cache
  - Data layout conversion
TRANSPOSING WITH SHARED MEMORY

- Read block coalesced into shared memory

Global Memory

Shared Memory
TRANSPOSING WITH SHARED MEMORY

Read block coalesced into shared memory

Transpose shared memory indices

Global Memory

Shared Memory
TRANSPOSING WITH SHARED MEMORY

- Read block\(_{ij}\) coalesced into shared memory
- Transpose shared memory indices
- Write transposed block to global memory
ALLOCATE SHARED MEMORY

```c
#define TILE_DIM 32

__global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
    int i, j;

    __shared__ float tile [ TILE_DIM ] [ TILE_DIM ];

    ...
}
```
#define TILE_DIM 32

_global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
    int i, j;
    __shared__ float tile [ TILE_DIM ] [ TILE_DIM ];
    ...
    ... = in [ j * cols + i ];
    ...
    out[ j * rows + i ] = ...
}
#define TILE_DIM 32

__global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
  int i, j;
  __shared__ float tile [ TILE_DIM ] [ TILE_DIM ];
  ...
  tile[ threadIdx.y ] [ threadIdx.x ] = in [ j * cols + i ];
  ...
  out[ j * rows + i ] = tile[ threadIdx.y ] [ threadIdx.x ];
}
#define TILE_DIM 32

__global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
    int i, j;
    __shared__ float tile [ TILE_DIM ] [ TILE_DIM ];
    ...
    tile[ threadIdx.y ] [ threadIdx.x ] = in [ j * cols + i ];
    ...
    out[ j * rows + i ] = tile[ threadIdx.x ] [ threadIdx.y ];
}
#define TILE_DIM 32

_global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
    int i, j;

    __shared__ float tile [ TILE_DIM ] [ TILE_DIM ];
    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;
    tile[ threadIdx.y ] [ threadIdx.x ] = in[ j * cols + i ];

    i = blockIdx.y * blockDim.y + threadIdx.x;
    j = blockIdx.x * blockDim.x + threadIdx.y;
    out[ j * rows + i ] = tile[ threadIdx.x ] [ threadIdx.y ];
}

TRANSPOSE BLOCK INDICES
#define TILE_DIM 32

__global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
    int i, j;
    __shared__ float tile [ TILE_DIM ] [ TILE_DIM ];

    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;

    tile[threadIdx.y][threadIdx.x] = in[j * cols + i];
    __syncthreads();

    i = blockIdx.y * blockDim.y + threadIdx.x;
    j = blockIdx.x * blockDim.x + threadIdx.y;
    out[j * rows + i] = tile[threadIdx.x][threadIdx.y];
}
Performance is worse
Shared memory should be faster than texture
What went wrong?

### RESULTS

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Throughput</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU+OMP</td>
<td>13.9 GB/s</td>
<td>--</td>
</tr>
<tr>
<td>GPU-1D</td>
<td>9.7 GB/s</td>
<td>0.7x</td>
</tr>
<tr>
<td>GPU-2D</td>
<td>41.6 GB/s</td>
<td>3.0x</td>
</tr>
<tr>
<td>GPU-Tex</td>
<td>137.4 GB/s</td>
<td>9.9x</td>
</tr>
<tr>
<td>GPU-Shared</td>
<td>116.0 GB/s</td>
<td>8.3x</td>
</tr>
</tbody>
</table>
SHARED MEMORY PROFILE

- **Shared Memory** is achieving good bandwidth
- **However, replay overhead** is high

- **Bottleneck**: Bank Conflicts
SHARED MEMORY ORGANIZATION

- Organized in 32 independent banks
- Optimal access: all words from different banks
  - Separate banks per thread
  - Banks can multicast
- Multiple words from same bank serialize
Example: **32x32 SMEM array**

Warp accesses a column:

- 32-way bank conflicts (threads in a warp access the same bank)

Bank 0

Bank 1

... 

Bank 31

Accesses along row produces 0 bank conflicts

Accesses along column produces 32 bank conflicts
Add a column for padding:

32x33 SMEM array

Warp accesses a column:

32 different banks, no bank conflicts

Bank 0
Bank 1
... Bank 31

Accesses along row produces 0 bank conflicts

Accesses along column produces 0 bank conflicts
#define TILE_DIM 32

_global__ void
gpuTranspose_kernel(int rows, int cols, float *in, float *out)
{
    int i, j;
    __shared__ float tile [ TILE_DIM ] [ TILE_DIM + 1 ];

    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;

    tile[ threadIdx.y ] [ threadIdx.x ] = in [ j * cols + i ];
    __syncthreads();

    i = blockIdx.y * blockDim.y + threadIdx.x;
    j = blockIdx.x * blockDim.x + threadIdx.y;
    out[ j * rows + i ] = tile[ threadIdx.x ] [ threadIdx.y ];
}
Performance is better

**Bottleneck:**
- Kepler requires 100+ lines in flight per SM to saturate DRAM
- 1 line-in-flight per warp @ 100% occupancy = 64 lines in flight

**Solution:**
- Process multiple elements
  - Instruction-level parallelism
  - More lines-in-flight
  - Less `__syncthreads` overhead
  - Amortize cost of indexing and thread launch

---

### RESULTS

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Throughput</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU+OMP</td>
<td>13.9 GB/s</td>
<td>--</td>
</tr>
<tr>
<td>GPU-1D</td>
<td>9.7 GB/s</td>
<td>0.7x</td>
</tr>
<tr>
<td>GPU-2D</td>
<td>41.6 GB/s</td>
<td>3.0x</td>
</tr>
<tr>
<td>GPU-Tex</td>
<td>137.4 GB/s</td>
<td>9.9x</td>
</tr>
<tr>
<td>GPU-Shared</td>
<td>116.0 GB/s</td>
<td>8.3x</td>
</tr>
<tr>
<td>GPU-Padded</td>
<td>150.6 GB/s</td>
<td>10.8x</td>
</tr>
</tbody>
</table>
KEY KERNEL OPTIMIZATIONS

Latency
- Expressing Concurrency
- Instruction Level Parallelism

Bandwidth
- Coalesced Memory Access
- Texture Memory
- Shared memory

Compute
- Reducing Divergence
PROCESS MULTIPLE ELEMENTS PER THREAD

- Increases ILP
- Reduces Exposed Latency
- Allows compiler to schedule instructions more efficiently
**FINAL PROFILE**

- 82% of peak bandwidth
- Approaching memcpy speeds

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Throughput</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU+OMP</td>
<td>13.9 GB/s</td>
<td>--</td>
</tr>
<tr>
<td>GPU-1D</td>
<td>9.7 GB/s</td>
<td>0.7x</td>
</tr>
<tr>
<td>GPU-2D</td>
<td>41.6 GB/s</td>
<td>3.0x</td>
</tr>
<tr>
<td>GPU-Tex</td>
<td>137.4 GB/s</td>
<td>9.9x</td>
</tr>
<tr>
<td>GPU-Shared</td>
<td>116.0 GB/s</td>
<td>8.3x</td>
</tr>
<tr>
<td>GPU-Padded</td>
<td>150.6 GB/s</td>
<td>10.8x</td>
</tr>
<tr>
<td>GPU-Multi</td>
<td>238.4 GB/s</td>
<td>17.2x</td>
</tr>
</tbody>
</table>
Final Results

- Use NVVP to identify bottlenecks
- Use optimization techniques to eliminate bottlenecks
- Refer to GTC archives for complete optimization techniques

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Throughput</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU+OMP</td>
<td>13.9 GB/s</td>
<td>--</td>
</tr>
<tr>
<td>GPU-1D</td>
<td>9.7 GB/s</td>
<td>0.7x</td>
</tr>
<tr>
<td>GPU-2D</td>
<td>41.6 GB/s</td>
<td>3.0x</td>
</tr>
<tr>
<td>GPU-Tex</td>
<td>137.4 GB/s</td>
<td>9.9x</td>
</tr>
<tr>
<td>GPU-Shared</td>
<td>116.0 GB/s</td>
<td>8.3x</td>
</tr>
<tr>
<td>GPU-Padded</td>
<td>150.6 GB/s</td>
<td>10.8x</td>
</tr>
<tr>
<td>GPU-Multi</td>
<td>238.4 GB/s</td>
<td>17.2x</td>
</tr>
</tbody>
</table>

www.gputechconf.com/gtcnew/on-demand-gtc.php

Search “GPU Performance Analysis and Optimization”
KEY KERNEL OPTIMIZATIONS

Latency
- Expressing Concurrency
- Instruction Level Parallelism

Bandwidth
- Coalesced Memory Access
- Texture Memory
- Shared memory

Compute
- Reducing Divergence
CONTROL FLOW

**Single-Instruction Multiple-Threads (SIMT) model**
- A single instruction is issued for a warp (thread-vector) at a time

**SIMT compared to SIMD:**
- SIMD requires vector code in each thread
- SIMT allows you to write scalar code per thread
  - Vectorization is handled by hardware

**Note:**
- All contemporary processors (CPUs and GPUs) are built by aggregating vector processing units
- Vectorization is needed to get performance on CPUs and GPUs
CONTROL FLOW

```c
if ( ... ) {
    // then-clause
}
else {
    // else-clause
}
```
EXECUTION WITHIN WARPS IS COHERENT

(“vector” of threads)
EXECUTION DIVERGES WITHIN A WARP
Don’t freak out about this.

Some `if` - `then` is OK.

Example -

Checking edge cases when you have tiled input...
MATH OPTIMIZATION TIPS

Use “intrinsics” for math operations if you can.

A couple bits lower precision, but much faster.

\[
\_\_\text{sin}() \quad \_\_\text{cos}() \quad \_\_\text{exp}()
\]

Use double-precision when you mean to.

\[
\text{float } a = b + 2.5f
\]
GRAND OPTIMIZATION SUMMARY

APOD = Assess   Parallelize   Optimize   Deploy
   Use profiler
   Deploy early
   Optimizing is increasing effort. Complicates code.

Measure & Improve Memory BW
   Express sufficient parallelism
   Coalesce global memory accesses
   Reduce threads in block when using synchthreads

Minimize Thread Divergence
   Avoid branchy code
   Don’t freak out over this

Consider Fast Math
   Intrinsics

Stream & Asynchronous Copies
   Overlap memory transfers and computation
Getting Started on Your Own
LINKS TO GET STARTED

developer.nvidia.com

Self-paced labs: nvidia.qwiklab.com

90-minute labs, simply need a supported web browser
UDACITY PARALLEL PROGRAMMING COURSE

10,000+ Students Already Taking the course!

IT’S FREE!

Udacity.com
Thank you!