An Introduction to Graphics Processing Unit Architecture and Programming Models

Argonne Training Program on Exascale Computing

Tim Warburton
John K. Costain Faculty Chair in the College of Science
Professor Of Mathematics and Affiliate Faculty in CMDA
Virginia Tech
Overview

Background on instructor.

Part 0: GPU Myths.

Part 1: NVIDIA Graphical Processing Unit

Part 2: The Compute Unified Device Architecture (CUDA)

- NVIDIA’s threaded offload programming model.
- Hands on: area of the Mandelbrot

Part 3: Interlude on CUDA optimization.

Part 4: Portable programming models:

- The Open Computing Language (OpenCL)
- The Open Concurrent Computing Abstraction (OCCA)
- Hands on:
  - Task: flow simulations movie.
  - Prep: find a png image with white background.
My Research...

Goal: fast, scalable, flexible & accurate numerical PDE solvers adapted for modern many-core architectures.

High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers. GPU programming tools & applications. Industrial collaboration.


Basic science  Application  Industrial Scale
My Research…

Goal: fast, scalable, flexible & accurate numerical PDE solvers adapted for modern many-core architectures.


Basic science  Application  Industrial Scale

High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers. GPU programming tools & applications. Industrial collaboration.
Some GPU Accelerated Apps…

We have developed accelerated solvers: seismic inversion, electromagnetics, fluid dynamics, gas dynamics, thermal therapy…

High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers. GPU programming tools & applications. Industrial collaboration.
Some GPU Accelerated Apps…

We have developed accelerated solvers: seismic inversion, electromagnetics, fluid dynamics, gas dynamics, thermal therapy…

- Flow modeling
- MRI Guided Laser Interstitial Therapy
- Electromagnetic scattering
- Tsunami propagation
- Global elastodynamics
- Kinetic Models
- Shock Flows

High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers. GPU programming tools & applications. Industrial collaboration.
Some GPU Accelerated Apps...

We have developed accelerated solvers: seismic inversion, electromagnetics, fluid dynamics, gas dynamics, thermal therapy...

High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers. GPU programming tools & applications. Industrial collaboration.
Some GPU Accelerated Apps...

We have developed accelerated solvers: seismic inversion, electromagnetics, fluid dynamics, gas dynamics, thermal therapy...

- Flow modeling
- MRI Guided Laser Interstitial Therapy
- Electromagnetic scattering
- Tsunami propagation
- Global elastodynamics
- Kinetic Models
- Shock Flows

High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers. GPU programming tools & applications. Industrial collaboration.
Accelerators: slides & repos

Slides:

www.math.vt.edu/people/tcew/ATPESC17

Examples:

git clone https://github.com/tcew/ATPESC17

OCCA repo:

git clone https://github.com/libocca/occa
Part 0: GPU Reality Check
Myth #1: GPU 100x faster than CPU

NVIDIA P100: High Bandwidth Memory up to 732 GB/s

Intel E7-8894 v4: 4 memory channels up to 85 GB/s per socket

http://www.nvidia.com/object/tesla-p100.html
https://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2.40-GHz

Majority of HPC codes are memory and network bound.
Myth #2: GPUs are expensive

NVIDIA P100: ~$5K

Intel E7-8894 v4: ~$9K

Price estimates from http://www.thinkmate.com/
http://www.anandtech.com/show/11121/intel-xeon-e7-8894-v4-cpu-24c-48t-9000-usd

Prices for non-consumer GPUs are carefully calibrated to be similar to CPU. [ these are both premium high-end processors ]
Myth 3: GPU & CPU are very different

GPUs and CPUs both consist of multiple cores each equipped with SIMD vector units.

NVIDIA P100: 56 “cores” with 4 32-way SIMT units

Intel E7-8894 v4: 24 hyper-threading cores with 256 bit AVX2 instructions

http://www.nvidia.com/object/tesla-p100.html
https://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2.40GHz
Myth #4.0: OpenACC is magic

“good” OpenACC codes are quite often derived from good CUDA implementations.
Myth #4.1: CUDA is magic

“good” CUDA codes quite often emerge from a prolonged gestation.
Reality Check

It takes more than 3 hours to master GPUs…
It takes more than 3 hours to master GPUs…

… but we can discuss some of the basics …
It takes more than 3 hours to master GPUs…

… but we can discuss some of the basics …

… there are many web resources …
Reality Check

It takes more than 3 hours to master GPUs…

… but we can discuss some of the basics …

… there are many web resources …

… and nothing beats hands on.
Reality Check

It takes more than 3 hours to master GPUs…

… but we can discuss some of the basics …

… add some background to Nikolay Sakharnykh’s talk & hands on …

… there are many web resources …

… and nothing beats hands on.
Part 1: From CPU to GPU
CPU: architecture follows purpose

Original design goals for CPUs:

• Make single threads very fast.

• Reduce latency through large caches.

• Predict, speculate.
Modern “CPU-Style” core design emphasizes individual thread performance.

Execution context: memory and hardware associated to a specific stream of instructions, e.g. registers.
GPU: massively parallel processing

The main purpose of graphics processing units is to project textured polygons onto the screen in a fiercely competitive consumer-facing industry.

This is an embarrassingly parallel process and specialized MPP chips have been created by ATi (now AMD), Intel, NVIDIA et al to perform floating point intensive operations to render scenes in realtime.
The main purpose of graphics processing units is to project textured polygons onto the screen in a fiercely competitive consumer-facing industry.

This is an embarrassingly parallel process and specialized MPP chips have been created by ATi (now AMD), Intel, NVIDIA et al to perform floating point intensive operations to render scenes in realtime.
The main purpose of graphics processing units is to project textured polygons onto the screen in a fiercely competitive consumer-facing industry. This is an embarrassingly parallel process and specialized MPP chips have been created by ATi (now AMD), Intel, NVIDIA et al to perform floating point intensive operations to render scenes in realtime.
Design goals for GPUs:

• Throughput matters and single threads do not.

• Hide memory latency through parallelism.

• Let programmer deal with “raw” storage hierarchy.

• Avoid high frequency clock speed:
  • Desirable for portable devices, consoles, laptops…

GPU: early example

Die floorplan: AMD RV770 (2008) 55 nm, 800 SP simultaneous ops
The majority of the silicon is devoted to computation
GPU: early example

Comparison of block diagram of vintage GPU and CPU

http://www.anandtech.com/show/2556/8
The result is a core that will stall due to code branching and memory fetches.

The first big idea that differentiates GPU and CPU core design: slim down the footprint of each core.

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
The first big idea that differentiates GPU and CPU core design: slim down the footprint of each core.

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

The result is a core that will stall due to code branching and memory fetches.
The first big idea that differentiates GPU and CPU core design: slim down the footprint of each core.

The result is a core that will stall due to code branching and memory fetches.

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
The first big idea that differentiates GPU and CPU core design: slim down the footprint of each core.

Idea #1:
Remove the modules that help a single instruction execute fast.

Large data cache

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

The result is a core that will stall due to code branching and memory fetches.
The result is a core that will stall due to code branching and memory fetches.

GPU: slim down the core

The first big idea that differentiates GPU and CPU core design: slim down the footprint of each core.

Idea #1: Remove the modules that help a single instruction execute fast.

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
GPU: replicate cores

Each simplified core takes less space so double up

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
GPU: replicate cores

... and again ...

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
GPU: the clone cores (ahem)

... and again to yield 16 independent instruction streams ...

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
GPU: the clone cores (ahem) ...

... and again to yield 16 independent instruction streams ...

But: GPU rendering instruction streams are typically very similar.

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
The Fetch/Decode units in those 16 clone cores are likely fetching and decoding the same instruction streams for rendering graphics...
“single instruction multiple data” SIMD model:
share the cost of the instruction stream across many ALUs

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
In reality the GPU cores are equipped with “substantial” register files and shared memory.

Provision each SIMD group with shared memory

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
GPU: multiple independent cores

Summary: 128 parallel instruction streams. Organized into 16 independent cores, each with 8 instruction streams.

The cores are cloned into a massively parallel processor.

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

This is an abstract GPU [ memory system not shown ]
16 Maxwell cores each have four SIMD clusters with 32 ALUs. Data streams at ~56 GFLOAT/s and peak 4.6 TFLOP/s (SP)
Compilers may need to be coaxed into generating vector instructions for CPU. Recall: “Performance, SIMD, Vectorization and Performance Tuning” talk by James Reindeer.
Compilers may need to be coaxed into generating vector instructions for CPU.

GPU cores are engineered to switch quickly between threads to recover stalls.

Skylake core: 180 Integer registers and 168 floating point registers

Maxwell core: 16K registers

http://en.wikichip.org/wiki/intel/microarchitectures/skylake
A GPU has multiple cores and each core:

- Has one (or more) wide SIMD vector units.
- Wide SIMD vector units execute one instruction stream.
- Has a pool of shared memory.
- Shares a register file shared privately among all the ALUs.
- Fast switches thread blocks to hide memory latency.
- Branching code ("ifs") involves partial serialization.

Nice summary:

* SIMD width here is the number of ALUs in one of the core’s vector unit. The actual specifics vary but this is a good abstract viewpoint.
Part 2: NVIDIA GPUs
Core Evolution

NVIDIA’s Compute Unified Device Architecture
GPU programming model
Theoretical peak performance requires ~28 FLOP per float moved between device & memory !!!

Note: for the Fermi generation cards they put the L1 and L2 caches back 😃

Modern GPUs combine: multiple wide vector processing cores with local and global shared-memory.

Each Fermi core (SM) has a SIMD clusters of 32 FPUs
Data streams at ~50 GFLOAT/s and computes up to 1.4 TFLOP/s (SP)
GPU: NIVIDA Titan

GK110: 15 cores that cluster 192 FPU each.

Each Kepler core (SMX) has six SIMD clusters of 32 ALUs.

Data streams at ~70 GFLOAT/s and peak 4+ TFLOP/s (SP)

http://www.tomshardware.com/reviews/geforce-gtx-titan-gk110-review,3438.html
16 Maxwell cores each have four SIMD clusters with 32 ALUs. Data streams at ~56 GFLOAT/s and peak 4.6 TFLOP/s (SP)
GPU: Pascal GPU

Professional NVIDIA Pascal GP100 architecture with 60 cores using 16nm fab size

60 Pascal GP100 cores each with two SIMD clusters of 32 ALUs (3840 Total).
HBM2 memory streams data at ~1 TB/s and peak 10.6 TFLOP/s (SP), 5.3TFLOP/s (DP)
Professional NVIDIA Pascal GP100 architecture with 60 cores using 16nm fab size

Consumer variants use GDDR5(x) memory with up to 480 GB/s bandwidth and up to 3584 ALUs with peak 10.1 TFLOP/s (SP) 0.3 TFLOP/s (DP)

60 Pascal GP100 cores each with two SIMD clusters of 32 ALUs (3840 Total). HBM2 memory streams data at ~1 TB/s and peak 10.6 TFLOP/s (SP), 5.3 TFLOP/s (DP)
The FPU cluster sizes have ballooned: 16 - 24 - 32 - 192 but the shared memory and register file have not grown accordingly.

Trend: smaller die process yields more space for shared memory and registers.

The FPU clusters ("core") in 3 latest NVIDIA processor architectures.

GPU: Kepler to Maxwell to Pascal

Kepler SMX

Maxwell SMM

Pascal SM

http://www.ubergizmo.com/2014/02/nvidia-maxwell-gpu-for-geforce-cards/

Figure 8. Pascal GP100 SM Unit

Designed for High Performance Double Precision

Double precision arithmetic is at the heart of many HPC applications such as linear algebra, numerical simulation, and quantum chemistry. Therefore, one of the key design goals for GP100 was to significantly improve the delivered performance for these use cases.

Each SM in GP100 features 32 double precision (FP64) CUDA Cores, which is one-half the number of FP32 single precision CUDA Cores. A full GP100 GPU has 1920 FP64 CUDA Cores. This 2:1 ratio of single precision (SP) units to double precision (DP) units aligns better with GP100's new datapath configuration, allowing the GPU to process DP workloads more efficiently. Like previous GPU architectures, GP100 supports full IEEE 754-2008 compliant single precision and double precision arithmetic, including support for the fused multiply-add (FMA) operation and full speed support for denormalized values.

Note: Kepler GK110 had a 3:1 ratio of SP units to DP units.
Part 2b: GPU programming with CUDA Threading

NVIDIA's Compute Unified Device Architecture
GPU programming model
Warped Terminology

CUDA

• Is laced (ahem) with terminology derived from weaving like “warp”, “thread”, “texture”.

• We refer instead to a thread array and SIMD groups.
CUDA is used to program NVIDIA GPUs. CUDA includes a HOST API and a DEVICE kernel programming language.
CUDA: offload model

The programmer explicitly moves data between HOST and DEVICE.

Key observation: the DEVICE and HOST are asynchronous. Operations are queued on the DEVICE.

https://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-780
CUDA: offload model

The programmer explicitly moves data between HOST and DEVICE

1. cudaMalloc: allocate memory for a DEVICE array

Key observation: the DEVICE and HOST are asynchronous. Operations are queued on the DEVICE.

https://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-780
CUDA: offload model

The programmer explicitly moves data between HOST and DEVICE.

1. cudaMalloc: allocate memory for a DEVICE array

2. cudaMemcpy: copy data from HOST to DEVICE array

Key observation: the DEVICE and HOST are asynchronous. Operations are queued on the DEVICE.
CUDA: offload model

The programmer explicitly moves data between HOST and DEVICE

1. cudaMemcpy: allocate memory for a DEVICE array

2. cudaMemcpy: copy data from HOST to DEVICE array

3. Queue kernel task on DEVICE

Key observation: the DEVICE and HOST are asynchronous. Operations are queued on the DEVICE.

https://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-780
CUDA: offload model

The programmer explicitly moves data between HOST and DEVICE.

1. cudaMemcpy: copy data from DEVICE to HOST array
2. cudaMalloc: allocate memory for a DEVICE array
3. Queue kernel task on DEVICE
4. cudaMemcpy: copy data from DEVICE to HOST array

Key observation: the DEVICE and HOST are asynchronous. Operations are queued on the DEVICE.

https://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-780
CUDA: offload model

The programmer explicitly moves data between HOST and DEVICE

1. cudaMalloc: allocate memory for a DEVICE array
2. cudaMemcpy: copy data from HOST to DEVICE array
3. Queue kernel task on DEVICE
4. cudaMemcpy: copy data from DEVICE to HOST array

Key observation: the DEVICE and HOST are asynchronous. Operations are queued on the DEVICE.

https://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-780
GPU: natural thread model

The GPU architecture admits a natural parallel threading model

- Programmer decides how to partition a compute task into kernel tasks:
  - Programmer then decides how to partition a kernel into independent work-blocks:
    - Work-block assigned to a core with sufficient resources to process it:
      - Each core processes the work-block with a work-group of “threads”
        - The work-group is batch processed in sub-groups of SIMD* work-items.
        - Each work-item processed by a “thread” passing through a SIMD lane.
        - A stalling SIMD group of “threads” is idled until it can continue.
        - “Threads” in a work-group can collaborate through shared memory.
        - The work-block stays resident until completed by core (using resources).
  - Main assumption: same instructions for independent work-groups.

* SIMD here is the number of ALUs in one of the core’s vector unit.
# CUDA: example HOST code

Overview of C-like CUDA code that runs on the HOST:

```c
#include "cuda.h"

int main(int argc, char **argv)
{
    int N = 3789; // size of array for this DEMO

    float *d_a; // Allocate DEVICE array
    cudaMalloc((void**)&d_a, N*sizeof(float));

    int B = 512;
dim3 dimBlock(B,1,1); // 512 threads per thread-block
dim3 dimGrid(((N+B-1)/B, 1, 1); // Enough thread-blocks to cover N

    // Queue kernel on DEVICE
    simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);

    // HOST array
    float *h_a = (float*) calloc(N, sizeof(float));

    // Transfer result from DEVICE array to HOST array
    cudaMemcpy(h_a, d_a, N*sizeof(float), cudaMemcpyDeviceToHost);

    // Print out result from HOST array
    for(int n=0; n<N; ++n) printf("h_a[%d] = %f\n", n, h_a[n]);
}
```

Note the .cu file extension.
We use NVIDIA's CUDA Compiler nvcc to compile .cu files.
CUDA: host code

Overview of C-like HOST code for a simple kernel that fills a vector of length N

1. Allocate array space on DEVICE:

2. Design thread-array:

3. Queue compute task on DEVICE:

4. Copy results from DEVICE to HOST:

Key API calls: cudaMemcpy, cudaMemcpy
CUDA: host code

Overview of C-like HOST code for a simple *kernel* that fills a vector of length N

1. Allocate array space on DEVICE:

   ```c
   float *d_a; // Allocate DEVICE array (pointers used as array handles)
   cudaMalloc((void**)&d_a, N*sizeof(float));
   ```

2. Design thread-array:

3. Queue compute task on DEVICE:

4. Copy results from DEVICE to HOST:

   Key API calls: cudaMalloc, cudaMemcpy
Overview of C-like HOST code for a simple *kernel* that fills a vector of length N

1. Allocate array space on DEVICE:

```c
float *d_a; // Allocate DEVICE array (pointers used as array handles)
cudaMalloc((void**) &d_a, N*sizeof(float));
```

2. Design thread-array:

```c
dim3 dimBlock(512,1,1); // 512 threads per thread-block
dim3 dimGrid((N+511)/512, 1, 1); // Enough thread-blocks to cover N
```

3. Queue compute task on DEVICE:

4. Copy results from DEVICE to HOST:

*Key API calls: cudaMalloc, cudaMemcpy*
CUDA: host code

Overview of C-like HOST code for a simple kernel that fills a vector of length N

1. Allocate array space on DEVICE:

   ```
   float *d_a; // Allocate DEVICE array (pointers used as array handles)
   cudaMalloc((void**) &d_a, N*sizeof(float));
   ```

2. Design thread-array:

   ```
   dim3 dimBlock(512,1,1); // 512 threads per thread-block
   dim3 dimGrid((N+511)/512, 1, 1); // Enough thread-blocks to cover N
   ```

3. Queue compute task on DEVICE:

   ```
   // specify number of threads with <<< block count, thread count >>>
   simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);
   ```

4. Copy results from DEVICE to HOST:

   ```
   ```

Key API calls: cudaMalloc, cudaMemcpy
CUDA: host code

Overview of C-like HOST code for a simple *kernel* that fills a vector of length N

1. Allocate array space on DEVICE:

   ```
   float *d_a; // Allocate DEVICE array (pointers used as array handles)
   cudaMalloc((void**)&d_a, N*sizeof(float));
   ```

2. Design thread-array:

   ```
   dim3 dimBlock(512,1,1); // 512 threads per thread-block
   dim3 dimGrid((N+511)/512, 1, 1); // Enough thread-blocks to cover N
   ```

3. Queue compute task on DEVICE:

   ```
   // specify number of threads with <<< block count, thread count >>>
   simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);
   ```

4. Copy results from DEVICE to HOST:

   ```
   float *h_a = (float*) calloc(N, sizeof(float));
   cudaMemcpy(h_a, d_a, N*sizeof(float), cudaMemcpyDeviceToHost)
   ```

*Key API calls: cudaMalloc, cudaMemcpy*
CUDA: motivating serial function

Before jumping into how to write a CUDA kernel we consider first a serial function that fills an array with entries 0:N-1

```c
void serialSimpleKernel(int N, float *d_a)
{
    for(n=0;n<N;++n){ // loop over N entries
        d_a[n] = n;
    }
}
```

To make a two level thread parallel implementation we partition (or chunk) the n-loop.
void serialSimpleKernel(int N, float *d_a){
    for(n=0;n<N;++n){ // loop over N entries
        d_a[n] = n;
    }
}

We can think of splitting the n-loop into tiles of size 4: n=t+4b.
Here: block dimension = 4 and grid dimension = 5.
We tile the n-loop into equal sized tiles (here tile size is blockDim)

```c
void tiledSerialSimpleKernel(int N, float *d_a){
    for(int b=0;b<gridDim;++b){ // loop over blocks
        for(int t=0;t<blockDim;++t){ // loop inside block
            // Convert thread and thread-block indices into array index
            const int n  = t + b*blockDim;
            // If index is in [0,N-1] add entries
            if(n<N) // guard against an inexact tiling
                d_a[n] = n;
        }
    }
}
```

We assume the loop boundaries (gridDim and blockDim) are externally specified variables. We also assume that: \( N \leq \text{gridDim} \times \text{blockDim} \). Tiling also referred to chunking sometimes.
CUDA: tiled serial function

We rename variables to conform with CUDA naming convention.
dim3 type intrinsic variables: threadIdx, blockDim, blockIdx, gridDim

```c
void tiledSerialSimpleKernel(int N, float *d_a){
    for(blockIdx.x=0;blockIdx.x<gridDim.x;++blockIdx.x){  // loop over blocks
        for(threadIdx.x=0;threadIdx.x<blockDim.x;++threadIdx.x){  // loop inside block
            // Convert thread and thread-block indices into array index
            const int n = threadIdx.x + blockDim.x*blockIdx.x;

            // If index is in [0,N-1] add entries
            if(n<N)
                d_a[n] = n;
        }
    }
}
```

Key observation: the body of the tiled loop can now be mapped to a thread.

We also assume that: $N \leq \text{gridDim.x} \times \text{blockDim.x}$
We rename variables to conform with CUDA naming convention.
dim3 type intrinsic variables: threadIdx, blockDim, blockIdx, gridDim

```c
void tiledSerialSimpleKernel(int N, float *d_a) {
    for (blockIdx.x = 0; blockIdx.x < gridDim.x; ++blockIdx.x) { // loop over blocks
        for (threadIdx.x = 0; threadIdx.x < blockDim.x; ++threadIdx.x) { // loop inside block

            // Convert thread and thread-block indices into array index
            const int n = threadIdx.x + blockDim.x * blockIdx.x;

            // If index is in [0,N-1] add entries
            if (n < N) {
                d_a[n] = n;
            }
        }
    }
}
```

Key observation: the body of the tiled loop can now be mapped to a thread.

We also assume that: $N \leq \text{gridDim.x} \times \text{blockDim.x}$
We rename variables to conform with CUDA naming convention. dim3 type intrinsic variables: threadIdx, blockDim, blockIdx, gridDim

We also assume that: \( N \leq \text{gridDim.x} \times \text{blockDim.x} \)
Each thread can determine its (multi-dimensional) rank with respect to both its rank in the thread-block and the rank of the thread-block itself.

<table>
<thead>
<tr>
<th>Description</th>
<th>Fastest index</th>
<th>Slowest index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread indices in thread-block</td>
<td>threadIdx.x</td>
<td>threadIdx.y</td>
</tr>
<tr>
<td>Dimensions of thread-block</td>
<td>blockDim.x</td>
<td>blockDim.y</td>
</tr>
<tr>
<td>Block indices.</td>
<td>blockIdx.x</td>
<td>blockIdx.y</td>
</tr>
<tr>
<td>Dimensions of grid of thread-blocks</td>
<td>gridDim.x</td>
<td>gridDim.y</td>
</tr>
</tbody>
</table>

Remember: we can identify task parallelism by associating tasks with combination of thread-index and block-index.

* Best practice: avoid frequent branching based on threadIdx or blockIdx.  
  * three dimensional grid of thread-blocks supported as of CUDA 2.*
### CUDA: limitations

The CUDA compute capability evolves with ongoing NVIDIA GPU hardware revisions.

<table>
<thead>
<tr>
<th>Technical specifications</th>
<th>Compute capability (version)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td>2</td>
</tr>
<tr>
<td>Maximum x-, y-, or z-dimension of a grid of thread blocks</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum dimensionality of thread block</td>
<td>3</td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>512</td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td>64</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread</td>
<td>128</td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
</tr>
</tbody>
</table>

Table credit: CUDA wikipedia page (http://en.wikipedia.org/wiki/CUDA)
void tiledSerialSimpleKernel(int N, float *d_a){
    for(blockIdx.x=0;blockId.x<gridDim.x;++blockIdx.x){ // loop over blocks
        for(threadIdx.x=0;threadIdx.x<blockDim.x;++threadIdx.x){ // loop inside block
            // Convert thread and thread-block indices into array index
            const int n = threadIdx.x + blockDim.x*blockIdx.x;
            // If index is in [0,N-1] add entries
            if(n<N)
                d_a[n] = n;
        }
    }
}

We also assume that: $N \leq \text{gridDim.x*blockDim.x}$
CUDA: tiled serial function

We rename variables to conform with CUDA naming convention. dim3 type intrinsic variables: threadIdx, blockDim, blockIdx, gridDim

```c
void tiledSerialSimpleKernel(int N, float *d_a){

    for(blockIdx.x=0;blockId.x<gridDim.x;++blockIdx.x){ // loop over blocks

        for(threadIdx.x=0;threadIdx.x<blockDim.x;++threadIdx.x){ // loop inside block

            // Convert thread and thread-block indices into array index
            const int n = threadIdx.x + blockDim.x*blockIdx.x;

            // If index is in [0,N-1] add entries
            if(n<N)
                d_a[n] = n;

        }
    }
}
```

Key observation: the body of the tiled loop can now be mapped to a thread.

We also assume that: \( N \leq \text{gridDim.x*blockDim.x} \)
CUDA: tiled serial function

We rename variables to conform with CUDA naming convention. dim3 type intrinsic variables: threadIdx, blockDim, blockIdx, gridDim

```c
void tiledSerialSimpleKernel(int N, float *d_a){
    for(blockIdx.x=0;blockId.x<gridDim.x;++blockIdx.x){ // loop over blocks
        for(threadIdx.x=0;threadIdx.x<blockDim.x;++threadIdx.x){ // loop inside block
            // Convert thread and thread-block indices into array index
            const int n = threadIdx.x + blockDim.x*blockIdx.x;
            // If index is in [0,N-1] add entries
            if(n<N)
                d_a[n] = n;
        }
    }
}
```

Key observation: the body of the tiled loop can now be mapped to a thread.

We also assume that: $N \leq \text{gridDim.x*blockDim.x}$
CUDA: simple array operation kernel

This body of the kernel function is the inner code from the chunked version of the function. The kernel is executed by every thread in the specified array of threads.
CUDA: simple array operation kernel

```c
__global__ void simpleKernel(int N, float *d_a) {
    // Convert thread and thread-block indices into array index
    const int n = threadIdx.x + blockDim.x*blockIdx.x;

    // If index is in [0,N-1] add entries
    if(n<N)
        d_a[n] = n;
}
```

Key observation: the loops are implicitly executed by thread parallelism and do not appear in the CUDA kernel code.

This body of the kernel function is the inner code from the chunked version of the function. The kernel is executed by every thread in the specified array of threads.
CUDA: simple array operation kernel

```c
__global__ void simpleKernel(int N, float *d_a){
    // Convert thread and thread-block indices into array index
    const int n = threadIdx.x + blockDim.x*blockIdx.x;
    // If index is in [0,N-1] add entries
    if(n<N)
        d_a[n] = n;
}
```

This body of the kernel function is the inner code from the chunked version of the function. The kernel is executed by every thread in the specified array of threads.

Key observation: the loops are implicitly executed by thread parallelism and do not appear in the CUDA kernel code.
CUDA: simple array operation kernel

```c
__global__
void simpleKernel(int N, float *d_a){
  // Convert thread and thread-block indices into array index
  const int n = threadIdx.x + blockDim.x*blockIdx.x;
  // If index is in [0,N-1] add entries
  if(n<N)
    d_a[n] = n;
}
```

This body of the kernel function is the inner code from the chunked version of the function. The kernel is executed by every thread in the specified array of threads.

Key observation: the loops are implicitly executed by thread parallelism and do not appear in the CUDA kernel code.
CUDA: array operation host code

```c
#include "cuda.h"
#include <stdio.h>
#include <stdlib.h>

__global__ void simpleKernel(int N, float *d_a)
{
    // Convert thread and thread-block indices into array index
    const int n = threadIdx.x + blockDim.x*blockIdx.x;

    // If index is in [0,N-1] add entries
    if(n<N)
        d_a[n] = n;
}

int main(int argc, char **argv)
{
    int N = 512; // size of array for this DEMO

    float *d_a; // Allocate DEVICE array
    cudaMalloc((void**)&d_a, N*sizeof(float));

    dim3 dimBlock(512,1,1); // 512 threads per thread-block
    dim3 dimGrid((N+511)/512,1,1); // Enough thread-blocks to cover N

    // Queue kernel on DEVICE
    simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);

    // HOST array
    float *h_a = (float*) malloc(N, sizeof(float));

    // Transfer result from DEVICE to HOST
}
```

**Note:**

a. dimBlock does not need to be a multiple of 32.
b. CUDA will pad to multiple of 32
c. excess threads will be idled.

TW: go to code on github for walk through:
CUDA: code samples @ github

Make sure you can compile and run this code (see next slide)!
Hands On #1: compiling/running CUDA

This example requires CUDA GPU, drivers, and SDK is installed.

```
colley.alcf.anl.gov: make sure .soft.cooley includes and resoft
+mvapich2
+cuda-7.5.18
+ffmpeg-1.0.1
@default
```

```
# clone the examples on the login node:
git clone https://github.com/tcew/ATPESC17

# if you haven’t already done so, queue an interactive job request:
qsub -A ATPESC17 -I -n 1 -t 120 -q training

# find the source
cd ATPESC17/examples/cuda/simple

# compile on node with the NVIDIA CUDA compiler (nvcc) installed
nvcc -o simple simple.cu

# run on node with the NVIDIA CUDA runtime libraries installed
./simple
```

Make sure you can complete this exercise now if possible!
Source code: https://github.com/tcew/ATPESC17/examples/cuda/simple
void tiledSerialMultidKernel(int N, float *d_a) {

    for(blockIdx.z=0; blockIdx.z<gridDim.z; ++blockIdx.z){  // loop over z-blocks
        for(blockIdx.y=0; blockIdx.y<gridDim.y; ++blockIdx.y){  // loop over y-blocks
            for(blockIdx.x=0; blockIdx.x<gridDim.x; ++blockIdx.x){  // loop over x-blocks

                // loop over thread indices in thread-block
                for(threadIdx.z=0; threadIdx.z<blockDim.z; ++threadIdx.z){
                    for(threadIdx.y=0; threadIdx.y<blockDim.y; ++threadIdx.y){
                        for(threadIdx.x=0; threadIdx.x<blockDim.x; ++threadIdx.x){

                            // Convert thread and thread-block indices into array index
                            const int nx = threadIdx.x + blockDim.x*blockIdx.x;
                            const int ny = threadIdx.y + blockDim.y*blockIdx.y;
                            const int nz = threadIdx.z + blockDim.z*blockIdx.z;

                            // Perform action based on thread-ranks
                            ...;

                        }
                    }
                }
            }
        }
    }
}

Key observation: the iterations in each iteration are assumed to be independent
CUDA: multi-d array operation kernel

__global__ void multidKernel(int N, float *d_a) {
    // Convert thread and thread-block indices into array index
    const int nx = threadIdx.x + blockDim.x*blockIdx.x;
    const int ny = threadIdx.y + blockDim.y*blockIdx.y;
    const int nz = threadIdx.z + blockDim.z*blockIdx.z;

    operations based on thread ranks;
}

Key observation: the loops are implicitly executed by thread parallelism and do not appear in the CUDA kernel code.

This body of the kernel function is the inner code from the chunked version of the function. The kernel is executed by every thread in the specified array of threads.
// HOST code to queue kernel
dim3 dimGrid(GX, GY, GZ), dimBlock(BX, BY, BZ);
multidKernel <<< dimGrid, dimBlock >>> (N, d_a);

__global__ void multidKernel(int N, float *d_a){

// Convert thread and thread-block indices into array index
const int nx = threadIdx.x + blockDim.x*blockIdx.x;
const int ny = threadIdx.y + blockDim.y*blockIdx.y;
const int nz = threadIdx.z + blockDim.z*blockIdx.z;

operations based on thread ranks;
}

Key observation: the loops are implicitly executed by thread parallelism and do not appear in the CUDA kernel code.

This body of the kernel function is the inner code from the chunked version of the function. The kernel is executed by every thread in the specified array of threads.
We consider a more substantial example: solving the Poisson problem.

Elliptic Poisson problem:

\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \text{ in } \Omega = [-1,1] \times [-1,1] \]

\[ u = 0 \text{ on } \partial \Omega \]
Elliptic Poisson problem:

\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \] in \( \Omega = [-1,1] \times [-1,1] \)

\[ u = 0 \] on \( \partial \Omega \)

We represent the numerical solution at a regular grid of finite-difference nodes.
CUDA: elliptic solver example

First step discretize the equations into a set of linear constraints.

Elliptic Poisson problem:

\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \text{ in } \Omega = [-1,1] \times [-1,1] \]

\[ u = 0 \text{ on } \partial \Omega \]

Discrete Poisson problem (assuming Cartesian grid):

\[ \left( \frac{u_{j(i+1)} - 2u_{ji} + u_{j(i-1)}}{\delta^2} \right) + \left( \frac{u_{(j+1)i} - 2u_{ji} + u_{(j-1)i}}{\delta^2} \right) = f_{ji} \text{ for } i, j = 1, \ldots, N \]

\[ u_{ji} = 0 \text{ for } i = 0, N + 1 \text{ or } j = 0, N + 1 \]

The derivative operators are approximated by second order differences.

The discrete Poisson problem is approximated at the finite difference nodes.
CUDA: elliptic solver example

First step discretize the equations into a set of linear constraints.

Elliptic Poisson problem:

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)
\]

\[u = 0 \text{ on } \partial \Omega\]

Discrete Poisson problem (assuming Cartesian grid):

\[
\left(\frac{u_{j(i+1)} - 2u_{ji} + u_{j(i-1)}}{\delta^2}\right) + \left(\frac{u_{(j+1)i} - 2u_{ji} + u_{(j-1)i}}{\delta^2}\right) = f_{ji}
\]

The derivative operators are approximated by second order differences. The discrete Poisson problem is approximated at the finite difference nodes.
CUDA: discrete elliptic example

We solve the linear system for the unknowns using the stationary iterative Jacobi method

Discrete Poisson problem (assuming Cartesian grid):

\[
\left( \frac{u_{j(i+1)} - 2u_{ji} + u_{j(i-1)}}{\delta^2} \right) + \left( \frac{u_{(j+1)i} - 2u_{ji} + u_{(j-1)i}}{\delta^2} \right) = f_{ji} \text{ for } i, j = 1, ..., N
\]

\[u_{ji} = 0 \text{ for } i = 0, N + 1 \text{ or } j = 0, N + 1\]

Jacobi iteration for discrete Poisson problem:

\[
\left( \frac{u_{j(i+1)}^k - 2u_{ji}^{k+1} + u_{j(i-1)}^k}{\delta^2} \right) + \left( \frac{u_{(j+1)i}^k - 2u_{ji}^{k+1} + u_{(j-1)i}^k}{\delta^2} \right) = f_{ji} \text{ for } i, j = 1, ..., N
\]

\[u_{ji} = 0 \text{ for } i = 0, N + 1 \text{ or } j = 0, N + 1\]
CUDA: elliptic solver example

Rearranging we are left with a simple five point recurrence:

Jacobi iteration for discrete Poisson problem:

\[
\left( \frac{u_{ji}^{k} - 2u_{ji}^{k+1} + u_{ji}^{k}}{\delta^2} \right) + \left( \frac{u_{(j+1)i}^{k} - 2u_{ji}^{k+1} + u_{(j-1)i}^{k}}{\delta^2} \right) = f_{ji} \text{ for } i, j = 1, \ldots, N
\]

\[
u_{ji} = 0 \text{ for } i = 0, N + 1 \text{ or } j = 0, N + 1
\]

 Iterate:

\[
u_{ji}^{k+1} = \frac{1}{4} \left( -\delta^2 f_{ji} + u_{(j+1)i}^{k} + u_{(j-1)i}^{k} + u_{j(i+1)}^{k} + u_{j(i-1)}^{k} \right) \text{ for } i, j = 1, \ldots, N
\]

while:

\[
\epsilon := \sqrt{\sum_{i=1}^{i=N} \sum_{j=1}^{j=N} (u_{ji}^{k+1} - u_{ji}^{k})^2} > tol
\]
CUDA: parallelism for solver example

For the iterate step we note:

each node can update independently for maximum parallelism.

Iterate:

\[ u_{ji}^{k+1} = \frac{1}{4} \left( -\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{j(i+1)}^k + u_{j(i-1)}^k \right) \text{ for } i, j = 1, ..., N \]
CUDA: parallelism for solver example

For the iterate step we note:
each node can update independently for maximum parallelism.

Iterate:

$$u_{ji}^{k+1} = \frac{1}{4} \left( -\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{j(i+1)}^k + u_{j(i-1)}^k \right)$$

for $i, j = 1, \ldots, N$.
At the start we set: rhs=-delta*delta*f

Iterate:

\[ u_{ji}^{k+1} = \frac{1}{4} \left( -\delta^2 f_{ji} + u_{j(i+1)i}^k + u_{j(j+1)i}^k + u_{j(i+1)i}^k + u_{j(i-1)i}^k \right) \text{ for } i, j = 1, ..., N \]

Serial kernel:

```c
void jacobi(const int N,
            const datafloat *rhs,
            const datafloat *u,
            datafloat *newu){
    for(int i=0; i<N; ++i){
        for(int j=0; j< N; ++j){
            // Get linear index into NxN
            // inner nodes of (N+2)x(N+2) grid
            const int id = (j + 1)*(N + 2) + (i + 1);
            newu[id] = 0.25f*(rhs[id]
                                 + u[id - (N+2)]
                                 + u[id + (N+2)]
                                 + u[id - 1]
                                 + u[id + 1]);
        }
    }
}
```

Note: we use an NxN array of threads and change leave the edge nodes unchanged.
CUDA: parallel Jacobi iteration

For CUDA: each thread can update a node independently for maximum parallelism.

Iterate:

\[ u_{ji}^{k+1} = \frac{1}{4} \left( -\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{j(i+1)}^k + u_{j(i-1)}^k \right) \quad \text{for } i, j = 1, ..., N \]

Note: we use an NxN array of threads and leave the edge nodes unchanged.

At the start we set: \( rhs = -\text{delta} \times \text{delta} \times f \)

https://github.com/tcew/ATPESC17/tree/master/examples/cuda/jacobi
CUDA: parallelism for solver reduction

To make this more parallel we need to split the termination into CUDA thread-blocks:

Reduction:

\[ \varepsilon := \sum_{i=0}^{i=N-1} v_i \]

Block reduction (B blocks)

\[ \varepsilon := \sum_{b=0}^{b=B-1} \left( \sum_{i=0}^{i=T-1} v_{i+bT} \right) \]

\[ B := \frac{N}{T} \]

Next we need to distribute the inner sum work over the threads in each of the B thread-blocks.
Thread-block tree reduction in pseudo-code:

\[
t = \text{thread index in thread block}; \\
\text{alive} = \text{number of threads in thread block}; \\
s_{\text{sumu}}[t] = u[\text{global thread index}]; \\
\]

\[
\textbf{while}(\text{alive}>1){ \\
\begin{align*}
&\text{synchronize threads in thread-block;} \\
&\text{alive} /= 2; \\
&\text{if}(t < \text{alive}) \\
&\hspace{1em} s_{\text{sumu}}[t] += s_{\text{sumu}}[t+\text{alive}]; \\
&\} \\
\text{if}(t==0) \text{blocksumu}[\text{block index}] = s_u[0];
\]

Here the \texttt{__shared__} array is read/writeable only by threads in the same thread-block.
All threads in the thread-block have to enter the \texttt{__syncthreads()} before any of them can continue.

CUDA: parallel reduction

Standard tree reduction at the thread-block level!!

\[
\text{Target: } \sum_{i=0}^{T-1} v_i
\]
CUDA: parallel reduction

Standard tree reduction at the thread-block level!!

CUDA partial reduction kernel:

```c
__global__ void partialReduceResidual(const int entries,
datafloat *u,
datafloat *newu,
datafloat *blocksum){

__shared__ datafloat s_blocksum[BDIM];
const int id = blockIdx.x*blockDim.x + threadIdx.x;
s_blocksum[threadIdx.x] = 0;
if(id < entries){
    const datafloat diff = u[id] - newu[id];
s_blocksum[threadIdx.x] = diff*diff;
}

int alive = blockDim.x;
int t = threadIdx.x;
while(alive>1){
    __syncthreads(); // barrier (make sure s_blocksum is ready)
    alive /= 2; // reduce active threads
    if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
}
if(t==0)
    blocksum[blockIdx.x] = s_blocksum[0];
}
```

Here the __shared__ array is read/writeable only by threads in the same thread-block. All threads in the thread-block have to enter the __syncthreads() before any of them can continue.
CUDA: parallel reduction

Standard tree reduction at the thread-block level!!

CUDA partial reduction kernel:

```c
__global__ void partialReduceResidual(const int entries,
datafloat *u,
datafloat *newu,
datafloat *blocksum){

__shared__ datafloat s_blocksum[BDIM];

cost int id = blockIdx.x*blockDim.x + threadIdx.x;
s_blocksum[threadIdx.x] = 0;

if(id < entries){
    const datafloat diff = u[id] - newu[id];
s_blocksum[threadIdx.x] = diff*diff;
}

int alive = blockDim.x;
int t = threadIdx.x;

while(alive>1){
    __syncthreads(); // barrier (make sure s_blocksum is ready)
    alive /= 2; // reduce active threads
    if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
}

if(t==0)
    blocksum[blockIdx.x] = s_blocksum[0];
}
```

Here the __shared__ array is read/writeable only by threads in the same thread-block. All threads in the thread-block have to enter the __syncthreads() before any of them can continue.
CUDA: parallel reduction

Standard tree reduction at the thread-block level!!

CUDA partial reduction kernel:

```c
__global__ void partialReduceResidual(const int entries,
datafloat *u,
datafloat *newu,
datafloat *blocksum){

__shared__ datafloat s_blocksum[BDIM];

const int id = blockIdx.x*blockDim.x + threadIdx.x;

s_blocksum[threadIdx.x] = 0;
if(id < entries){
    const datafloat diff = u[id] - newu[id];
    s_blocksum[threadIdx.x] = diff*diff;
}

int alive = blockDim.x;
int t = threadIdx.x;
while(alive>1){
    __syncthreads(); // barrier (make sure s_blocksum is ready)
    alive /= 2; // reduce active threads
    if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
}
if(t==0)
    blocksum[blockIdx.x] = s_blocksum[0];
}
```

Here the __shared__ array is read/writeable only by threads in the same thread-block. All threads in the thread-block have to enter the __syncthreads() before any of them can continue.
**CUDA partial reduction kernel:**

```c
__global__ void partialReduceResidual(const int entries,
datafloat *u,
datafloat *newu,
datafloat *blocksum){

__shared__ datafloat s_blocksum[BDIM];
const int id = blockIdx.x*blockDim.x + threadIdx.x;
s_blocksum[threadIdx.x] = 0;
if(id < entries){
    const datafloat diff = u[id] - newu[id];
    s_blocksum[threadIdx.x] = diff*diff;
}

int alive = blockDim.x;
int t = threadIdx.x;
while(alive>1){
    __syncthreads(); // barrier (make sure s_blocksum is ready)
    alive /= 2; // reduce active threads
    if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
}
if(t==0)
    blocksum[blockIdx.x] = s_blocksum[0];
}
```

Here the __shared__ array is read/writeable only by threads in the same thread-block. All threads in the thread-block have to enter the __syncthreads() before any of them can continue.
CUDA: parallel reduction

Standard tree reduction at the thread-block level!!

<table>
<thead>
<tr>
<th>Step</th>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td>1+7</td>
<td>8</td>
<td>8+11</td>
<td>19</td>
<td>19+17</td>
<td>36</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>3</td>
<td>3+8</td>
<td>11</td>
<td>11+6</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>5</td>
<td>5+6</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2</td>
<td>2+4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Target: \[ \sum_{i=0}^{i=T-1} v_i \]

Here the __shared__ array is read/writeable only by threads in the same thread-block. All threads in the thread-block have to enter the __syncthreads() before any of them can continue.
CUDA: parallel reduction

Standard tree reduction at the thread-block level!!

CUDA partial reduction kernel:

```c
__global__ void partialReduceResidual(const int entries,
  datafloat *u,
  datafloat *newu,
  datafloat *blocksum) {

  __shared__ datafloat s_blocksum[BDIM];

  const int id = blockIdx.x*blockDim.x + threadIdx.x;
  s_blocksum[threadIdx.x] = 0;
  if(id < entries) {
    const datafloat diff = u[id] - newu[id];
    s_blocksum[threadIdx.x] = diff*diff;
  }

  int alive = blockDim.x;
  int t = threadIdx.x;
  while(alive>1) {
    __syncthreads(); // barrier (make sure s_blocksum is ready)
    alive /= 2; // reduce active threads
    if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
  }
  if(t==0) blocksum[blockIdx.x] = s_blocksum[0];
}
```

Here the __shared__ array is read/writeable only by threads in the same thread-block. All threads in the thread-block have to enter the __syncthreads() before any of them can continue.
CUDA: parallel reduction

The .cu file contains both the partialSum reduction DEVICE kernel and the HOST code. Note: the HOST code includes event based timing of the kernel execution.
Converting the Mandelbrot example from Tim Mattson’s talk to CUDA.

#1. Retrieve the files:
```bash
git clone https://github.com/tcew/ATPESC17
```

#2. Complete the skeleton code

ATPESC17/handsOn/mandelbrot/mandelbrot.cu

[ Do things labelled TASK, don’t touch things marked FREEBIE ]

#3. Hints:

To compile (on `cooley.alcf.anl.gov`):
```bash
nvcc -arch=sm_30  -o mandelbrot mandelbrot.cu -lm
```

To run (on a cooley compute node):
```bash
./mandelbrot
```

Useful CUDA keywords (google if :

*thread rank*: threadIdx.x, threadIdx.y, blockIdx.x, blockIdx.y, blockDim

*keywords*: __device__, __global__

[ to turn off CUDA kernel optimization: nvcc  -Xptxas -O3 -arch sm_30 -o mandelbrot mandelbrot.cu -lm ]
Part 3: Interlude on CUDA performance
Classic Definition of “Supercomputer”

This is a well known definition of a “supercomputer”

“A supercomputer is a device for turning compute-bound problems into I/O-bound problems.”

Ken Batcher*

Attribution is a little cloudy: *possibly Seymour Cray
“Arithmetic is cheap, bandwidth is money, latency is physics.”

Mark Hoemmen*

NVIDIA can be viewed as a company that sells expensive GDDR memory.
CUDA: memory options

The different memory spaces on the GPU have different characteristics

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Latency</th>
<th>Cached</th>
<th>Access</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On-chip</td>
<td>1</td>
<td>N/A</td>
<td>Read/write</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>1000</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>2</td>
<td>N/A</td>
<td>Read/write</td>
<td>All threads in a block</td>
<td>Block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>1000</td>
<td>Yes*</td>
<td>Read/write</td>
<td>All threads &amp; host</td>
<td>Application</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>1-1000</td>
<td>Yes</td>
<td>Read</td>
<td>All threads &amp; host</td>
<td>Application</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>1000</td>
<td>Yes</td>
<td>Read</td>
<td>All threads in a block</td>
<td>Application</td>
</tr>
<tr>
<td>Read-only Cache</td>
<td>On-chip</td>
<td>Low</td>
<td>Yes</td>
<td>Read/write</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Adapted from Timothy Lanfear's CUDA Tutorial Slides
## CUDA: limitations

Recall the table showing that CUDA compute capabilities have evolved over time

<table>
<thead>
<tr>
<th>Technical specifications</th>
<th>Compute capability (version)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td></td>
</tr>
<tr>
<td>Maximum x-, y-, or z-dimension of a grid of thread blocks</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum dimensionality of thread block</td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td></td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread</td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td></td>
</tr>
</tbody>
</table>

There are several interesting tidbits here.

Table credit: CUDA wikipedia page (http://en.wikipedia.org/wiki/CUDA)
CUDA: occupancy calculator

The amount of register space is highly constrained: kernels with high register count will have low occupancy.

CUDA Occupancy Calculator: (download) spreadsheet tallies up register count, shared memory count, and thread count per thread-block to estimate how many thread-blocks can be resident.
## CUDA: shared memory banks

Shared memory is organized as interwoven “memory banks” with separate managers. A shared memory array spans up to 32 independent memory banks.

<table>
<thead>
<tr>
<th>Bank</th>
<th>31</th>
<th>63</th>
<th>95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank 30</td>
<td>30</td>
<td>62</td>
<td>94</td>
</tr>
<tr>
<td>Bank 5</td>
<td>5</td>
<td>37</td>
<td>69</td>
</tr>
<tr>
<td>Bank 4</td>
<td>4</td>
<td>36</td>
<td>68</td>
</tr>
<tr>
<td>Bank 3</td>
<td>3</td>
<td>35</td>
<td>67</td>
</tr>
<tr>
<td>Bank 2</td>
<td>2</td>
<td>34</td>
<td>66</td>
</tr>
<tr>
<td>Bank 1</td>
<td>1</td>
<td>33</td>
<td>65</td>
</tr>
<tr>
<td>Bank 0</td>
<td>0</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>
CUDA: shared memory banks

To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.

OK: all threads in the SIMD group access different shared memory banks
CUDA: shared memory banks

To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.

<table>
<thead>
<tr>
<th>Shared Memory: memory space organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank 31</td>
</tr>
<tr>
<td>Bank 30</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>Bank 5</td>
</tr>
<tr>
<td>Bank 4</td>
</tr>
<tr>
<td>Bank 3</td>
</tr>
<tr>
<td>Bank 2</td>
</tr>
<tr>
<td>Bank 1</td>
</tr>
<tr>
<td>Bank 0</td>
</tr>
</tbody>
</table>

OK: all threads in the SIMD group access different shared memory banks
CUDA: shared memory broadcast

To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.

OK: all threads in the SIMD group access the same entry results in an efficient broadcast.
CUDA: shared memory broadcast

To maintain parallelism each of the 32 threads in a “Warp” (SIMD group) should access a different bank unless they all access the same entry.

BAD: all threads in the SIMD group access the same bank resulting in serialization.
CUDA: accessing device memory

High end NVIDIA GPUs either have 256 or 384 bit wide memory bus to device memory

1. GPU has a “coalescer” that collects DRAM memory requests.
2. The coalescer efficiently streams contiguous, aligned blocks of memory by avoiding repeated address setup.
3. The GPU bus to DRAM consists of 6x 64 bit busses.
4. Each bus has an independent memory controller.

Rule of thumb: avoid non unitary stride DEVICE (DRAM) array access.
Useful slides, these, and image credit: link
CPU Optimization Techniques

Cache

- Data loaded into cache from aligned contiguous blocks (cache lines)

Vectorization

- Use large registers instructions to perform operations in parallel.
- Also uses continuous load instructions to vectorize efficiently.

Continuous memory accesses are used for both, cache storage and vectorization
**Multithreading**

- Threads capable of fully parallelizing *generic* instructions (ignoring bandwidth).

  - Perfect scaling … **without** barriers, joins, or other types of thread-dependencies.

- **SIMD Lanes**
GPU Architecture

- Independent work-groups are launched.
- Work-groups contain groups of work-items, “parallel” threads.

Kernel code describes the work-item operations
GPU Optimization Techniques

Work-groups

- Groups of work-items.
- No communication between work-groups.
- Designed for independent group parallelism.
- Avoid inter-block synchronization (deadlocks).
- Avoid data race dependencies between blocks.

Work-items

- Work-items are executed in parallel, able to barrier and share data using shared memory (& CUDA's shuffle).
- Avoid data race dependencies between work-items.
Parallel Work-item Execution

- Work-items are launched in subsets of 32 or 64.
- Each set of work-items execute same instructions.
- No parallel branching (in the subset).

Data Transfer

- Low individual bandwidth and high latency.
- Coalesced memory access on contiguous and aligned work-items.
Exposing vectorization / SIMD parallelism are vital in both architectures.
Part 4: Portable programming models
RIP: Blinky 01/15/14-08/03/15

Blinky was a MacBook Pro with discrete NVIDIA GPU and OpenMP, OpenCL, CUDA …

Latest MBP comes with AMD GPU and/or Intel Iris GPU, no CUDA, and default clang compilers omit OpenMP.
Many-core: fragmentation

Zoo of competing architectures and programming models (with vendor bias)

Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Many-core: fragmentation

Zoo of competing architectures and programming models (with vendor bias)

Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Many-core: fragmentation

Zoo of competing architectures and programming models (with vendor bias)

Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Many-core: fragmentation

Zoo of competing architectures and programming models (with vendor bias)

Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Many-core: fragmentation

Zoo of competing architectures and programming models (with vendor bias)

Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Many-core: fragmentation

Zoo of competing architectures and programming models (with vendor bias)

Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Part 4a: Open Compute Language (OpenCL)
OpenCL: standards committee

Quick-reference-card for OpenCL 2.0: (link)

The Khronos Group administers the OpenCL standard.
OpenCL: standard for multicore

OpenCL allows us to write cross platform code
(customization need for best performance)
OpenCL Working Group

- Diverse industry participation
  - Processor vendors, system OEMs, middleware vendors, application developers
- Many industry-leading experts involved in OpenCL’s design
  - A healthy diversity of industry perspectives
- Apple made initial proposal and is very active in the working group
  - Serving as specification editor

The OpenCL standard changes relatively slowly over time compared to CUDA.
Credit: Khronos Group
OpenCL: why?

Processor Parallelism

CPUs
Multiple cores driving performance increases

Multi-processor programming – e.g. OpenMP

Emerging Intersection

OpenCL

Heterogeneous Computing

GPUs
Increasingly general purpose data-parallel computing

Graphics APIs and Shading Languages

OpenCL is a programming framework for heterogeneous compute resources

Emphasis on heterogeneous computing.
Credit: Khronos Group
OpenCL: why?

It’s a Heterogeneous World

- A modern platform Includes:
  - One or more CPUs
  - One or more GPUs
  - DSP processors
  - ... other?

OpenCL lets Programmers write a single portable program that uses ALL resources in the heterogeneous platform.

GMCH = graphics memory control hub
ICH = Input/output control hub
OpenCL: when?

CUDA and OpenCL are competing standards for GPGPU programming

GPGPU “quiet time”

Only a few hardy souls tried GPU computing before CUDA was released.
OpenCL is **very** closely related to CUDA

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>Kernel</td>
</tr>
<tr>
<td>Host program</td>
<td>Host program</td>
</tr>
<tr>
<td>Thread</td>
<td>Work item</td>
</tr>
<tr>
<td>Thread block</td>
<td>Work group</td>
</tr>
<tr>
<td>Grid</td>
<td>NDRange (index space)</td>
</tr>
</tbody>
</table>

The rapid development of OpenCL helps explain the similarities
OpenCL: thread indexing

OpenCL is **very** closely related to CUDA

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Local indices:</strong></td>
<td><strong>Local indices:</strong></td>
</tr>
<tr>
<td>threadIdx.x</td>
<td>threadIdx.y</td>
</tr>
<tr>
<td></td>
<td>get_local_id(0)</td>
</tr>
<tr>
<td></td>
<td>get_local_id(1)</td>
</tr>
<tr>
<td><strong>Global indices:</strong></td>
<td><strong>Global indices:</strong></td>
</tr>
<tr>
<td>blockIdx.x*blockDim.x + threadIdx.x</td>
<td>blockIdx.y*blockDim.y + threadIdx.y</td>
</tr>
<tr>
<td></td>
<td>get_global_id(0)</td>
</tr>
<tr>
<td></td>
<td>get_global_id(1)</td>
</tr>
</tbody>
</table>

The rapid development of OpenCL helps explain the similarities
OpenCL: thread array dimensions

OpenCL is **very** closely related to CUDA

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>gridDim.x</td>
<td>get_num_groups(0)</td>
</tr>
<tr>
<td>blockIdx.x</td>
<td>get_group_id(0)</td>
</tr>
<tr>
<td>blockDim.x</td>
<td>get_local_size(0)</td>
</tr>
<tr>
<td>gridDim.x*blockDim.</td>
<td>get_global_size(0)</td>
</tr>
</tbody>
</table>
OpenCL: kernel language qualifiers

OpenCL is **very** closely related to CUDA

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong> function</td>
<td>__kernel function</td>
</tr>
<tr>
<td><strong>device</strong> function</td>
<td>function</td>
</tr>
<tr>
<td><strong>constant</strong> variable</td>
<td>__constant variable</td>
</tr>
<tr>
<td><strong>device</strong> variable</td>
<td>__global variable</td>
</tr>
<tr>
<td><strong>shared</strong> variable</td>
<td>__local variable</td>
</tr>
</tbody>
</table>
OpenCL: memory model

Again, the memory model for CUDA and OpenCL are very similar

The rapid development of OpenCL helps explain the similarities

Image system not shown

AMD OpenCL slides
OpenCL is very flexible, allowing simultaneous heterogeneous computing with possibly multiple implementations, command queues, & devices in one system [CPU+GPUs]

To set up a device:

1. Choose platform (implementation of OpenCL) from list of platforms:
   • `clGetPlatformIDs`

2. Choose device on that platform (for instance a specific CPU or GPU):
   • `clGetDeviceIDs`

3. Create a context on the device (manager for tasks):
   • `clCreateContext`

4. Create command queue on a context on the chosen device:
   • `clCreateCommandQueue`
The include files ...

CUDA
#include <cuda.h>

OpenCL
#if defined __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif
Any given system may have multiple OpenCL platforms from different vendors installed. We will choose one of the returned platform IDs.

For flexibility we first have to choose the OpenCL “platform”

```c
#include <cuda.h>

int main()
{
    // nothing special to do (really only one CUDA platform)

    ... 
    cl_platform_id platforms[100];
    cl_uint platforms_n;
    
    /* get list of platforms(platform == OpenCL implementation) */
    clGetPlatformIDs(100, platforms, &platforms_n);
    ...
```
Each OpenCL platform can interact with one or more compute devices.

Next we choose a device supported by the platform.

```c
int dev = 0;
cudaSetDevice(dev);
...
```

```c
cl_device_id devices[100];
cl_uint ndevices;

clGetDeviceIDs(platforms[plat], CL_DEVICE_TYPE_ALL, 100, devices, &ndevices);

if(dev>=ndevices) {
    printf("invalid device\n");
    exit(0);
}

// choose user specified device
cl_device_id device = devices[dev];
...
```
OpenCL: setting up a context

Next we choose a context (manager) for the chosen device.

```c
cl_context context;

// make compute context on device (pfn_notify is an error callback function)
context = clCreateContext((cl_context_properties *)NULL, 1, &device, &pfn_notify, (void*)NULL, &err);
```
Next we choose a context (manager) for the chosen device.

```c
// make compute context on device (PFN_notify is an error callback function)
cl_command_queue queue =
    clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &err);
```

// not necessary although you may wish to use cudaStreamCreate

...
OpenCL: compiling a DEVICE kernel

I wasn’t kidding about flexibility.

Since the platform+device+context is chosen at runtime
it is customary to build compute kernels at runtime.

To set up a kernel on a DEVICE:

1. Represent kernel source code as a C character array:

   ```
   const char *source = 
   "__kernel void foo(int N, __global float *x){
     int id = get_global_id(0);
     if(id<N) 
       x[id] = id;
   }"
   
   ``

2. Create a “program” from the source code:

   - `clCreateProgramWithSource`

3. Compile and build the “program”:

   - `clBuildProgram`

4. Check for compilation errors:

   - `clGetProgramBuildInfo`

5. Build executable kernel:

   - `clCreateKernel`
OpenCL: building a kernel

We now need to build the kernel [ some steps skipped for brevity ]

```c
/* create program from source */
cl_program program = clCreateProgramWithSource(context, 1,
                                               (const char **) &source, (size_t*) NULL, &err);

/* compile and build program */
const char *allFlags = " ";
err = clBuildProgram(program, 1, &device, allFlags,
                      (void (*)(cl_program, void*)) NULL, NULL);

/* omitted error checking */
...
/* create runnable kernel */
cl_kernel kernel = clCreateKernel(program, functionName, &err);
```

And we have to do that for each kernel.
OpenCL: are we there yet?

Unbelievably no.

To execute the kernel:

1. Just like CUDA we need to allocate storage on the DEVICE:
   - clCreateBuffer

2. We need to add the input arguments one at a time to the kernel:
   - clSetKernelArg

3. Specify the local work-group size and global thread array sizes.

4. Queue the kernel
   - clEnqueueNDRangeKernel

5. Wait for the kernel to finish:
   - clFinish
In this case we have provided CL with a host pointer and clCreateBuffer copies from h_x to c_x.

We next allocate array space on the DEVICE:

```c
int N = 100; /* vector size */

/* size of array */
size_t sz = N*sizeof(float);

float *d_a; // CUDA uses pointer for array handles

cudaMalloc((void**)&d_a, N*sizeof(float));
```

```c
int N = 100; /* vector size */

/* size of array */
size_t sz = N*sizeof(float);

/* create device buffer and copy from host buffer */
cl_mem c_x = clCreateBuffer(context,
                            CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sz, h_x, &err);
```
Not quite: we now need to specify each kernel argument one by one.

```c
/* now set kernel arguments one by one */
clSetKernelArg(kernel, 0, sizeof(int), &N);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &c_x);

/* set thread array */
int dim = 1;
size_t local[3] = {256, 1, 1};
size_t global[3] = {256*(((N+255-1)/256)), 1, 1};

/* queue up kernel */
clEnqueueNDRangeKernel(queue, kernel, dim, 0, global, local, 0, (cl_event*)NULL, NULL);
```
The kernel programming languages are similar:

**CUDA**

```c
__global__ void simpleKernel(int N, float *a)
{
    /* get thread coordinates */
    int i = threadIdx.x + blockIdx.x*blockDim.x;

    /* do simple task */
    if(i<N)
        a[i] = i;
}
```

**OpenCL**

```c
__kernel void simpleKernel(int N, __global float *a)
{
    /* get thread coordinates */
    int i = get_global_id(0);

    /* do simple task */
    if(i<N)
        a[i] = i;
}
```

Some minor differences in syntax & identifiers
OpenCL: summary

OpenCL seems to be a panacea: it works on everything…

- OpenCL has a bit of a bad reputation:
  - CUDA has a richer ecosystem of tools & libraries.
  - CUDA has more extensive documentation and tutorials.
  - Platform/device/context/queue complexity.
  - Competing vendor priorities.
  - The vendors offer differing levels of support.
    - OpenCL: Intel: CPU vectorization is flaky.
    - OpenCL: OS X: CPU limited work-items per work-group
  - Rumors constantly circulate about EOL.

- On the other hand:
  - Runtime compilation adds several optimization opportunities without templating.
  - OpenCL is library based, so no special compilers are required.
  - Vendor independence is important.
Iterate:
\[
    u_{ji}^{k+1} = \frac{1}{4} (-\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{ji(i+1)}^k + u_{ji(i-1)}^k) \text{ for } i, j = 1, \ldots, N
\]

Serial kernel:
```c
void jacobi(const int N,
             const double *rhs,
             const double *u,
             double *newu){
    for(int i=0;i<N;++i) {
        for(int j=0;j<N;++j) {
            // Get linear index into NxN
            // inner nodes of (N+2)x(N+2) grid
            const int id = (j + 1)*(N + 2) + (i + 1);
            newu[id] = 0.25f*(rhs[id] + u[id - (N+2)] + u[id + (N+2)] + u[id - 1] + u[id + 1]);
        }
    }
}
```

CUDA kernel:
```c
__global__ void jacobi(const int N,
                       const double *rhs,
                       const double *u,
                       double *newu){
    // Get thread indices
    const int i = blockIdx.x*blockDim.x + threadIdx.x;
    const int j = blockIdx.y*blockDim.y + threadIdx.y;
    // Check that this is a legal node
    if((i < N) && (j < N)){
        // Get linear index onto (N+2)x(N+2) grid
        const int id = (j + 1)*(N + 2) + (i + 1);
        newu[id] = 0.25f*(rhs[id] + u[id - (N+2)] + u[id + (N+2)] + u[id - 1] + u[id + 1]);
    }
}
```

OpenCL kernel:
```c
void jacobi(const int N,
            const double *rhs,
            const double *u,
            double *newu){
    // Get thread indices
    const int i = ??;
    const int j = ??;
    if((i < N) && (j < N)){
        // Get linear index onto (N+2)x(N+2) grid
        const int id = (j + 1)*(N + 2) + (i + 1);
        newu[id] = 0.25f*(rhs[id] + u[id - (N+2)] + u[id + (N+2)] + u[id - 1] + u[id + 1]);
    }
}
```

Note explicit loops in serial kernel and hidden loops in CUDA and OpenCL kernels.
OpenCL: comparing Jacobi kernels

Recalling the Poisson example: side by side comparison of serial v. CUDA v. OpenCL kernel

Iterate:

\[
\begin{align*}
    u_{ji}^{k+1} &= \frac{1}{4} \left( -\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{j(i+1)}^k + u_{j(i-1)}^k \right) \\
    &\text{for } i, j = 1, ..., N
\end{align*}
\]

Serial kernel:

```c
void jacobi(const int N,
            const double *rhs,
            const double *u,
            double *newu){
    for(int i=0;i<N;++i){
        for(int j=0;j<N;++j){
            // Get linear index into NxN
            // inner nodes of (N+2)x(N+2) grid
            const int id = (j + 1)*(N + 2) + (i + 1);

            newu[id] = 0.25f*(rhs[id] + u[id - (N+2)] + u[id + (N+2)] + u[id - 1] + u[id + 1]);
        }
    }
}
```

CUDA kernel:

```c
__global__ void jacobi(const int N,
                       const double *rhs,
                       const double *u,
                       double *newu){
    // Get thread indices
    const int i = blockIdx.x*blockDim.x + threadIdx.x;
    const int j = blockIdx.y*blockDim.y + threadIdx.y;
    // Check that this is a legal node
    if((i < N) & (j < N)){
        // Get linear index onto (N+2)x(N+2) grid
        const int id = (j + 1)*(N + 2) + (i + 1);

        newu[id] = 0.25f*(rhs[id] + u[id - (N+2)] + u[id + (N+2)] + u[id - 1] + u[id + 1]);
    }
}
```

OpenCL kernel:

```c
__kernel void jacobi(const int N,
                     __global const double *rhs,
                     __global const double *u,
                     __global double *newu){
    // Get thread indices
    const int i = get_global_id(0);
    const int j = get_global_id(1);
    if((i < N) & (j < N)){
        // Get linear index into (N+2)x(N+2) grid
        const int id = (j + 1)*(N + 2) + (i + 1);

        newu[id] = 0.25f*(rhs[id] + u[id - (N+2)] + u[id + (N+2)] + u[id - 1] + u[id + 1]);
    }
}
```

Note explicit loops in serial kernel and hidden loops in CUDA and OpenCL kernels.
OpenCL: partial reduction

Standard tree reduction at the thread-block level!!

More details on OpenCL reduction next time.
OpenCL: partial reduction

Standard tree reduction at the thread-block level!!

CUDA partial reduction kernel:

```c
__global__ void partialReduceResidual(const int entries,
            double *u,
            double *newu,
            double *blocksum){

__shared__ double s_blocksum[BDIM];
const int id = blockIdx.x*blockDim.x + threadIdx.x;
int alive = blockDim.x;
int t = threadIdx.x;
s_blocksum[threadIdx.x] = 0;
if(id < entries){
    const double diff = u[id] - newu[id];
    s_blocksum[threadIdx.x] = diff*diff;
}
while(alive>1){
    __syncthreads(); // barrier (make sure s_blocksum is ready)
    alive /= 2;
    if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
}
if(t==0)
    blocksum[blockIdx.x] = s_blocksum[0];
}
```

More details on OpenCL reduction next time.
OpenCL: partial reduction

Standard tree reduction at the thread-block level!!

CUDA partial reduction kernel:
```c
__global__ void partialReduceResidual(const int entries,
    double *u,
    double *newu,
    double *blocksum){

    __shared__ double s_blocksum[BDIM];

    const int id = blockIdx.x*blockDim.x + threadIdx.x;
    int alive = blockDim.x;
    int t = threadIdx.x;
    s_blocksum[threadIdx.x] = 0;
    if(id < entries){
        const double diff = u[id] - newu[id];
        s_blocksum[threadIdx.x] = diff*diff;
    }

    while(alive>1){
        __syncthreads(); // barrier (make sure s_blocksum is ready)
        alive /= 2;
        if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
    }

    if(t==0)
        blocksum[blockIdx.x] = s_blocksum[0];
}
```

OpenCL partial reduction kernel:
```c
__kernel void partialReduce(const int entries,
    ?? const double *u,
    ?? const double *newu,
    ?? double *blocksum){

    __local double s_blocksum[BDIM];
    const int id = get_global_id();
    int alive = ???;
    int t = ???;
    s_blocksum[t] = 0;

    // load global data into local memory if in range
    if(id < entries){
        const double diff = u[id] - newu[id];
        s_blocksum[t] = diff*diff;
    }

    while(alive>1){
        barrier(CLK_LOCAL_MEMFENCE); // barrier (make sure s_blocksum is ready)
        alive /= 2;
        if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
    }

    if(t==0)
        blocksum[get_group_id(0)] = s_blocksum[0];
}
```
Part 4b: Portability alternatives to OpenCL
Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Many-core: updates

There are additional options

Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Many-core: updates

There are additional options

Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming.
Many-core: updates

There are additional options

Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Many-core: updates

There are additional options

Intel CPU
AMD APU
Intel Xeon Phi
NVIDIA GPU
AMD Tahiti GPU
IBM Power 7
Altera FPGA

OpenCL

Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Many-core: porting from CUDA

Existing CUDA code can be ported to other frameworks.

Translation to intermediate languages happens at the level of source code or assembly code. However, this is predicated on using CUDA as the source language.
Portability Approaches: directives

Directive approach

- Use of optional [#pragma]'s to give compiler transformation hints
- Aims for portability, performance and programmability

- Introduced for accelerator support through directives (2012)
- Compilers with OpenACC support:

- OpenMP has been around for a while (1997)
- OpenMP 4.0 specifications (2013) includes accelerator support

```c
#pragma omp target teams distribute parallel for
for(int i = 0; i < N; ++i){
    y[i] = a*x[i] + y[i];
}
```

Code taken from: WHAT’S NEW IN OPENACC 2.0 AND OPENMP 4.0, GTC ‘14
Directive approach

- Not centralized anymore due to the offload model
- OpenACC and OpenMP begin to resemble an API rather than code decorations

Code taken from:
WHAT’S NEW IN OPENACC 2.0 AND OPENMP 4.0, GTC ‘14
My opinion on “Maturity” balanced against “Ease of use” for portable many-core programming.
Step Back: MPI + X?

Which “X” is going to dominate on-node threaded computing?

- MPI + MPI
- MPI + OpenMP
- MPI + pThreads
- MPI + CUDA
- MPI + OpenCL
- MPI + OpenACC
- MPI + TBB
- MPI + Cilk Plus
- MPI + ?

Should it even matter what “X” is?
Part 4c: OCCA
Open Concurrent Compute Abstraction
OCCA (Open Concurrent Compute Abstraction) is an open-source (MIT-license) library used to program current multi-core/many-core architectures. Devices (such as CPUs, GPUs, Intel's Xeon Phi, FPGAs, etc) are abstracted using an offload model for application development and programming for the devices is done.

Visit the OCCA GitHub repository at [github.com/libocca/occa](https://github.com/libocca/occa) for further information.
What does OCCA provide?

Open Concurrent Compute Architecture

API:
- Single, unified, library for programming heterogeneous devices.
- Flexibility at run-time to choose background thread programming model.
- Caching of compiled kernel binaries.

Portability:
- The OCCA-IR kernel language is portable across OpenMP, OpenCL, CUDA.
- The OCCA parser translates the OCCA Kernel Language (OKL) to the OCCA-IR.

Lightweight Framework:
- 50+30 KLOC (most auto-generated).
- Self-contained: dependencies are the backend compilers/drivers/runtimes.

Accessible:
- Exposes loop parallel paradigm … without an exotic programming model.
- C, C++, F90, Python … library front ends.
What does OCCA not do?

Open Concurrent Compute Architecture, no magic unicorns.

Auto-parallelize:
• Some programmer intervention is required to identify parallel for loops.

Auto-optimize:
• Programmer knowledge of architecture is still invaluable.

Auto-layout:
• The programmer needs to decide how data is arranged in memory.

Auto-distribute:
• You can use MPI+OCCA but you have to write the MPI code.
• We considered M-OCCA but it devolves quickly into a PGAS.

Low-level code:
• We do not circumvent the vendor compilers.
OCCA: easy portability

github.com/libocca/occa
libocca.org
Building the OCCA library:

git clone https://github.com/libocca/occa
cd occa
export OCCA_DIR=`pwd`
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OCCA_DIR/lib
make -j

Building example:

cd examples/addVector/cpp
make
./main

Try changing the threading model to OpenCL, CUDA, or OpenMP:

emacs main.cpp
### Portability: approaches of use

Numerous approaches to portability

<table>
<thead>
<tr>
<th>API</th>
<th>Type</th>
<th>Front-ends</th>
<th>Kernel</th>
<th>Back-ends</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kokkos</td>
<td>ND arrays</td>
<td>C++</td>
<td>Custom</td>
<td>CUDA &amp; OpenMP</td>
</tr>
<tr>
<td>VexCL</td>
<td>Vector class</td>
<td>C++</td>
<td>-</td>
<td>CUDA &amp; OpenCL</td>
</tr>
<tr>
<td>RAJA</td>
<td>Library</td>
<td>C++</td>
<td>C++ Lambdas</td>
<td>CUDA, OpenMP, OpenACC</td>
</tr>
<tr>
<td>OCCA</td>
<td>API, Source-to-source, Kernel Languages</td>
<td>C,C++,C#, F90, Python,MATLAB, Julia</td>
<td>OpenCL, CUDA, &amp; custom unified kernel language</td>
<td>CUDA, OpenCL, pThreads, OpenMP, Intel COI</td>
</tr>
<tr>
<td>CU2CL *</td>
<td>Source-to-source</td>
<td>App</td>
<td>CUDA</td>
<td>OpenCL</td>
</tr>
<tr>
<td>Insieme</td>
<td>Source-to-source compiler</td>
<td>C</td>
<td>OpenMP, Cilk, MPI, OpenCL</td>
<td>OpenCL, MPI, Insieme IR runtime</td>
</tr>
<tr>
<td>Trellis</td>
<td>Directives</td>
<td>C/C++</td>
<td>#pragma trellis</td>
<td>OpenMP, OpenACC, CUDA</td>
</tr>
<tr>
<td>OmpSs</td>
<td>Directives + kernels</td>
<td>C,C++</td>
<td>Hybrid OpenMP, OpenCL, CUDA</td>
<td>OpenMP, OpenCL, CUDA</td>
</tr>
<tr>
<td>Ocelot</td>
<td>PTX Translator</td>
<td>CUDA</td>
<td>CUDA</td>
<td>OpenCL</td>
</tr>
</tbody>
</table>

OCCA emphasis: lightweight and extensible.

* Wu Feng et al @ VT !
OCCA: OKL kernel language

Description

- Minimal extensions to C, familiar for regular programmers
- Explicit loops expose parallelism for modern multicore CPUs and accelerators
- Parallel loops are explicit through the fourth for-loop *inner* and *outer* labels

```
kernel void kernelName(...){
  ...
  for(int groupZ = 0; groupZ < zGroups; ++groupZ; outer2){
    for(int groupY = 0; groupY < yGroups; ++groupY; outer1){
      for(int groupX = 0; groupX < xGroups; ++groupX; outer0){  // Work-group implicit loops
        for(int itemZ = 0; itemZ < zItems; ++itemZ; inner2){
          for(int itemY = 0; itemY < yItems; ++itemY; inner1){
            for(int itemX = 0; itemX < xItems; ++itemX; inner0){  // Work-item implicit loops
              // GPU Kernel Scope
            }
          }
        }
      }
    }
  }
  ...
}
```

The concept of iterating over groups and items is simple
OCCA:OKL kernel language

Description

- Minimal extensions to C, familiar for regular programmers
- Explicit loops expose parallelism for modern multicore CPUs and accelerators
- Parallel loops are explicit through the fourth for-loop inner and outer labels

```cpp
kernel void kernelName(...)
{
    ... 
    for(int groupZ = 0; groupZ < zGroups; ++groupZ; outer2){
        for(int groupY = 0; groupY < yGroups; ++groupY; outer1){
            for(int groupX = 0; groupX < xGroups; ++groupX; outer0){ // Work-group implicit loops

                for(int itemZ = 0; itemZ < zItems; ++itemZ; inner2){
                    for(int itemY = 0; itemY < yItems; ++itemY; inner1){
                        for(int itemX = 0; itemX < xItems; ++itemX; inner0){ // Work-item implicit loops
                            // GPU Kernel Scope
                        }
                    }
                }
            }
        }
    }
    ...
} 
```
**OCCA:OKL kernel language**

**Description**

- Minimal extensions to C, familiar for regular programmers
- Explicit loops expose parallelism for modern multicore CPUs and accelerators
- Parallel loops are explicit through the fourth for-loop inner and outer labels

```c
kernel void kernelName(...){
...
for(int groupZ = 0; groupZ < zGroups; ++groupZ; outer2){
  for(int groupY = 0; groupY < yGroups; ++groupY; outer1){
    for(int groupX = 0; groupX < xGroups; ++groupX; outer0){
      // Work-group implicit loops
      for(int itemZ = 0; itemZ < zItems; ++itemZ; inner2){
        for(int itemY = 0; itemY < yItems; ++itemY; inner1){
          for(int itemX = 0; itemX < xItems; ++itemX; inner0){
            // Work-item implicit loops
            // GPU Kernel Scope
          }
        }
      }
    }
  }
}
```
Description

- Minimal extensions to C, familiar for regular programmers
- Explicit loops expose parallelism for modern multicore CPUs and accelerators
- Parallel loops are explicit through the fourth for-loop *inner* and *outer* labels

```c
kernel void kernelName(...) {
    ...
    for(int groupZ = 0; groupZ < zGroups; ++groupZ; outer2){
        for(int groupY = 0; groupY < yGroups; ++groupY; outer1){
            for(int groupX = 0; groupX < xGroups; ++groupX; outer0){
                // Work-group implicit loops
                for(int itemZ = 0; itemZ < zItems; ++itemZ; inner2){
                    for(int itemY = 0; itemY < yItems; ++itemY; inner1){
                        for(int itemX = 0; itemX < xItems; ++itemX; inner0){
                            // Work-item implicit loops
                            // GPU Kernel Scope
                        }
                    }
                }
            }
        }
    }
    ...
}
```

The concept of iterating over groups and items is simple

- `dim3 blockDim(xGroups, yGroups, zGroups);`
- `dim3 threadDim(xItems, yItems, zItems);`
- `kernelName<<<blockDim, threadDim>>>(...);`
Outer-loops

- **Outer-loops** are synonymous with CUDA and OpenCL kernels
- Extension: allow for multiple outer-loops per kernel

```c
kernel void kernelName(...){

...

for(int groupZ = 0; groupZ < zGroups; ++groupZ; outer2){
    for(int groupY = 0; groupY < yGroups; ++groupY; outer1){
        for(int groupX = 0; groupX < xGroups; ++groupX; outer0){ // Work-group implicit loops

            for(int itemZ = 0; itemZ < zItems; ++itemZ; inner2){
                for(int itemY = 0; itemY < yItems; ++itemY; inner1){
                    for(int itemX = 0; itemX < xItems; ++itemX; inner0){ // Work-item implicit loops
                        // GPU Kernel Scope
                    }
                }
            }
        }
    }
}

...
```

Data dependencies are found through a variable dependency graph
Outer-loops

- **Outer-loops** are synonymous with CUDA and OpenCL kernels
- Extension: allow for multiple outer-loops per kernel

```c
kernel void kernelName(...){
  ...

  for(outer){
    for(inner){
      }
  }

  ...
}
```
Outer-loops

- **Outer-loops** are synonymous with CUDA and OpenCL kernels
- Extension: allow for multiple outer-loops per kernel

```okl
kernel void kernelName(...)
{
...

  for(outer){
    for(inner){
  
  
  
  ...

  
}
}
```
Outer-loops

- **Outer-loops** are synonymous with CUDA and OpenCL kernels
- Extension: allow for multiple outer-loops per kernel

```okl
kernel void kernelName(...){
    for(outer){
        for(inner){
        }
    }

    for(outer){
        for(inner){
        }
    }

    for(outer){
        for(inner){
        }
    }
}
```
Outer-loops

- **Outer-loops** are synonymous with CUDA and OpenCL kernels
- Extension: allow for multiple outer-loops per kernel

```c
kernel void kernelName(...){
    for(outer){
        for(inner){
        }
    }

    for(outer){
        for(inner){
        }
    }

    for(outer){
        for(inner){
        }
    }
}
```

Data dependencies are found through a variable dependency graph
Outer-loops

- **Outer-loops** are synonymous with CUDA and OpenCL kernels
- Extension: allow for multiple outer-loops per kernel

```c
kernel void kernelName(...){
  if(expr){
    for(outer){
      for(inner){
    }
  }
  else{
    for(outer){
      for(inner){
    }
  }
}
while(expr){
  for(outer){
    for(inner){
  }
}
}
```
OCCA: OKL kernel language

Shared memory

```okl
for(int groupX = 0; groupX < xGroups; ++groupX; outer0){  // Work-group implicit loops
    shared int sharedVar[16];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){  // Work-item implicit loops
        sharedVar[itemX] = itemX;
    }

    // Auto-insert [barrier(localMemFence);]
}
```

Exclusive memory

```okl
for(int groupX = 0; groupX < xGroups; ++groupX; outer0){  // Work-group implicit loops
    exclusive int exclusiveVar, exclusiveArray[10];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){  // Work-item implicit loops
        exclusiveVar = itemX;  // Pre-fetch
    }

    // Auto-insert [barrier(localMemFence);]
}
```

Local barriers are auto-inserted
OCCA:OKL kernel language

Shared memory

```c
for(int groupX = 0; groupX < xGroups; ++groupX; outer0){ // Work-group implicit loops
    shared int sharedVar[16];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){ // Work-item implicit loops
        sharedVar[itemX] = itemX;
    }
}
```

// Auto-insert [barrier(localMemFence);]

```c
for(int itemX = 0; itemX < 16; ++itemX; inner0){ // Work-item implicit loops
    int i = (sharedVar[itemX] + sharedVar[(itemX + 1) % 16]);
}
```

Exclusive memory (similar to threadPrivate)

```c
for(int groupX = 0; groupX < xGroups; ++groupX; outer0){ // Work-group implicit loops
    exclusive int exclusiveVar, exclusiveArray[10];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){ // Work-item implicit loops
        exclusiveVar = itemX; // Pre-fetch
    }
}
```

// Auto-insert [barrier(localMemFence);]

```c
for(int itemX = 0; itemX < 16; ++itemX; inner0){ // Work-item implicit loops
    int i = exclusiveVar; // Use pre-fetched data
}
```

Local barriers are auto-inserted

Exclusive memory (similar to threadPrivate)

```c
for(int groupX = 0; groupX < xGroups; ++groupX; outer0){ // Work-group implicit loops
    exclusive int exclusiveVar, exclusiveArray[10];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){ // Work-item implicit loops
        exclusiveVar = itemX; // Pre-fetch
    }
}
```

// Auto-insert [barrier(localMemFence);]

```c
for(int itemX = 0; itemX < 16; ++itemX; inner0){ // Work-item implicit loops
    int i = exclusiveVar; // Use pre-fetched data
}
```
OCCA: OKL kernel language

Shared memory

for(int groupX = 0; groupX < xGroups; ++groupX; outer0){  // Work-group implicit loops
    shared int sharedVar[16];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){  // Work-item implicit loops
        sharedVar[itemX] = itemX;
    }

    // Auto-insert [barrier(localMemFence);]

    for(int itemX = 0; itemX < 16; ++itemX; inner0){  // Work-item implicit loops
        int i = (sharedVar[itemX] + sharedVar[(itemX + 1) % 16]);
    }
}

Exclusive memory (similar to threadPrivate)

for(int groupX = 0; groupX < xGroups; ++groupX; outer0){  // Work-group implicit loops
    exclusive int exclusiveVar, exclusiveArray[10];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){  // Work-item implicit loops
        exclusiveVar = itemX;  // Pre-fetch
    }

    // Auto-insert [barrier(localMemFence);]

    for(int itemX = 0; itemX < 16; ++itemX; inner0){  // Work-item implicit loops
        int i = exclusiveVar;  // Use pre-fetched data
    }
}

Local barriers are auto-inserted
OCCA: OKL kernel language

Shared memory

```c
for(int groupX = 0; groupX < xGroups; ++groupX; outer0){ // Work-group implicit loops
    shared int sharedVar[16];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){ // Work-item implicit loops
        sharedVar[itemX] = itemX;
    }

    // Auto-insert [barrier(localMemFence);]

    for(int itemX = 0; itemX < 16; ++itemX; inner0){ // Work-item implicit loops
        int i = (sharedVar[itemX] + sharedVar[(itemX + 1) % 16]);
    }
}
```

Exclusive memory (similar to threadPrivate)

```c
for(int groupX = 0; groupX < xGroups; ++groupX; outer0){ // Work-group implicit loops
    exclusive int exclusiveVar, exclusiveArray[10];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){ // Work-item implicit loops
        exclusiveVar = itemX; // Pre-fetch
    }

    // Auto-insert [barrier(localMemFence);]

    for(int itemX = 0; itemX < 16; ++itemX; inner0){ // Work-item implicit loops
        int i = exclusiveVar; // Use pre-fetched data
    }
}
```
OCCA: OKL kernel language

Shared memory

```c
for(int groupX = 0; groupX < xGroups; ++groupX; outer0){  // Work-group implicit loops
    shared int sharedVar[16];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){  // Work-item implicit loops
        sharedVar[itemX] = itemX;
    }

    // Auto-insert [barrier(localMemFence)];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){  // Work-item implicit loops
        int i = (sharedVar[itemX] + sharedVar[(itemX + 1) % 16]);
    }
}
```

Exclusive memory (similar to threadPrivate)

```c
exclusiveVar = 0
exclusiveVar = 1
exclusiveVar = 2
.
.
.

for(int groupX = 0; groupX < xGroups; ++groupX; outer0){  // Work-group implicit loops
    exclusive int exclusiveVar, exclusiveArray[10];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){  // Work-item implicit loops
        exclusiveVar = itemX;  // Pre-fetch
    }

    // Auto-insert [barrier(localMemFence)];

    for(int itemX = 0; itemX < 16; ++itemX; inner0){  // Work-item implicit loops
        int i = exclusiveVar;  // Use pre-fetched data
    }
}
```

Local barriers are auto-inserted
OpenCL/CUDA to OCCA IR

Description

- Parser can translate OpenCL/CUDA kernels to OCCA IR*
- Although OCCA IR was derived from the GPU model, there are complexities

Since we derived OCCA IR from the GPU model, the inverse should be easy ... right?
```cpp
#include <iostream>
#include "occa.hpp"

int main(int argc, char **argv) {
    float *a = new float[N];
    float *b = new float[N];
    float *ab = new float[N];

    for(int i = 0; i < N; ++i) {
        a[i] = i;
        b[i] = 1 - i;
        ab[i] = 0;
    }

    occa::device device;
    occa::kernel addVectors;
    occa::memory o_a, o_b, o_ab;

    device.setup("mode = OpenCL , platformID = 0, deviceID = 0");

    o_a = device.malloc(N*sizeof(float));
    o_b = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

    o_a.copyFrom(a);
    o_b.copyFrom(b);

    addVectors = device.buildKernelFromSource("addVectors.okl", "addVectors");

    addVectors(N, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
        std::cout << i << " : " << ab[i] << '
';
}
```
#include <iostream>

```
#include "occa.hpp"

int main(int argc, char **argv){
    float *a = new float[N];
    float *b = new float[N];
    float *ab = new float[N];

    for(int i = 0; i < N; ++i){
        a[i] = i;
        b[i] = 1 - i;
        ab[i] = 0;
    }

    occa::device device;
    occa::kernel addVectors;
    occa::memory o_a, o_b, o_ab;

    device.setup("mode = OpenCL , platformID = 0, deviceID = 0");

    o_a = device.malloc(N*sizeof(float));
    o_b = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

    o_a.copyFrom(a);
    o_b.copyFrom(b);

    addVectors = device.buildKernelFromSource("addVectors.okl", "addVectors");

    addVectors(N, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
        std::cout << i << " : " << ab[i] << 'n';
```
```cpp
#include <iostream>
#include "occa.hpp"

int main(int argc, char **argv){
  float *a  = new float[N];
  float *b  = new float[N];
  float *ab = new float[N];

  for(int i = 0; i < N; ++i){
    a[i]  = i;
    b[i]  = 1 - i;
    ab[i] = 0;
  }

  occa::device device;
  occa::kernel addVectors;
  occa::memory o_a, o_b, o_ab;

  device.setup("mode = OpenCL , platformID = 0, deviceID = 0");

  o_a  = device.malloc(N*sizeof(float));
  o_b  = device.malloc(N*sizeof(float));
  o_ab = device.malloc(N*sizeof(float));

  o_a.copyFrom(a);
  o_b.copyFrom(b);
  addVectors = device.buildKernelFromSource("addVectors.okl", "addVectors");
  addVectors(N, o_a, o_b, o_ab);
  o_ab.copyTo(ab);

  for(int i = 0; i < 5; ++i)
    std::cout << i << " : " << ab[i] << '\n';
}
```
#include <iostream>
#include "occa.hpp"

int main(int argc, char **argv) {
    float *a = new float[N];
    float *b = new float[N];
    float *ab = new float[N];

    for (int i = 0; i < N; ++i) {
        a[i] = i;
        b[i] = 1 - i;
        ab[i] = 0;
    }

    occa::device device;
    occa::kernel addVectors;
    occa::memory o_a, o_b, o_ab;

    device.setup("mode = OpenCL , platformID = 0, deviceID = 0");

    o_a = device.malloc(N*sizeof(float));
    o_b = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

    o_a.copyFrom(a);
    o_b.copyFrom(b);

    addVectors = device.buildKernelFromSource("addVectors.okl", "addVectors");

    addVectors(N, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

    for (int i = 0; i < 5; ++i)
        std::cout << i << " : " << ab[i] << 'n';
```cpp
#include <iostream>
#include "occa.hpp"

int main(int argc, char **argv){
    float *a = new float[N];
    float *b = new float[N];
    float *ab = new float[N];

    for(int i = 0; i < N; ++i){
        a[i] = i;
        b[i] = 1 - i;
        ab[i] = 0;
    }

    occa::device device;
    occa::kernel addVectors;
    occa::memory o_a, o_b, o_ab;

    device.setup("mode = OpenCL , platformID = 0, deviceID = 0");

    o_a = device.malloc(N*sizeof(float));
    o_b = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

    o_a.copyFrom(a);
    o_b.copyFrom(b);
    addVectors = device.buildKernelFromSource("addVectors.okl", "addVectors");

    addVectors(N, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
        std::cout << i << " : " << ab[i] << 'n';
}
```
#include <iostream>
#include "occa.hpp"

int main(int argc, char **argv){
    float *a = new float[N];
    float *b = new float[N];
    float *ab = new float[N];

    for(int i = 0; i < N; ++i){
        a[i] = i;
        b[i] = 1 - i;
        ab[i] = 0;
    }

    occa::device device;
    occa::kernel addVectors;
    occa::memory o_a, o_b, o_ab;

    device.setup("mode = OpenCL , platformID = 0, deviceID = 0");

    o_a  = device.malloc(N*sizeof(float));
    o_b  = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

    o_a.copyFrom(a);
    o_b.copyFrom(b);

    addVectors = device.buildKernelFromSource("addVectors.okl", "addVectors");

    addVectors(N, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
        std::cout << i << " : " << ab[i] << '
';
```cpp
#include <iostream>
#include "occa.hpp"

int main(int argc, char **argv){
    float *a  = new float[N];
    float *b  = new float[N];
    float *ab = new float[N];

    for(int i = 0; i < N; ++i){
      a[i]  = i;
      b[i]  = 1 - i;
      ab[i] = 0;
    }

    occa::device device;
    occa::kernel addVectors;
    occa::memory o_a, o_b, o_ab;

    device.setup("mode = OpenCL , platformID = 0, deviceID = 0");

    o_a  = device.malloc(N*sizeof(float));
    o_b  = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

    o_a.copyFrom(a);
    o_b.copyFrom(b);

    addVectors = device.buildKernelFromSource("addVectors.okl", "addVectors");

    addVectors(N, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
      std::cout << i << " : " << ab[i] << "\n";
}
```
OCCA: example adding two vectors

```cpp
#include <iostream>
#include "occa.hpp"

int main(int argc, char **argv)
{
    float *a = new float[N];
    float *b = new float[N];
    float *ab = new float[N];

    for(int i = 0; i < N; ++i)
    {
        a[i] = i;
        b[i] = 1 - i;
        ab[i] = 0;
    }

    occa::device device;
    occa::kernel addVectors;
    occa::memory o_a, o_b, o_ab;

    device.setup("mode = OpenCL , platformID = 0, deviceID = 0");

    o_a = device.malloc(N*sizeof(float));
    o_b = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

    o_a.copyFrom(a);
    o_b.copyFrom(b);

    addVectors = device.buildKernelFromSource("addVectors.okl", "addVectors");

    addVectors(N, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
    {
        std::cout << i << " : " << ab[i] << 'n';
    }
}
```
#include <iostream>
#include "occa.hpp"

int main(int argc, char **argv){
    float *a  = new float[N];
    float *b  = new float[N];
    float *ab = new float[N];

    for(int i = 0; i < N; ++i){
        a[i] = i;
        b[i] = 1 - i;
        ab[i] = 0;
    }

    occa::device device;
    occa::kernel addVectors;
    occa::memory o_a, o_b, o_ab;

    device.setup("mode = OpenCL , platformID = 0, deviceID = 0");

    o_a  = device.malloc(N*sizeof(float));
    o_b  = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

    o_a.copyFrom(a);
    o_b.copyFrom(b);

    addVectors = device.buildKernelFromSource("addVectors.okl", "addVectors");

    addVectors(N, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
        std::cout << i << " : " << ab[i] << '
';
```cpp
#include <iostream>
#include "occa.hpp"

int main(int argc, char **argv)
{
    int N = 50;
    float *a = new float[N];
    float *b = new float[N];
    float *ab = new float[N];

    for(int i = 0; i < N; ++i)
    {
        a[i] = i;  
        b[i] = 1 - i;  
        ab[i] = 0;
    }

    occa::device device;
    occa::kernel addVectors;
    occa::memory o_a, o_b, o_ab;

    device.setup("OpenCL", 0, 0);  // (Platform, Device) = (0, 0)

    o_a  = device.malloc(N*sizeof(float));
    o_b  = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

    o_a.copyFrom(a);
    o_b.copyFrom(b);

    addVectors = device.buildKernelFromSource("addVectors.okl", "addVectors");

    addVectors(N, o_a, o_b, o_ab);
    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
    {
        std::cout << i << " \n";
    }
}
```
OCCA: example adding two vectors

```cpp
#include <iostream>
#include "occa.hpp"

int main(int argc, char **argv)
{
    int N = 50;
    float *a = new float[N];
    float *b = new float[N];
    float *ab = new float[N];

    for(int i = 0; i < N; ++i)
    {
        a[i] = i;
        b[i] = 1 - i;
        ab[i] = 0;
    }

    occa::device device;
    occa::kernel addVectors;
    occa::memory o_a, o_b, o_ab;

    device.setup("OpenCL", 0, 0); // (Platform, Device) = (0, 0)

    o_a = device.malloc(N*sizeof(float));
    o_b = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

    o_a.copyFrom(a);
    o_b.copyFrom(b);

    addVectors = device.buildKernelFromSource("addVectors.okl", "addVectors");

    addVectors(N, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
    {
        std::cout << "\n";
        std::cout << i << " \n";
    }
}
```

Kernel code:
```
kernl void addVectors(const int entries, const float *a,
                      const float *b,
                      float *ab)
{
    for(int b=0;b<entries;b+=10;outer0)
    {
        for(int n=b;n<b+10;++n;inner0)
        {
            if(n < entries)
                ab[n] = a[n] + b[n];
        }
    }
}
```
Native Host Codes

All the HOST codes use the same kernel.

Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors
All the HOST codes use the same kernel.

Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors
All the HOST codes use the same kernel.

Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors
Native Host Codes

All the HOST codes use the same kernel.

Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors
All the HOST codes use the same kernel.

Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors
All the HOST codes use the same kernel.
Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors
In OCCA we split the i and j loops both into outer and inner loops. From the OCCA kernel we can reproduce the serial, CUDA, and OpenCL kernels (also pthreads, openmp...).
In OCCA we split the i and j loops both into outer and inner loops. From the OCCA kernel we can reproduce the serial, CUDA, and OpenCL kernels (also pthreads, openmp...).
In OCCA we split the i and j loops both into outer and inner loops.

From the OCCA kernel we can reproduce the serial, CUDA, and OpenCL kernels (also pthreads, openmp...).
Online Compilation

Source-to-Source Compilation

• Extended C and Fortran to expose parallelism, making use of the OCCA IR

Custom compilation tools tailored for code manipulation and analysis
Source-to-Source Compilation

- Extended C and Fortran to expose parallelism & make use of OCCA IR

---

```
#define N 10

int i = N;
```

---

Custom compilation tools tailored for code manipulation and analysis
OCCA Infrastructure

Source-to-Source Compilation

- Extended **C** and **Fortran** to expose parallelism & make use of OCCA IR

```
#define N 10

int i = N;
```

```
int i = 10;
```

Custom compilation tools tailored for code manipulation and analysis
OCCA Infrastructure

Source-to-Source Compilation

- Extended C and Fortran to expose parallelism & make use of OCCA IR

Source-to-Source Compilation

- Extended C and Fortran to expose parallelism & make use of OCCA IR

Custom compilation tools tailored for code manipulation and analysis
OCCA Infrastructure

Source-to-Source Compilation

- Extended C and Fortran to expose parallelism & make use of OCCA IR

Custom compilation tools tailored for code manipulation and analysis
OCCA Infrastructure

Source-to-Source Compilation
- Extended C and Fortran to expose parallelism & make use of OCCA IR

Preprocessor → Tokenize → Initial AST → OpenCL/CUDA Setup → Transform OCCA Loops → Prototypes, Barriers, Arguments → Split Kernels, Setup Work Dimensions → Kernel Analysis

Custom compilation tools tailored for code manipulation and analysis
OCCA Infrastructure

Source-to-Source Compilation

- Extended C and Fortran to expose parallelism & make use of OCCA IR

Custom compilation tools tailored for code manipulation and analysis
In this exercise you will create a flow simulation

Work in teams of two.
Hands On #3: OCCA flow simulation

In this exercise you will create a flow simulation

Work in teams of two.
#1. build the OCCA library:

- clone the OCCA repo
  
git clone https://github.com/libocca/occa

- change into the OCCA directory
  
cd occa

- build OCCA
  
make -j

- add OCCA_DIR to env
  
export OCCA_DIR=`pwd`

#2. build the OCCA LBM code:

- clone the ATPESC17 repo
  
git clone https://github.com/tcew/ATPESC17

- change into the lbm directory
  
cd ATPESC17/lbm

- build OCCA lbm solver
  
make -f makefile.occa

#3. save a png format image with white background to the lbm directory:

- run the lbm code with your png image
  
# (using 400 as a flow volume threshold)

./occaLBM yourImageName.png 400
The image pixels become flow nodes in a lattice: the Lattice Boltzmann Method tracks the density of 9 species of colliding particles constrained to move on the lattice.

Details: a D2Q9 lattice Boltzmann method is implemented using an OCCA update kernel that uses a single thread to updates the 9 particle densities at each lattice node.
#5. the lbm code generates bah#####.png image files:

# To make a movie:
ffmpeg -r 24 -i bah%06d.png -b:v 16384k -vf scale=1024:-1 foo.mp4

#6. transfer foo.mp4 to your laptop via globus and open with movie player:

Raise your hand and demo your movie when done.
#5. the lbm code generates bah#####.png image files:

# To make a movie:
ffmpeg -r 24 -i bah%06d.png -b:v 16384k -vf scale=1024:-1 foo.mp4

#6. transfer foo.mp4 to your laptop via globus and open with movie player:

Raise your hand and demo your movie when done.
OCCA Flow: changing thread model

The lbm code is set up to use CUDA by default.

#7. Find out what compute modes are available:

$OCCA_DIR/bin/occainfo

#8. change OCCA device setup in main to change the thread model:

```cpp
occa::device device;
// device.setup("mode=OpenCL, deviceID=1, platformID=0");
device.setup("mode=CUDA, deviceID=0");
// device.setup("mode=OpenMP");
```

#9. re-make the executable:

make -f makefile.occa

#10. rerun

./occaLBM yourImageName.png 400

# Do you notice a speed change?

#11. try installing and running on your laptop - this might be tricky.

Congratulations: you have found out how easy it can be to switch compute mode with OCCA.
Part 4d: OCCA Application Examples
OCCA: apps & benchmarks

High-order finite difference for Reverse Time Migration (imaging algorithm)

- Original OpenMP
- OCCA:OpenMP (CPU-kernel)
- OCCA:OpenMP (GPU-kernel)

- Original CUDA
- OCCA::CUDA (GPU-kernel)
- OCCA::OpenCL (GPU-kernel)
- OCCA::CUDA (CPU-kernel)
- OCCA::OpenCL (CPU-kernel)

OpenMP: Intel Xeon CPU E5-2640
OpenCL/CUDA: NVIDIA Tesla K10
OCCA: apps & benchmarks

Discontinuous Galerkin for RTM

![Graph showing estimated GFLOPS for different polynomial orders and OpenCL and CUDA implementations.](image)
**OCCA: apps & benchmarks**

Lattice Boltzmann Method in Core Sample Analysis

Comparison across platforms (Normalized with original code)

<table>
<thead>
<tr>
<th>API Mode</th>
<th>Device</th>
<th>Model</th>
<th>Wall Clock</th>
<th>BW (GB/s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref dense code</td>
<td>CPU</td>
<td>Intel i7-5960X</td>
<td>1290</td>
<td>—</td>
<td>x 1</td>
</tr>
<tr>
<td>[-O3 in gcc 4.8]</td>
<td>CPU</td>
<td>Intel i7-5960X</td>
<td>11.12</td>
<td>22</td>
<td>x 116</td>
</tr>
<tr>
<td>OpenMP</td>
<td>CPU</td>
<td>Intel i7-5960X</td>
<td>11.18</td>
<td>22</td>
<td>x 115</td>
</tr>
<tr>
<td>OpenCL: Intel</td>
<td>CPU</td>
<td>Intel i7-5960X</td>
<td>1.39</td>
<td>176</td>
<td>x 928</td>
</tr>
<tr>
<td>OpenCL: AMD</td>
<td>GPU</td>
<td>AMD 7990</td>
<td>1.25</td>
<td>196</td>
<td>x 1032</td>
</tr>
<tr>
<td>OpenCL: NVIDIA</td>
<td>GPU</td>
<td>GTX 980</td>
<td>1.20</td>
<td>205</td>
<td>x 1075</td>
</tr>
<tr>
<td>CUDA: NVIDIA</td>
<td>GPU</td>
<td>GTX 980</td>
<td>1.20</td>
<td>205</td>
<td>x 1075</td>
</tr>
</tbody>
</table>

Comparison across platforms (Normalized with OCCA::OpenMP)

<table>
<thead>
<tr>
<th>API Mode</th>
<th>Device</th>
<th>Model</th>
<th>Wall Clock</th>
<th>BW (GB/s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP</td>
<td>CPU</td>
<td>Intel i7-5960X</td>
<td>11.12</td>
<td>22</td>
<td>x 1.0</td>
</tr>
<tr>
<td>OpenCL: Intel</td>
<td>CPU</td>
<td>Intel i7-5960X</td>
<td>11.18</td>
<td>22</td>
<td>x 1.0</td>
</tr>
<tr>
<td>OpenCL: AMD</td>
<td>GPU</td>
<td>AMD 7990</td>
<td>1.39</td>
<td>176</td>
<td>x 8.0</td>
</tr>
<tr>
<td>OpenCL: NVIDIA</td>
<td>GPU</td>
<td>GTX 980</td>
<td>1.25</td>
<td>196</td>
<td>x 8.9</td>
</tr>
<tr>
<td>CUDA: NVIDIA</td>
<td>GPU</td>
<td>GTX 980</td>
<td>1.20</td>
<td>205</td>
<td>x 9.3</td>
</tr>
</tbody>
</table>
Applications

Discontinuous Galerkin for shallow water equations

<table>
<thead>
<tr>
<th>Polynomial Order</th>
<th>Compute-Time vs Real-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x650</td>
</tr>
<tr>
<td>2</td>
<td>x208</td>
</tr>
<tr>
<td>3</td>
<td>x95</td>
</tr>
<tr>
<td>4</td>
<td>x47</td>
</tr>
</tbody>
</table>
Applications

Discontinuous Galerkin for shallow water equations

<table>
<thead>
<tr>
<th>Polynomial Order</th>
<th>Compute-Time vs Real-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x650</td>
</tr>
<tr>
<td>2</td>
<td>x208</td>
</tr>
<tr>
<td>3</td>
<td>x95</td>
</tr>
<tr>
<td>4</td>
<td>x47</td>
</tr>
</tbody>
</table>

Million Nodes per second

- OCCA:CL, K40
- OCCA:CUDA, K40
- OCCA:CL, Tahiti
- OCCA:CL, K40
- OCCA:CUDA, K40
- OCCA:CL, Tahiti

Polynomial Order

- Compute-Time vs Real-Time

- Intel i7
- OCCA:CL
- OCCA:OpenMP, Intel i7
OCCA: apps & benchmarks

Algebraic multigrid for elliptic problems

Setup Time

Solve Time

# of unknowns

M. unknowns per sec

CUDA on Titan
OpenCL on Titan
OpenCL on Tahiti
OpenCL on Intel i7
OpenMP on Intel i7
OCCA: apps & benchmarks

Monte Carlo for neutronics
Collaborations with Argonne National Lab

XSBench

RSBench

OpenMP: Intel Xeon CPU E5-2650
OpenCL/CUDA: NVIDIA Tesla K20c
OCCA: apps & benchmarks

Three of our ported Rodinia benchmarks, based on the “11 Dwarves”

Backprop

BFS

Needleman

OpenMP : Intel Xeon CPU E5-2650
OpenCL/CUDA : NVIDIA Tesla K20c

https://github.com/dmed256/OCCA-Benchmarks