Accelerators

Argonne Training Program on Extreme-Scale Computing 2014

Tim Warburton
Professor of Computational & Applied Mathematics @ Rice University
Reality Check

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

It takes more than 3 hours to master GPUs…

… but we can discuss some of the basics …
Reality Check

It takes more than 3 hours to master GPUs…

… but we can discuss some of the basics …

… there are many web resources …

Seems like 21 hours of fun:
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.
git clone https://github.com/tcew/ATPESC14
Accelerators: slides & repos

- git clone https://github.com/tcew/ATPESC14
- git clone https://github.com/tcew/OCCA2
Accelerators: slides & repos

Slides: http://www.caam.rice.edu/~timwar/ATPESC14/

```bash
git clone https://github.com/tcew/ATPESC14
```

```bash
git clone https://github.com/tcew/OCCA2
```
Overview

Part 0: Background on instructor.

Part 1: Processor architecture trends.

- CPU v. GPU.

Part 2: CUDA

- NVIDIA’s threaded offload programming model.
- Poisson solver example.

Part 3: Interlude on CUDA optimization.

Part 4: OpenCL

- Open Computing Language

Part 5: OCCA

- Unified and extensible many-core programming model.
All are GPU enabled and are building accelerated micro/mini-apps.

Q: which many-core programming model(s) should they learn?
Advanced SCALable Simulations

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

Methods
Numerical Analysis
Accelerate
Scale

Electromagnetics
Shallow water
Fluid & gas dynamics
Seismic

All Multi-GPU

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

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

Methods
Numerical Analysis
Accelerate
Scale

Electromagnetics
Shallow water
Fluid & gas dynamics
Seismic

High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers.
GPU programming tools & applications. Industrial collaboration.
Part 1: Processor Trends

CPU v. GPU
Design goals for CPUs:

- Make a single thread very fast.
- Reduce latency through large caches.
- Predict, speculate.
CPU: simplified core architecture

A loose diagramatic representation of a basic processing core.

Core: minimally a control unit with an independent instruction stream, arithmetic logic units, register, memory interface.
Everything synchronizes to the Clock.

Control Unit (“CU”): The brains of the operation. Everything connects to it.

CU controls gates, tells other units about ‘what’ and ‘how’:

- What operation?
- Which register?
  - Which addressing mode?

Bus entries/exits are gated and (potentially) buffered.

From: NYU Lecture notes by Klöckner
An ALU is not a core.

ALUs are the basic compute units on a core.

One or two operands A, B

Operation selector (Op):

- (Integer) Addition, Subtraction.
- (Logical) And, Or, Not.
- (Bitwise) Shifts.
- (Integer) Multiplication, Division.

Specialized ALUs:

- **Floating Point** Unit (FPU).
- Address ALU.

Operates on *binary representations of numbers.*
CPU: abstract modern architecture

Modern “CPU-Style” core design emphasizes individual thread performance.

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

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
CPU: example die partition

Older single core processor example.

Die floorplan: VIA Isaiah (2008). 65 nm, 4 SP ops at a time, 1 MiB L2.
Only a small portion of the chip footprint is devoted to computation. Most of the silicon is devoted to managing the flow of instructions.
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.

GPU: massively parallel compute

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.

GPU: early example

Die floorplan: AMD RV770 (2008) 55 nm, 800 SP ops at a time
The majority of the silicon is devoted to computation

http://www.anandtech.com/show/2556/8
GPU: slim down the core

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

- Instruction Fetch/Decode
- ALU (Execute)
- Execution context
- Out-of-order control logic
- Branch predictor logic
- Memory pre fetch unit
- 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.
GPU: slim down the core

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.
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
GPU: reduce # of instruction streams

The Fetch/Decode units in those 16 clone cores are likely fetching and decoding the same instruction streams for rendering graphics...

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
GPU: SIMD groups

“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.
GPU: multiple independent cores

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

The cores are cloned into a massively parallel processor.

This is an abstract GPU [ memory system not shown ]

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
One instruction unit per core impacts instruction branching.

Idle ALUs execute NOPs when code branches. In this example we could see 1/8 of maximum performance.
GPU: high latency memory fetches

Accessing off-chip GPU memory takes a long time (partly latency)

- Work on registers;
- Work on registers;
- Work on registers;

Load registers from main memory;

It takes $O(1000)$ cycles to load data from off-chip memory into the SM registers file.

These ALUs are idled (stalled) after a load.
GPU: multiple contexts per core

Fast hardware context switching can help hide high memory latency

Idea: allocate enough registers and shared memory to allow for multiple contexts

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
GPU: high latency memory fetches

We can hide latency if the core can switch SIMD contexts

<table>
<thead>
<tr>
<th>Time</th>
<th>ALU 1</th>
<th>ALU 2</th>
<th>ALU 3</th>
<th>ALU 4</th>
<th>ALU 5</th>
<th>ALU 6</th>
<th>ALU 7</th>
<th>ALU 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Ctx1: work on registers;
Ctx1: work on registers;
Ctx1: work on registers;
Ctx1: load request, switch context;

Ctx3: work on registers;
Ctx3: work on registers;
Ctx3: work on registers;
Ctx3: load request, switch context;

Ctx2: work on registers;
Ctx2: work on registers;
Ctx2: work on registers;
Ctx2: load request, switch context;

Ctx1: load done so continue

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
The amount of latency hiding possible depends on how many contexts can be simultaneously resident on the core.

GPU: slow threads ok

By design it is ok if a threads stall as long as we can switch to other threads

This context takes longer to complete

The stall time is recovered by switching to other contexts

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian
GPU: summary of architecture

Summary of multi-level GPU parallel architecture

- Multiple cores.
- Each core has one (or more) wide SIMD vector units.
  - Each wide SIMD vector unit executes one instruction stream.
- Each core has a pool of shared memory.
- Each core hardware can switch between multiple contexts to hide memory latency.
- Branching code involves partial serialization.

* SIMD here is the number of ALUs in one of the core's vector unit. *
GPU: natural thread model
The GPU architecture admits a natural parallel threading model

- Programmer divides a compute task 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 (ALU)
      - 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 (blocking context resources).
  - Main assumption: same instructions for independent work-groups.

* SIMD here is the number of ALUs in one of the core’s vector unit.
### Accelerators: Big Iron Computing

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Super Computer Center in Guangzhou</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi</td>
<td>3120000</td>
<td>33862.7</td>
<td>54902.4</td>
<td>17808</td>
</tr>
<tr>
<td>2</td>
<td>DOE/SC/Oak Ridge National Laboratory United States</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x</td>
<td>560640</td>
<td>17590.0</td>
<td>27112.5</td>
<td>8209</td>
</tr>
<tr>
<td>3</td>
<td>DOE/NNSA/LLNL United States</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM</td>
<td>1572864</td>
<td>17173.2</td>
<td>20132.7</td>
<td>7890</td>
</tr>
<tr>
<td>4</td>
<td>RIKEN Advanced Institute for Computational Science (AICS)</td>
<td>K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu</td>
<td>705024</td>
<td>10510.0</td>
<td>11280.4</td>
<td>12660</td>
</tr>
<tr>
<td>5</td>
<td>DOE/SC/Argonne National Laboratory United States</td>
<td>Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM</td>
<td>786432</td>
<td>8586.6</td>
<td>10066.3</td>
<td>3945</td>
</tr>
<tr>
<td>6</td>
<td>Swiss National Supercomputing Centre (CSCS)</td>
<td>Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect, NVIDIA K20x</td>
<td>115984</td>
<td>6271.0</td>
<td>7788.9</td>
<td>2325</td>
</tr>
<tr>
<td>7</td>
<td>Texas Advanced Computing Center/Univ. of Texas</td>
<td>Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Aries FDR, Intel Xeon Phi SE10P</td>
<td>462462</td>
<td>5168.1</td>
<td>8520.1</td>
<td>4510</td>
</tr>
<tr>
<td>8</td>
<td>Forschungszentrum Juelich (FZJ) Germany</td>
<td>JUQUEEN - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect</td>
<td>458752</td>
<td>5008.9</td>
<td>5872.0</td>
<td>2301</td>
</tr>
<tr>
<td>9</td>
<td>DOE/NNSA/LLNL United States</td>
<td>Vulcan - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect</td>
<td>393216</td>
<td>4293.3</td>
<td>5033.2</td>
<td>1972</td>
</tr>
<tr>
<td>10</td>
<td>Government United States</td>
<td>Cray XC30, Intel Xeon E5-2697v2 12C 2.7GHz, Aries interconnect</td>
<td>225984</td>
<td>3143.5</td>
<td>4881.3</td>
<td>4881.3</td>
</tr>
<tr>
<td>11</td>
<td>Exploration &amp; Production - Eni S.p.A. Italy</td>
<td>HPC2 - iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, NVIDIA K20x</td>
<td>62640</td>
<td>3003.0</td>
<td>4006.3</td>
<td>1067</td>
</tr>
<tr>
<td>12</td>
<td>Leibniz Rechenzentrum Germany</td>
<td>SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR</td>
<td>147456</td>
<td>2897.0</td>
<td>3185.1</td>
<td>3423</td>
</tr>
<tr>
<td>13</td>
<td>GSIC Center, Tokyo Institute of Technology Japan</td>
<td>TSUBAME 2.5 - Cluster Platform SL390s G7, Xeon X5670 6C 2.93GHz, Infiniband QDR, NVIDIA K20x</td>
<td>76032</td>
<td>2785.0</td>
<td>5735.7</td>
<td>1399</td>
</tr>
<tr>
<td>14</td>
<td>National Supercomputing Center in Tianjin China</td>
<td>Tianhe-1A - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050</td>
<td>186368</td>
<td>2566.0</td>
<td>4701.0</td>
<td>4040</td>
</tr>
<tr>
<td>15</td>
<td>DOE/SC/Pacific Northwest National Laboratory</td>
<td>cascade - Atipa Visione IF442 Blade Server, Xeon E5-2670 8C 2.600GHz, Infiniband FDR, Intel Xeon Phi 5110P</td>
<td>194616</td>
<td>2539.1</td>
<td>3388.0</td>
<td>1384</td>
</tr>
<tr>
<td>16</td>
<td>Total Exploration Production France</td>
<td>Pangea - SGI ICE X, Xeon E5-2670 8C 2.600GHz, Infiniband FDR</td>
<td>110400</td>
<td>2098.1</td>
<td>2296.3</td>
<td>2118</td>
</tr>
</tbody>
</table>

*top500.org: 06/2014 list of large scale systems.*
The GPU architecture admits a natural parallel threading model. The following table presents the details of various supercomputers:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System Details</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Super Computer Center in Guangzhou</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi</td>
<td>3400000</td>
<td>33862.7</td>
<td>54902.4</td>
<td>17808</td>
</tr>
<tr>
<td>2</td>
<td>DOE/SC/Oak Ridge National Laboratory</td>
<td>United States - Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x</td>
<td>560640</td>
<td>17590.0</td>
<td>27112.5</td>
<td>8209</td>
</tr>
<tr>
<td>3</td>
<td>DOE/NNSA/LLNL</td>
<td>United States - Sequoia - BlueGene/Q, Power BQC 16C 1.600GHz, Custom IBM</td>
<td>1572864</td>
<td>17173.2</td>
<td>20132.7</td>
<td>7890</td>
</tr>
<tr>
<td>4</td>
<td>RIKEN Advanced Institute for Computational Science (AICS)</td>
<td>K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect - Fujitsu</td>
<td>705024</td>
<td>10510.0</td>
<td>11280.4</td>
<td>12660</td>
</tr>
<tr>
<td>5</td>
<td>DOE/SC/Argonne National Laboratory</td>
<td>United States - Mira - BlueGene/Q, Power BQC 16C 1.600GHz, Custom IBM</td>
<td>786432</td>
<td>8586.6</td>
<td>10066.3</td>
<td>3945</td>
</tr>
<tr>
<td>6</td>
<td>Swiss National Supercomputing Centre</td>
<td>Switzerland - Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect, NVIDIA K20x</td>
<td>115984</td>
<td>6271.0</td>
<td>7788.9</td>
<td>2325</td>
</tr>
<tr>
<td>7</td>
<td>Texas Advanced Computing Center/Univ. of Texas</td>
<td>Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P</td>
<td>462462</td>
<td>5168.1</td>
<td>8520.1</td>
<td>4510</td>
</tr>
<tr>
<td>8</td>
<td>Forschungszentrum Juelich (FZJ)</td>
<td>Germany - JUQUEEN - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect</td>
<td>458752</td>
<td>5008.9</td>
<td>5872.0</td>
<td>2301</td>
</tr>
<tr>
<td>9</td>
<td>DOE/NNSA/LLNL</td>
<td>United States - Vulcan - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect</td>
<td>393216</td>
<td>4293.3</td>
<td>5033.2</td>
<td>1972</td>
</tr>
<tr>
<td>10</td>
<td>Government</td>
<td>United States - Pangea - Cray XC30, Intel Xeon E5-2697v2 12C 2.7GHz, Aries interconnect</td>
<td>225984</td>
<td>3143.5</td>
<td>4881.3</td>
<td>1067</td>
</tr>
<tr>
<td>11</td>
<td>Exploration &amp; Production - Eni S.p.A.</td>
<td>Italy - HPC2 - iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, NVIDIA K20x</td>
<td>62640</td>
<td>3003.0</td>
<td>4006.3</td>
<td>1067</td>
</tr>
<tr>
<td>12</td>
<td>Leibniz Rechenzentrum</td>
<td>Germany - SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.700GHz, Infiniband FDR</td>
<td>147456</td>
<td>2897.0</td>
<td>3185.1</td>
<td>3423</td>
</tr>
<tr>
<td>13</td>
<td>GSIC Center, Tokyo Institute of Technology</td>
<td>Japan - TSUBAME 2.5 - Cluster Platform SL390s G7, Xeon X5670 6C 2.93GHz, Infiniband QDR, NVIDIA K20x</td>
<td>76032</td>
<td>2785.0</td>
<td>5735.7</td>
<td>1399</td>
</tr>
<tr>
<td>14</td>
<td>National Supercomputing Center in Tianjin</td>
<td>China - Tianhe-1A - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050</td>
<td>186368</td>
<td>2566.0</td>
<td>4701.0</td>
<td>4040</td>
</tr>
<tr>
<td>15</td>
<td>DOE/SC/Pacific Northwest National Laboratory</td>
<td>Cascade - Atipa Visione IF442 Blade Server, Xeon E5-2670 8C 2.600GHz, Infiniband FDR, Intel Xeon Phi 5110P</td>
<td>194616</td>
<td>2539.1</td>
<td>3388.0</td>
<td>1384</td>
</tr>
<tr>
<td>16</td>
<td>Total Exploration Production</td>
<td>France - Pangea - SGI ICE X, Xeon E5-2670 8C 2.600GHz, Infiniband FDR</td>
<td>110400</td>
<td>2098.1</td>
<td>2296.3</td>
<td>2118</td>
</tr>
<tr>
<td>17</td>
<td>CINECA</td>
<td>Italy - Fermi - BlueGene/Q, Power BQC 16C 1.600GHz, Custom IBM</td>
<td>163840</td>
<td>1788.9</td>
<td>2097.2</td>
<td>822</td>
</tr>
<tr>
<td>18</td>
<td>DOE/SC/LBNL/NERSC</td>
<td>United States - Edison - Cray XC30, Intel Xeon E5-2695v2 12C 2.4GHz, Aries interconnect</td>
<td>133824</td>
<td>1654.7</td>
<td>2569.4</td>
<td>822</td>
</tr>
</tbody>
</table>

**top500.org: 06/2014 list of large scale systems.**
Part 2: NVIDIA GPUs & CUDA

NVIDIA’s Compute Unified Device Architecture
GPU programming model
CUDA

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

- We refer instead to a thread array and SIMD groups.
Each discrete GPU is a nearly self-contained daughter card (aka sidecar or accelerator).
Data has to be off loaded from the HOST to the DEVICE.
Latest GPUs use the PCI Express 3.0 interface standard.

The GPU “device” communicates with the “host” through the PCI slot.

GTX 590 GPU

PCI Express 2.0 interface
[ all data exchange between CPU & GPU crosses this bus ]
Discrete GPU cards have one or two graphics processing units. Latest cards from NVIDIA have Kepler GPU chip(s).
The amount of “device memory” and “device bandwidth” depends on the specific model.
The GPU power requirement is typically limited by the PCI Express standard.

Power inputs:
TDP* 365W

Wiki description of TDP: “The thermal design power (TDP), sometimes called thermal design point, refers to the maximum amount of power the cooling system in a computer is required to dissipate.”

A high end GPU might draw 250W. See: NVIDIA GPU wiki for detailed specs.
Two GPUs (DEVICEs) plugged into HOST motherboard.
GPU: excess ALUs

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)

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 😊
GPU: NVIDIA 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)

Image credits:
http://www.anandtech.com/show/6446/nvidia-launches-tesla-k20-k20x-gk110-arrives-at-last/3
http://www.tomshardware.com/reviews/geforce-gtx-titan-gk110-review,3438.html
The FPU cluster sizes have ballooned: 16 - 24 - 32 - 192 but the shared memory and register file have not grown accordingly.
Maxwell notes: Partitioned register files, multiple instruction buffers, partitioned core, same shared memory, multiple texture/L1 caches.

GPU: Kepler to Maxwell

The FPU clusters ("core") in 2 latest NVIDIA processor architectures

http://www.ubergizmo.com/2014/02/nvidia-maxwell-gpu-for-geforce-cards/
Major bump in performance Maxwell - Pascal transition is contingent on stacked memory. 
[ http://www.anandtech.com/show/7900/nvidia-updates-gpu-roadmap-unveils-pascal-architecture-for-2016 ]
There is a major bottleneck between the GPU and CPU. Data localization is critical for high-performance.
CUDA is used to program NVIDIA GPUs.
CUDA includes a HOST API and a DEVICE kernel programming language.

CUDA was released by NVIDIA in 2007.
CUDA: dataflow

In CUDA the programmer explicitly moves data between HOST and DEVICE

Key observation: the DEVICE and HOST are asynchronous.

Detail: cudaMemcpy will block until the queued DEVICE actions complete.
CUDA: dataflow

In CUDA the programmer explicitly moves data between HOST and DEVICE

Key observation: the DEVICE and HOST are asynchronous.

Detail: cudaMemcpy will block until the queued DEVICE actions complete.
CUDA: dataflow

In CUDA the programmer explicitly moves data between HOST and DEVICE.

Key observation: the DEVICE and HOST are asynchronous.

1. cudaMalloc: allocate memory for a DEVICE array
2. cudaMemcpy: copy data from HOST to DEVICE array

Detail: cudaMemcpy will block until the queued DEVICE actions complete.
CUDA: dataflow

In CUDA 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

Key observation: the DEVICE and HOST are asynchronous.

Detail: `cudaMemcpy` will block until the queued DEVICE actions complete.
CUDA: dataflow

In CUDA the programmer explicitly moves data between HOST and DEVICE.

1. cudaMemcpy: copy data from DEVICE to HOST 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.

Detail: cudaMemcpy will block until the queued DEVICE actions complete.
CUDA: dataflow

In CUDA 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.

Detail: `cudaMemcpy` will block until the queued DEVICE actions complete.
CUDA: HOST code

Overview of C-like 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 = 17;
    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: `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:

```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
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:

   ```c
   float *d_a; // Allocate DEVICE array (pointers used as array handles)
   cudaMemcpy((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:

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

4. Copy results from DEVICE to HOST:

   ```c
   ```

Key API calls: `cudaMalloc`, `cudaMemcpy`
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:

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

4. Copy results from DEVICE to HOST:

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

**Key API calls:** `cudaMalloc, cudaMemcpy`
void serialSimpleKernel(int N, float *d_a) {
    for(n=0;n<N;++n){ // loop over N entries
        d_a[n] = n;
    }
}
CUDA: motivating serial function

Consider the case with N=20 - then break the for loop into independent tiles:

```c
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: \( N \leq gridDim.x*blockDim.x \)
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;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: $N \leq \text{gridDim.x} \times \text{blockDim.x}$
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: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: $N \leq \text{gridDim.x*blockDim.x}$
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>blockDim.x</td>
<td>blockDim.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></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>512</td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</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>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>

```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.*
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 kernel code

```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 kernel code

```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: code samples @ github

github repo: https://github.com/tcew/ATPESC14
See: examples/cuda/simple
# compile on node with the NVIDIA CUDA compiler (nvcc) installed
nvcc -o simpleKernel simpleKernel.cu

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

Source code: [https://github.com/tcew/ATPESC14/examples/cuda/simple](https://github.com/tcew/ATPESC14/examples/cuda/simple)
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:

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

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

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.
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):

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

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

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):

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

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

Jacobi iteration for discrete Poisson problem:

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

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

Rearranging we are left with a simple five point recurrence:

Jacobi iteration for discrete Poisson problem:

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

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

Iterate:

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

while:

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

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

Iterate:

\[ u_{ji}^{k+1} = \frac{1}{4} \left( -\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{j(i+1)}^k + u_{j(i-1)}^k \right) \text{ for } i, j = 1, \ldots, N \]
For the iterate step we note: each node can update independently for maximum parallelism.

$$u_{ji}^{k+1} = \frac{1}{4} \left( -\delta^2 f_{ji} + u_{(j+1)i}^k + u_{(j-1)i}^k + u_{j(i+1)}^k + u_{j(i-1)}^k \right) \text{ for } i, j = 1, \ldots, N$$
**CUDA: serial Jacobi iteration**

The explicit serial loop structure for the Jacobi iteration shows no loop carry dependence:

Iterate:

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

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.

At the start we set: \( rhs = -\delta \cdot \delta \cdot f \)
CUDA: parallel Jacobi iteration

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

Iterate:

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

Note: we use an N\times N array of threads and leave the edge nodes unchanged.

CUDA kernel:

```c
__global__ void jacobi(const int N,
               const datafloat *rhs,
               const datafloat *u,
               datafloat *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]);
    }
}
```

At the start we set: \( rhs = -\delta \cdot \delta \cdot f \)
CUDA: parallel reduction

Second step: reduce solution array to a single scalar.

Check if:

\[ \varepsilon \coloneq \sqrt{\sum_{i=1}^{i=N} \sum_{j=1}^{j=N} \left( u_{ji}^{k+1} - u_{ji}^k \right)^2} > tol \]

Simplify to reduction of a linear vector:

\[ \varepsilon \coloneq \sum_{i=0}^{i=M-1} v_i \]

Ah - this looks like a very serial sum.
To make this more parallel we need to split it into CUDA thread-blocks:

**Reduction:**

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

**Block reduction (B blocks):**

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

\[ B := \frac{M}{T} \]
CUDA: parallel reduction

Standard tree reduction at the thread-block level!!

Thread-block tree reduction in pseudo-code:

```
t = thread index in thread box;
alive = number of threads in thread block;
s_sumu[t] = u[global thread index];

while(alive>1){
    synchronize threads in thread-block;
    alive /=2;
    if(t < alive)
        s_sumu[t] += s_sumu[t+alive];
}
if(t==0) blocksumu[block index] = s_u[0];
```

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

Target: \[ \sum_{i=0}^{i=T-1} v_i \]
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;
        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 thread-block. All threads in the thread-block have to enter the __syncthreads() before any of them can return.
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;
    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 thread-block. All threads in the thread-block have to enter the __syncthreads() before any of them can return.

CUDA: parallel reduction

Standard tree reduction at the thread-block level!!
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;
    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 thread-block. All threads in the thread-block have to enter the __syncthreads() before any of them can return.
Thread-block tree reduction in pseudo-code:

\[ t = \text{thread index in thread box}; \]
\[ \text{alive} = \text{number of threads in thread block}; \]
\[ s\text{\_sum}_u[t] = u[\text{global thread index}]; \]

\[ \text{while (alive}>1) \{
\]
\[ \text{synchronize threads in thread-block;} \]
\[ \text{alive} /= 2; \]
\[ \text{if (t} < \text{alive)} s\text{\_sum}_u[t] += s\text{\_sum}_u[t+alive]; \]
\[ \}
\[ \text{if(t==0) blocksum[block index] = s\text{\_sum}_u[0];} \]

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;
    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 thread-block. All threads in the thread-block have to enter the __syncthreads() before any of them can return.

CUDA: parallel reduction

Standard tree reduction at the thread-block level!!
CUDA: elliptic solver sample code

See the ATPESC14 github for a full implementation [with some optional goodies]

Compile each example with: make
Run with a 102x102 grid and tolerance 1e-4: ./main 100 1e-4

Let me know if you have any problems with this.
CUDA: elliptic solver sample code

See the ATPESC14 github for a full implementation [ with some optional goodies ]

Compile each example with: make
Run with a 102x102 grid and tolerance 1e-4: ./main 100 1e-4

Let me know if you have any problems with this.
CUDA: elliptic solver sample code

See the ATPESC14 github for a full implementation [with some optional goodies]

Compile each example with: make
Run with a 102x102 grid and tolerance 1e-4: ./main 100 1e-4

Let me know if you have any problems with this.
1.1: Study event timing used to estimate compute time:
   - see NVIDIA tutorial on event timing: link

1.2: Upgrade to higher order (wider) finite difference stencil:
   - see wiki on stencil weights: link

1.3: Change the kernel & host code to use the conjugate gradient method:
   - see wiki on conjugate gradient: link

1.4: Use shared memory to reduce the high-latency device memory traffic.

1.5: Use MPI+CUDA to partition the nodes across multiple GPUs.

1.6: Use a red-black numbering of the nodes to implement Gauss-Seidel iteration:
   - tutorial by Jonathan Cohen (NVIDIA): link

1.7: Experiment with CUDA based OpenCurrent structured grid PDE framework:
   - google code page: link

1.8: Study Mark Harris’ CUDA optimized reduction:
   - lecture notes: link

None of these tasks are particularly difficult, but they are ordered by difficulty.
Part 3: Interlude on CUDA performance

Dark Arts Indeed
“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
In much the same vain…

“A many-core processor is a device for turning a compute-bound problem into a memory-bound problem.”

Kathy Yelick

The latest GPUs have O(2800) floating point units but only O(300) GB/s memory bandwidth off chip…
In much the same vain...

“Arithmetic is cheap, bandwidth is money, latency is physics.”

Mark Hoemmen*

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

*Student of Jim Demmel: thesis web link
## 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*
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>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></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></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</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>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>

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 GPU Occupancy Calculator

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>Select Compute Capability</td>
<td>3.5</td>
</tr>
<tr>
<td>2)</td>
<td>Enter resource usage</td>
<td>Threads Per Block: 234, Registers Per Thread: 32, Shared Memory Per Block: 4096</td>
</tr>
<tr>
<td>3)</td>
<td>GPU Occupancy Data</td>
<td></td>
</tr>
</tbody>
</table>

**Physical Limits for GPU Compute Capability:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per Warp</td>
<td>32</td>
</tr>
<tr>
<td>Warps per Multiprocessor</td>
<td>64</td>
</tr>
<tr>
<td>Threads per Multiprocessor</td>
<td>2096</td>
</tr>
<tr>
<td>Thread Blocks per Multiprocessor</td>
<td>16</td>
</tr>
<tr>
<td>Total # of 32-bit registers per Multiprocessor</td>
<td>665536</td>
</tr>
<tr>
<td>Register allocation unit size</td>
<td>256</td>
</tr>
<tr>
<td>Warp allocation granularity</td>
<td>warp</td>
</tr>
<tr>
<td>Registers per Thread</td>
<td>256</td>
</tr>
<tr>
<td>Shared Memory per Multiprocessor (bytes)</td>
<td>4096</td>
</tr>
<tr>
<td>Warp allocation granularity</td>
<td>4</td>
</tr>
<tr>
<td>Maximum Thread Block Size</td>
<td>1024</td>
</tr>
</tbody>
</table>

**Allocated Resources**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Per Block</th>
<th>Limit Per SM</th>
<th>Allocatable Blocks Per SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warps</td>
<td>(Threads Per Block / Threads Per Warp)</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>Registers</td>
<td>(Warp limit per SM due to parallelizing count)</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>Shared Memory (Bytes)</td>
<td>4096</td>
<td>4096</td>
<td></td>
</tr>
</tbody>
</table>

**Limited by Max Warps or Max Blocks per Multiprocessor**

<table>
<thead>
<tr>
<th>Block</th>
<th>Shares/Warp</th>
<th>Shares/Warps</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM</td>
<td>8</td>
<td>64</td>
</tr>
</tbody>
</table>

**Impact of Varying Register Count Per Thread**

**Impact of Varying Block Size**

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>0</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank 0</td>
<td>0</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>Bank 1</td>
<td>1</td>
<td>33</td>
<td>65</td>
<td>...</td>
</tr>
<tr>
<td>Bank 2</td>
<td>2</td>
<td>34</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>Bank 3</td>
<td>3</td>
<td>35</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>Bank 4</td>
<td>4</td>
<td>36</td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>Bank 5</td>
<td>5</td>
<td>37</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>Bank 30</td>
<td>30</td>
<td>62</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>Bank 31</td>
<td>31</td>
<td>63</td>
<td>95</td>
<td></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.

### Shared Memory: memory space organization

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 31</td>
<td>Thread 30</td>
<td>Thread 29</td>
<td>Thread 28</td>
<td>Thread 27</td>
<td>Thread 26</td>
<td>Thread 25</td>
<td>Thread 24</td>
<td>Thread 23</td>
<td>Thread 22</td>
<td>Thread 21</td>
<td>Thread 20</td>
<td>Thread 19</td>
<td>Thread 18</td>
<td>Thread 17</td>
<td>Thread 16</td>
<td>Thread 15</td>
<td>Thread 14</td>
<td>Thread 13</td>
<td>Thread 12</td>
<td>Thread 11</td>
<td>Thread 10</td>
<td>Thread 9</td>
<td>Thread 8</td>
<td>Thread 7</td>
<td>Thread 6</td>
<td></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.

**Shared Memory: memory space organization**

- Bank 31
- Bank 30
- Bank 5
- Bank 4
- Bank 3
- Bank 2
- Bank 1
- Bank 0

**Thread 31**
- Thread 30
- Thread 5
- Thread 4
- Thread 3
- Thread 2
- Thread 1
- Thread 0

*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.
1. GPU has a “coalescer” that collects SIMD lane 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
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.
CPU Optimization Techniques

**Multithreading**

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

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

- SIMD Lanes
GPU Optimization Techniques

GPU Architecture

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

Kernel code describes the work-item operations

David S Medina
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.
Apple drove the creation of the OpenCL standard for portable cross-vendor many-core programming.
OpenCL: standards body

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)

The Khronos Group administers the OpenCL standard.
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
- GPUs: Increasingly general purpose data-parallel computing

Emerging Intersection

Heterogeneous Computing

- Multi-processor programming – e.g. OpenMP
- Graphics APIs and Shading Languages

OpenCL is a programming framework for heterogeneous compute resources

Emphasis on heterogeneous computing.

Credit: Khronos Group
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.

Only a few hardy souls tried GPU computing before CUDA was released.
OpenCL: terminology?

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 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>get_local_id(0)</td>
</tr>
<tr>
<td>threadIdx.y</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>get_global_id(0)</td>
</tr>
<tr>
<td>blockIdx.y*blockDim.y + threadIdx.y</td>
<td>get_global_id(1)</td>
</tr>
</tbody>
</table>
# 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><code>gridDim.x</code></td>
<td><code>get_num_groups(0)</code></td>
</tr>
<tr>
<td><code>blockIdx.x</code></td>
<td><code>get_group_id(0)</code></td>
</tr>
<tr>
<td><code>blockDim.x</code></td>
<td><code>get_local_size(0)</code></td>
</tr>
<tr>
<td><code>gridDim.x*blockDim.</code></td>
<td><code>get_global_size(0)</code></td>
</tr>
</tbody>
</table>

The rapid development of OpenCL helps explain the similarities.
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>

The rapid development of OpenCL helps explain the similarities.
The rapid development of OpenCL helps explain the similarities.

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

CUDA

OpenCL

Image system not shown

AMD OpenCL slides
OpenCL: setting up a DEVICE

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`
OpenCL: HOST code API headers

The include files ...

CUDA

```
#include <cuda.h>
```

OpenCL

```
#ifdef __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.

OpenCL: setting up a platform

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];
```
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);
```
OpenCL: setting up a common queue

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

```c
...
```
OpenCL: compiling a DEVICE kernel

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:
   
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`

```c
const char *source =
    "__kernel void foo(int N, __global float *x){
        int id = get_global_id(0);
        if(id<N)
            x[id] = id;
    }";
```
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);
```
OpenCL: kernel good to go?

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);
```

Note: CUDA uses block sizes + number of blocks.
OpenCL uses block sizes and global number of threads.
Some minor differences in syntax & identifiers
Queue the live demo.
OpenCL: compiling & running example

There may be variations depending on how your system is set up.

```
# compile on node with the g++ compiler and an OpenCL framework
# Linux:
g++ -o simple simple.cpp -lOpenCL

# OS X
g++ -o simple simple.cpp -framework OpenCL

# run on node that has OpenCL installed
./simple
```

Make sure you can complete this exercise!

Source code: https://github.com/tcew/ATPESC14/examples/cuda/simple
Recalling the Poisson example: side by side comparison of serial v. CUDA v. OpenCL kernel

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 \( i, j = 1, ..., N \)

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]);
        }
    }
}
```

CUDA kernel:
```c
__global__ void jacobi(const int N,
                      const datafloat *rhs,
                      const datafloat *u,
                      datafloat *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 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]);
    }
}
```

OpenCL kernel:
```c
__kernel void jacobi(const int N,
                    __global const datafloat *rhs,
                    __global const datafloat *u,
                    __global datafloat *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!!

The example code in ATPESC14/examples/opencl/jacobi/reduce.cl is more verbose
OpenCL: partial 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;

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

s_blocksum[threadIdx.x] = 0;

if(id < entries){
    const datafloat 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];
}
}
```

The example code in ATPESC14/examples/opencl/jacobi/reduce.cl is more verbose
OpenCL: partial 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;
int alive = blockDim.x;
int t = threadIdx.x;
s_blocksum[threadIdx.x] = 0;
if(id < entries){
    const datafloat 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, 
__global const datafloat *u, 
__global const datafloat *newu, 
__global datafloat *blocksum){
__local datafloat s_blocksum[BDIM];
const int id = get_global_id(0);
int alive = get_local_size(0);
int t = get_local_id(0);
s_blocksum[t] = 0;
// load global data into local memory if in range
if(id < entries){
    const datafloat diff = u[id] - newu[id]; 
    s_blocksum[t] = 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[get_group_id(0)] = s_blocksum[0];
}
```

The example code in ATPESC14/examples/opencl/jacobi/reduce.cl is more verbose
Part 5: OCCA

Extensible API for portable many-core computing
https://github.com/tcew/OCCA2
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 + ?

Does it even matter what “X” is?
May’s Law

Have you noticed that computers get better specs but do not seem to get substantially faster:

“First, a disappointment. It is widely accepted that hardware efficiency doubles every 18 months, following Moore’s law. Let me now introduce you to May’s law:

software efficiency halves every 18 months, compensating Moore’s law!

It’s not clear what has caused this, but the tendency to add features, programming using copy-paste techniques, and programming by ‘debugging the null-program’ - starting with a debugger and an empty screen and debugging interactively until the desired program emerges - have probably all contributed.”

David May
Popular Multi-threading APIs

OpenMP:
- Shared memory model.
-Pragma based code parallelization.
- Standard maintained by the OpenMP Architecture Review Board.
- Targets: currently CPUs + Intel Xeon Phi, with plans for GPU support.

CUDA:
- Manages host-device (accelerator) communications and on-device calculations.
- Kernel based calculations.
- Proprietary language developed by NVIDIA that targets NVIDIA accelerators [and x86].

OpenCL:
- Similar in structure to CUDA.
- Standard maintained by the Khronos Group.
- Targets: almost all CPUs, GPUs, FPGAs, and Accelerators.

OpenACC:
-Pragma based accelerator coding.
- Developed by Cray, CAPS, NVIDIA, and PGI (now an NVIDIA company).
- Work in progress, may merge with OpenMP?

OpenACC will be discussed this afternoon.
Can we insulate against uncertainty and fragmentation?
Can we avoid maintaining multiple code bases?
Many-core Smorgasbord

There is a zoo of competing architectures for many-core devices...

Can we insulate against uncertainty and fragmentation?
Can we avoid maintaining multiple code bases?
Can we insulate against uncertainty and fragmentation?
Can we avoid maintaining multiple code bases?

Many-core Smorgasbord

There is a zoo of competing architectures for many-core devices...

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

OpenMP

OpenACC

... and a range of thread programming models (with vendor favorite pairs).
Many-core Smorgasbord

There is a zoo of competing architectures for many-core devices…

Can we insulate against uncertainty and fragmentation?
Can we avoid maintaining multiple code bases?
Many-core Smorgasbord

There is a zoo of competing architectures for many-core devices...

... and a range of thread programming models (with vendor favorite pairs).

Can we insulate against uncertainty and fragmentation?
Can we avoid maintaining multiple code bases?
Uncertainty:

- Code life cycle measured in decades.
- Architecture & API life cycles measured in Moore doubling periods.
- Example: if you coded for the IBM Cell processor API...

Fragmentation:

- Pthreads, CUDA, OpenCL, OpenMP, OpenACC, Intel TBB... are not source code compatible.
- Not all APIs are installed on all systems.

Performance*:

- Naively porting between OpenMP, CUDA, OpenCL may yield low performance.
- Manufacturers devote resources to enhancing specific APIs.
- High level APIs may induce excess data movement.

Parallelism*:

- The programmer has to expose parallelism in memory access and operations.

We need a simple and uniform approach so codes can run with CUDA, OpenCL, Pthreads, or OpenMP backends automatically.
### Subset of Approaches to Portability

Numerous approaches to portability

<table>
<thead>
<tr>
<th>API</th>
<th>Type</th>
<th>Front-ends</th>
<th>Kernel Language</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>OCCA 1.0</td>
<td>API</td>
<td>C,C++,F90</td>
<td>Custom Unified Kernel Language</td>
<td>CUDA, OpenCL, &amp; OpenMP</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>Intermediate</td>
<td>C</td>
<td>OpenMP,Cilk,MPI, OpenCL</td>
<td>OpenCL,MPI, Insieme IR runtime</td>
</tr>
<tr>
<td></td>
<td>Representation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OmpSs</td>
<td>Directives +</td>
<td>C,C++</td>
<td>Hybrid OpenMP, OpenCL, CUDA</td>
<td>OpenMP, OpenCL, CUDA</td>
</tr>
<tr>
<td></td>
<td>kernels</td>
<td></td>
<td></td>
<td></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.**
OCCA 2.0: unified threading model

Portability & extensibility: device independent kernel language and native host APIs.

https://github.com/tcew/OCCA2

See talk by David Medina (MS6: Tuesday 10am)
OCCA 2.0: API stack

OCCA 2.0 communicates with several different run-times.

Continuing to add new front-ends and back-ends. Access to pre-alpha available on request.
OCCA 2.0: C++ host API

**occa::memory class:** abstracts the memory handles found in each language

- occa::memory
- occa::device::malloc()

**occa::kernel class:** encapsulates function handles and uses run-time compilation

- void* malloc()
- cl_mem clCreateBuffer()
- CUdeviceptr cuMemAlloc()
- g++/icpc, i.e. void (*fptr)()
- clBuildProgram, i.e. cl_kernel
- nvcc, i.e. CUmodule

OCCA uses the CUDA Driver API for run-time compilation of OCCA kernels as CUDA kernels.
OCCA: exposed kernel loop structure

OCCA Kernel API

- Relies on macros for masking the different supported languages.
- Uses the GPU programming model of work-groups and work-items.

Explicit Work-groups and Work-Items

- Work-group and work-item are explicitly expressed as OCCA for-loops.
OCCA: shared memory and registers

Shared memory

- Shared memory crucial for GPU-modes (requires barrier for syncing work-items).
- Manually caching in CPU-mode.

```c
// Shared memory defined here
occaInnerFor2{
    occaInnerFor1{
        occaInnerFor0{
            // ...
        }
    }
}
occaBarrier(occaLocalMemFence);

occaInnerFor2{
    occaInnerFor1{
        occaInnerFor0{
            // ...
        }
    }
}
```

Register memory

- Fastest memory on GPUs.
- Another issue with for-loop scopes … require thread-local-storage-type of data.
OCCA: private variables

occaPrivate Classes

• Template thread-safe private classes in OpenMP-mode.

```c
occaKernel void kernelExample(occaKernelInfoArg, ...){
    occaOuterFor0{
        occaPrivate(int, reg);       // Register : int reg;
        occaPrivateArray(int, regArray, 2); // Register Array: int regArray[2];
        // . . .
        occaInnerFor0{
            reg = occaGlobalId0;       // reg would normally be overwritten by the loop
            regArray[0] = 0;           // regArray would also normally be overwritten
            regArray[1] = 1;
            // . . .
        }
    }
    occaBarrier(occaLocalMemFence);
    occaInnerFor0{
        int i = reg;                // Allocating registers normally for only one scope
        int d = regArray[0];
        // . . .
    }
}
```
OCCA Programming Model

Goal: one implementation for each kernel.

__global__ void kernelFunction(arguments){

// embarrassingly parallel outer loops (implicit)

// shared memory for collaboration between threads
__shared__ float s_a[100];

// embarrassingly parallel inner loops (implicit)
kernel code here;

// synchronization writes to shared memory
__syncthreads();

// embarrassingly parallel inner loops
kernel code here;
}

__kernel void kernelFunction(arguments){

// embarrassingly parallel outer loops implicit

// shared memory for collaboration between threads
__local float s_a[100];

// embarrassingly parallel inner loops (implicit)
kernel code here;

// synchronization writes to shared memory
barrier(CLK_LOCAL_MEM_FENCE);

// embarrassingly parallel inner loops (implicit)
kernel code here;
}

Confusing CUDA kernel
[ hidden parallel for loops ]

Confusing OpenCL kernel
[ hidden parallel for loops ]

Thread array <> parallel loop map made explicit.
OpenCL and CUDA kernels use an implicit parallel loop structure.

OCCA exposes these loops.

OCCA kernels compile at runtime to serial, OpenCL, CUDA, OpenMP, or Pthreads.

Minimum requirement:
1 outer loop & 1 inner loop.

Loop bounds are specified

```c
occaKernel void kernelFunction(arguments){
    // embarrassingly parallel outer loops (explicit)
    occaOuterFor1 {
        occaOuterFor0 {
            // shared memory for collaboration between threads
            occaShared float s_a[100];

            // embarrassingly parallel inner loops (explicit)
            occaInnerFor2{
                occaInnerFor1{
                    occaInnerFor0{
                        kernel code here;
                    }
                }
            }

            // synchronization writes to shared memory
            occaBarrier(occaLocalMemFence);

            // embarrassingly parallel inner loops (explicit)
            occaInnerFor2{
                occaInnerFor1{
                    occaInnerFor0{
                        kernel code here;
                    }
                }
            }
        }
    }

    // synchronization writes to shared memory
    occaBarrier(occaLocalMemFence);

    // embarrassingly parallel inner loops (explicit)
    occaInnerFor2{
        occaInnerFor1{
            occaInnerFor0{
                kernel code here;
            }
        }
    }
}
```
#include <iostream>
#include "occa.hpp"

int main(int argc, char **argv){
    float *a  = new float[5];
    float *b  = new float[5];
    float *ab = new float[5];

    for(int i = 0; i < 5; ++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(5*sizeof(float));
    o_b  = device.malloc(5*sizeof(float));
    o_ab = device.malloc(5*sizeof(float));

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

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

    int dims = 1;
    int itemsPerGroup(2);
    int groups((5 + itemsPerGroup - 1)/itemsPerGroup);

    addVectors.setWorkingDims(dims, itemsPerGroup, groups);

    addVectors(5, o_a, o_b, o_ab);
    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
        std::cout << i << "i << "ab[i] << 'n';
OCCA 2.0: example adding two vectors

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

int main(int argc, char **argv){
    float *a = new float[5];
    float *b = new float[5];
    float *ab = new float[5];

    for(int i = 0; i < 5; ++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(5*sizeof(float));
    o_b = device.malloc(5*sizeof(float));
    o_ab = device.malloc(5*sizeof(float));

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

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

    int dims = 1;
    int itemsPerGroup(2);
    int groups((5 + itemsPerGroup - 1)/itemsPerGroup);

    addVectors.setWorkingDims(dims, itemsPerGroup, groups);

    addVectors(5, o_a, o_b, o_ab);
    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
        std::cout << i << " : " << ab[i] << '
';
}
```
OCCA 2.0: example adding two vectors

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

int main(int argc, char **argv) {
    float *a = new float[5];
    float *b = new float[5];
    float *ab = new float[5];

    for (int i = 0; i < 5; ++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(5*sizeof(float));
    o_b  = device.malloc(5*sizeof(float));
    o_ab = device.malloc(5*sizeof(float));

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

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

    int dims = 1;
    int itemsPerGroup(2);
    int groups((5 + itemsPerGroup - 1)/itemsPerGroup);

    addVectors.setWorkingDims(dims, itemsPerGroup, groups);

    addVectors(5, 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[5];
    float *b = new float[5];
    float *ab = new float[5];

    for(int i = 0; i < 5; ++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(5*sizeof(float));
    o_b = device.malloc(5*sizeof(float));
    o_ab = device.malloc(5*sizeof(float));

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

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

    int dims = 1;
    int itemsPerGroup(2);
    int groups((5 + itemsPerGroup - 1)/itemsPerGroup);

    addVectors.setWorkingDims(dims, itemsPerGroup, groups);

    addVectors(5, 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[5];
    float *b = new float[5];
    float *ab = new float[5];

    for (int i = 0; i < 5; ++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(5*sizeof(float));
    o_b = device.malloc(5*sizeof(float));
    o_ab = device.malloc(5*sizeof(float));

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

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

    int dims = 1;
    int itemsPerGroup(2);
    int groups((5 + itemsPerGroup - 1)/itemsPerGroup);

    addVectors.setWorkingDims(dims, itemsPerGroup, groups);

    addVectors(5, o_a, o_b, o_ab);
    o_ab.copyTo(ab);

    for (int i = 0; i < 5; ++i)
        std::cout << i << " : " << ab[i] << '
';
}
```
OCCA 2.0: example adding two vectors

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

int main(int argc, char **argv){
    float *a = new float[5];
    float *b = new float[5];
    float *ab = new float[5];

    for(int i = 0; i < 5; ++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(5*sizeof(float));
    o_b  = device.malloc(5*sizeof(float));
    o_ab = device.malloc(5*sizeof(float));

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

    addVectors = device.buildKernelFromSource(“addVectors.occa”, "addVectors");

    int dims = 1;
    int itemsPerGroup(2);
    int groups((5 + itemsPerGroup - 1)/itemsPerGroup);

    addVectors.setWorkingDims(dims, itemsPerGroup, groups);

    addVectors(5, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

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

int main(int argc, char **argv)
{
    float *a = new float[5];
    float *b = new float[5];
    float *ab = new float[5];

    for(int i = 0; i < 5; ++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(5*sizeof(float));
    o_b = device.malloc(5*sizeof(float));
    o_ab = device.malloc(5*sizeof(float));

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

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

    int dims = 1;
    int itemsPerGroup(2);
    int groups((5 + itemsPerGroup - 1)/itemsPerGroup);

    addVectors.setWorkingDims(dims, itemsPerGroup, groups);

    addVectors(5, 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[5];
    float *b = new float[5];
    float *ab = new float[5];

    for(int i = 0; i < 5; ++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(5*sizeof(float));
    o_b = device.malloc(5*sizeof(float));
    o_ab = device.malloc(5*sizeof(float));

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

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

    int dims = 1;
    int itemsPerGroup(2);
    int groups((5 + itemsPerGroup - 1)/itemsPerGroup);

    addVectors.setWorkingDims(dims, itemsPerGroup, groups);

    addVectors(5, 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[5];
    float *b = new float[5];
    float *ab = new float[5];

    for(int i = 0; i < 5; ++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(5*sizeof(float));
    o_b  = device.malloc(5*sizeof(float));
    o_ab = device.malloc(5*sizeof(float));

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

    int dims = 1;
    int itemsPerGroup(2);
    int groups((5 + itemsPerGroup - 1)/itemsPerGroup);
    
    addVectors.setWorkingDims(dims, itemsPerGroup, groups);

    addVectors(5, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

    for(int i = 0; i < 5; ++i)
        std::cout << i <<": " << ab[i] << '
';
OCCA 2.0: example adding two vectors

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

int main(int argc, char **argv){
    float *a = new float[5];
    float *b = new float[5];
    float *ab = new float[5];

    for(int i = 0; i < 5; ++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(5*sizeof(float));
    o_b = device.malloc(5*sizeof(float));
    o_ab = device.malloc(5*sizeof(float));

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

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

    int dims = 1;
    int itemsPerGroup(2);
    int groups((5 + itemsPerGroup - 1)/itemsPerGroup);

    addVectors.setWorkingDims(dims, itemsPerGroup, groups);

    addVectors(5, 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[5];
    float *b = new float[5];
    float *ab = new float[5];

    for(int i = 0; i < 5; ++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(5*sizeof(float));
    o_b  = device.malloc(5*sizeof(float));
    o_ab = device.malloc(5*sizeof(float));

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

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

    int dims = 1;
    int itemsPerGroup(2);
    int groups((5 + itemsPerGroup - 1)/itemsPerGroup);

    addVectors.setWorkingDims(dims, itemsPerGroup, groups);
    addVectors(5, 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[5];
    float *b = new float[5];
    float *ab = new float[5];

    for(int i = 0; i < 5; ++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(5*sizeof(float));
    o_b = device.malloc(5*sizeof(float));
    o_ab = device.malloc(5*sizeof(float));

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

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

    int dims = 1;
    int itemsPerGroup(2);
    int groups((5 + itemsPerGroup - 1)/itemsPerGroup);

    addVectors.setWorkingDims(dims, itemsPerGroup, groups);

    addVectors(5, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

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

OCCA 2.0 kernel language syntax & macros: http://www.github.com/tcew/OCCA2
Example HOST & DEVICE 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

```c
#include "stdlib.h"
#include "stdio.h"
#include "occa_c.h"

int main(int argc, char **argv)
{
    int i;
    float *a = (float*) malloc(5*sizeof(float));
    float *b = (float*) malloc(5*sizeof(float));
    float *ab = (float*) malloc(5*sizeof(float));

    for(i = 0; i < 5; ++i){
        a[i] = i;
        b[i] = 1 - i;
        ab[i] = 0;
    }

    occaDevice device;
    occaKernel addVectors;
    occaMemory o_a, o_b, o_ab;

    device = occaGetDevice("OpenCL", 0, 0);

    o_a = occaDeviceMalloc(device, 5*sizeof(float), NULL);
    o_b = occaDeviceMalloc(device, 5*sizeof(float), NULL);
    o_ab = occaDeviceMalloc(device, 5*sizeof(float), NULL);

    addVectors = occaBuildKernelFromSource(device,
                                           "addVectors.occa", "addVectors",
                                           occaNoKernelInfo);

    int dims = 1;
    occaDim itemsPerGroup, groups;

    itemsPerGroup.x = 2;
    groups.x = (5 + itemsPerGroup.x - 1)/itemsPerGroup.x;

    occaKernelSetWorkingDims(addVectors,
                              dims, itemsPerGroup, groups);

    occaCopyPtrToMem(o_a, a, 5*sizeof(float), 0);
    occaCopyPtrToMem(o_b, b, occaAutoSize, occaNoOffset);

    occaKernelRun(addVectors,
                  occaInt(5), o_a, o_b, o_ab);

    occaCopyMemToPtr(ab, o_ab, occaAutoSize, occaNoOffset);

    for(i = 0; i < 5; ++i)
        printf("%d = %f \n", i, ab[i]);
}
```
Native Host Codes

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

```c
#include "stdlib.h"
#include "stdio.h"
#include "stdlib.h"
#include "stdlib.h"
#include "stdlib.h"

int main(int argc, char **argv) {
    int i;
    float a[2];
    float b[2];
    float ab[2];
    for(i = 0; i < 5; ++i)
        printf("%d = %f\n", i, ab[i]);
}
```

```c
#include "stdlib.h"
#include "stdlib.h"
#include "stdlib.h"
#include "stdlib.h"
#include "stdlib.h"

int main(int argc, char **argv) {
    int i;
    float a[2];
    float b[2];
    float ab[2];
    for(i = 0; i < 5; ++i)
        printf("%d = %f\n", i, ab[i]);
}
```

```c
#include "stdlib.h"
#include "stdlib.h"
#include "stdlib.h"
#include "stdlib.h"
#include "stdlib.h"

int main(int argc, char **argv) {
    int i;
    float a[2];
    float b[2];
    float ab[2];
    for(i = 0; i < 5; ++i)
        printf("%d = %f\n", i, ab[i]);
}
```
```python
from ctypes import *
import occa

entries = 5
a = [i for i in xrange(entries)]
b = [1 - i for i in xrange(entries)]
ab = [0 for i in xrange(entries)]

device = occa.device("OpenCL", 0, 0)
o_a = device.malloc(a, c_float)
o_b = device.malloc(b, c_float)
o_ab = device.malloc(ab, c_float)

addVectors = device.buildKernelFromSource("addVectors.occa",
"addVectors")

dims = 1
itemsPerGroup = 2
groups = (entries + itemsPerGroup - 1)/itemsPerGroup

addVectors.setWorkingDims(dims, itemsPerGroup, groups)
addVectors([c_int(entries),
o_a, o_b, o_ab])
o_ab.copyTo(ab, c_float)
print ab

println(ab)

occaMemcpy(ab, o_ab)

```

All the HOST codes use the same kernel.
Example HOST code: https://github.com/tcew/OCCA2/tree/master/examples/addVectors
```python
# Dynamic range?
entries = 5
a = [i for i in xrange(entries)]
b = [1 - i for i in xrange(entries)]
ab = [0 for i in xrange(entries)]

# Dynamic range?
entries = 5
a = ones(entries, 1)
b = ones(entries, 1)
ab = zeros(entries, 1)

device = occa.device('OpenCL', 0, 0);
o_a = device.malloc(a , 'single');
o_b = device.malloc(b , 'single');
o_ab = device.malloc(ab, 'single');
addVectors = device.buildKernelFromSource('addVectors.occa', ...
               'addVectors');

dims = 1;
itemsPerGroup = 2;
groups = (entries + itemsPerGroup - 1)/itemsPerGroup;
addVectors.setWorkingDims(dims, itemsPerGroup, groups);
addVectors(o_a, o_b, o_ab);
print ab
```

**Note:** The Python code shown is a representation of the same functionality as the C code in the image. It is not a direct translation but serves to illustrate how the operations can be implemented in Python using the OCCA library. The image also contains C code excerpts which are not translated here due to the focus on the Python example. For a complete understanding, one should refer to the full code listings shared in the image or linked resources.
OCCA: portability?

Optimization strategies for different platforms are converging to some extent.

**CPU Optimizations**

- **Cache**
  - L1 Cache
- **Vectorization**
- **Thread Independence**
  - 0 32 64 96 128 160 192 224 256 288 320
  - SSE

**GPU Optimizations**

- **Coalescing**
  - 0 32 64 96 128 160 192 224 256 288 320
  - 0 1 2 3
- **No Branching**
- **Work-group Independence**
  - Shared

Caching & coalescing are similar in spirit.
OCCA: portability?

Optimization strategies for different platforms are converging to some extent.

CPU Optimizations

- Cache
- Vectorization
- Thread Independence

GPU Optimizations

- Coalescing
- No Branching
- Work-group Independence

Caching & coalescing are similar in spirit.
Explicit vectorization can help on AMD GPUs or Intel/AMD CPUs.

OCCA: portability?

Don’t branch if you can help it.
OCCA: portability?

Don’t branch if you can help it.

CPU Optimizations

Cache

Vectorization

Thread Independence

L1 Cache

Vectorization

Thread Independence

CPU Optimizations

GPU Optimizations

Explicit vectorization can help on AMD GPUs or Intel/AMD CPUs.
An experimental version of OCCA2 is available from github

https://github.com/tcew/OCCA2
1: git clone and build the OCCA library. 2: set up the environment variables for OCCA
3: Let me know ** when ** you have any problems with this.

See the ATPESC14 github for a full implementation [ with some optional goodies ]
OCCA: elliptic solver sample code

See the ATPESC14 github for a full implementation [ with some optional goodies ]

1: git clone and build the OCCA library.
2: set up the environment variables for OCCA.
3: Let me know ** when ** you have any problems with this.
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...)

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]);
        }
    }
}
```

CUDA kernel:

```c
__global__ void jacobi(const int N,
                       const datafloat *rhs,
                       const datafloat *u,
                       datafloat *newu){
    // Get thread indices
    const int i = blockIdx.x*blockDim.x + threadIdx.x;
    const int j = blockIdx.y*blockDim.y + threadIdx.y;
    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,
                     const datafloat *rhs,
                     const datafloat *u,
                     datafloat *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]);
    }
}
```
OCCA: comparing Jacobi kernels

Recalling the Poisson example: comparison of serial v. CUDA v. OpenCL v. OCCA kernels

Serial kernel:

```c
void jacobi(const int N,
            const datafloat *rhs,
            const datafloat *u,
            datafloat *newu){
    for(int i=0;i<N;++i){
        // 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]);
    }
}
```

CUDA kernel:

```c
__global__ void jacobi(const int N,
                       const datafloat *rhs,
                       const datafloat *u,
                       datafloat *newu){
    // Get thread indices
    const int i = blockIdx.x*blockDim.x + threadIdx.x;
    const int j = blockIdx.y*blockDim.y + threadIdx.y;
    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]);
    }
}
```

OpenCL kernel:

```
__kernel void jacobi(const int N,
                     __global const datafloat *rhs,
                     __global const datafloat *u,
                     __global datafloat *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]);
    }
}
```

OCCA kernel:

```
occaKernel void jacobi(occaKernelInfoArg,
                        const occaVariable N,
                        occaPointer const datafloat *rhs,
                        occaPointer const datafloat *u,
                        occaPointer datafloat *newu){
    occaOuterFor1{
        occaOuterFor0{
            occaInnerFor1{
                occaInnerFor0{
                    // Get thread indices
                    const int i = occaGlobalId0;
                    const int j = occaGlobalId1;
                    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]);
                    }
                }
            }
        }
    }
}
```

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...).
OCCA Based Applications

OCCA applications include: shallow water modeling, seismic imaging, fluid flow, algebraic multi-grid solvers, gas dynamics, immiscible multi fluids, radiation transport, electromagnetics, …
gNek: high-order spectral element method

We use a double sweep threaded algorithm for high-order elements:
up to $N=13$ with OpenCL and $N=29$ with CUDA.

Threads use shared memory to perform tensor-product contractions (derivatives, projections, ...)

The PCG kernels are generally memory bound.
OCCA: 3D Spectral Elements

Scatter and matrix-free elliptic operator application.

- Kernel: application of local stiffness matrix: \((\lambda u_h, \phi_h)_\Omega + (k \nabla_h u_h, \nabla_h \phi_h)_\Omega = (f, \phi_h)_\Omega\)
- Two-dimensional thread array.

Work-items are padded to multiple of 8 to take advantage of Intel OpenCL vectorization on CPU.
Performance of gNek Elliptic Solver (DP)

- CUDA: K40
- OpenCL: Hawaii
- OpenCL: Tahiti
- ICPC: Intel I7
- OpenCL: Intel I7

Performance aggregate for all kernels in PCG solver.
**OCCCA: comparison with expert SEM**

Comparison of expert coded SEM solver performance for preconditioned PCG iterative solves: degree 8 elements

<table>
<thead>
<tr>
<th>Code Base</th>
<th>Parallel Model</th>
<th>Processor</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (DP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nek5K</td>
<td>MPI</td>
<td>Intel E5430 (4core)</td>
<td>Est. 6.3</td>
</tr>
<tr>
<td>Nek5k</td>
<td>MPI</td>
<td>Intel SandyBridge</td>
<td>Est. 13</td>
</tr>
<tr>
<td>gNek</td>
<td>OCCA:OpenMP</td>
<td>Intel i7-3930K (6core)</td>
<td>18 icpc</td>
</tr>
<tr>
<td>gNek</td>
<td>OCCA:OpenCL</td>
<td>Intel i7-3930K (6core)</td>
<td>13</td>
</tr>
<tr>
<td>GPU (DP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NekBone</td>
<td>CUDA</td>
<td>NVIDIA K40</td>
<td>NDA</td>
</tr>
<tr>
<td>gNek</td>
<td>OCCA:CUDA</td>
<td>NVIDIA K40</td>
<td>123</td>
</tr>
<tr>
<td>gNek</td>
<td>OCCA:OpenCL</td>
<td>AMD HD 7970</td>
<td>114*</td>
</tr>
<tr>
<td>GPU (SP)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gNek</td>
<td>OCCA:CUDA</td>
<td>NVIDIA Titan</td>
<td>196</td>
</tr>
<tr>
<td>gNek</td>
<td>OCCA:OpenCL</td>
<td>AMD HD 7970</td>
<td>267</td>
</tr>
</tbody>
</table>

*Performance aggregate for all kernels in PCG solver.*
Strange Things Afoot at the Circle K

With OCCA we can run in OpenCL or CUDA mode on NVIDIA GPUs.

NVIDIA Titan (CUDA v OpenCL)
Scatter/Local matvec FP performance

\[ \mathbf{a}_L = A_L \mathbf{Z}_P \mathbf{g} \]

Sometimes we notice that weird things happen after compiler/driver upgrades…
a good reason to maximize choices.
Strange Things Afoot at the Circle K

With OCCA we can run in OpenCL or CUDA mode on NVIDIA GPUs.

NVIDIA Titan (CUDA v OpenCL)
Scatter/Local matvec FP performance

Polynomial Order

Sometimes we notice that weird things happen after compiler/driver upgrades…
a good reason to maximize choices.
OCCA: summary

OCCA:
✅ Lightweight API that hedges against shifts in programming models.
✅ Front ends: C++, C, Julia, Python, MATLAB, F90,…
✅ Currently works with OpenCL, CUDA, OpenMP, & pThreads back ends.

In progress:
✅ Version for Windows.
~ C# and Java front-ends.
~ Testing on the Xeon Phi.
~ New multi-threading languages will be added as they emerge and stabilize.

OCCA applications:
✅ Portability and performance for finite-difference, finite volume, & finite elements.
~ Scalability of fully accelerated algebraic multi-grid preconditioner setup/cycling.
~ ALMOND performance is a work in progress.

Comments:
✅ OCCA does reduce exposure to vendor-lock and churn in programming models.
✅ OCCA does provide an extra optimization direction for auto-tuning.
❌ OCCA does not provide high-level support for tuning and optimization.
OCCA: arXiv report

A report describing OCCA is available from arXiv.org
OCCA 2.0: github

Check out [OCCA2 on GitHub](https://github.com/tcew/OCCA2)
See the talk by John Levesque (Cray) this afternoon to learn about programming accelerators using OpenACC’s directive style programming model.
Random Extra Slides