git clone https://github.com/tcew/ATPESC18
“To be ready for supercomputing … … you are going to need to know … OpenCL or CUDA for GPUs.”

Tim Mattson, Intel.

… and what Tim’s NVIDIA rant unintentionally reveals about coding @CPUs …
Overview

Part 0: GPU Myths.

Part 1: NVIDIA Graphical Processing Unit

Part 2: Compute Unified Device Architecture (CUDA)
- NVIDIA’s threaded offload programming model.
- Hands on: area of the Mandelbrot

Part 3: Interlude on CUDA optimization.

Part 4: Portable programming models:
- Open Computing Language (OpenCL)
- Open Concurrent Computing Abstraction (OCCA)

Part 5: Hands on flow simulation:
- Prep: find a png image with white background.
- Run GPU flow simulation using your image.
- Visualize your results as a movie.
- Enter competition.
My Research...

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

Approximation Theory
Numerical Analysis
Numerical methods & Physical PDE Modeling
Accelerated Computing
High Performance Scalability

Basic science Application Industrial Scale

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

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

Flow modeling

MRI Guided Laser Interstitial Therapy

Electromagnetic scattering

Tsunami propagation

Global elastodynamics

Kinetic Models

Shock Flows

High order, GPU accelerated, Galerkin & discontinuous Galerkin solvers.

GPU programming tools & applications. Industrial collaboration.
Exascale Co-Design

The Center for Efficient Exascale Discretizations (CEED) is a co-design center within the U.S. Department of Energy (DOE) Exascale Computing Project (ECP) with the following goals:

- Help applications leverage future architectures by providing them with state-of-the-art discretization algorithms that better exploit the hardware and deliver a significant performance gain over conventional low-order methods.

- Collaborate with hardware vendors and software technologies projects to utilize and impact the upcoming exascale hardware and its software stack through CEED-developed proxies and miniapps.

- Provide an efficient and user-friendly unstructured PDE discretization component for the upcoming exascale software ecosystem.

CEED is a research partnership involving 30+ computational scientists from two DOE labs and five universities, including members of the Nek5000, MFEM, MAGMA, OCCA and PETSc projects. You can reach us by emailing ceed-users@llnl.gov or by leaving a comment in the CEED user forum.

The center's co-design efforts are organized in four interconnected R&D thrusts, focused on the following computational motifs and their performance on exascale hardware. See also our publications.

You will hear more about this from Tzanio Kolev (LLNL) & Mark Shepard (RPI) on 08/06
libParanumal: GPU enabled solvers

Goal: Drop in replacement for core elliptic & flow solver functionality of the Nek5000 simulation code.
Animation from Giannakopoulos et al and the SEAL lab http://fischerp.cs.illinois.edu/seal/

https://github.com/paranumal/libparanumal
**Resources:**

libParanumal release on 8.1.18

**Web page:** paranumal.com

**Blog:** paranumal.com/blog

**Twitter:** twitter.com/paranumal

**VT CEED BK paper:** P100

**BKS:** github.com/kswirydo/CEED-Ax

**VT INS2D+OCCA+Sub-cycling+AMG**

You can also send queries to: paranumal@vt.edu

libParanumal source code: https://github.com/paranumal/libparanumal
Today: slides & repos

Slides:

www.math.vt.edu/people/tcew/ATPESC18

Examples:

git clone https://github.com/tcew/ATPESC18

OCCA repo (0.2 branch version for this tutorial):


git clone https://github.com/libocca/occa -b 0.2
Part 0: GPU Reality Check
Myth #1: GPU 100x faster than CPU

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

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

http://www.nvidia.com/object/tesla-p100.html
https://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2_40-GHz

Majority of HPC codes are memory and network bound.
Myth #1: GPU 100x faster than CPU

NVIDIA V100 GPU: up to 900 GB/s (HBM2)

Intel Intel® Xeon® Platinum 8180 CPU: up to 119 GB/s bandwidth per socket

For well optimized bandwidth limited codes with more data than cache => one GPU is about 3.5x faster than a dual socket CPU.

https://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2.40-GHz
https://www.gamepc.com/shop/products?sku=900-2G503-0000-000-16GB

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

NVIDIA P100: ~$5K
Intel E7-8894 v4: ~$9K

Prices for non-consumer GPUs are carefully calibrated to be similar to CPU. [these are both premium high-end processors]
Myth #2: GPUs are expensive

NVIDIA V100 GPU: $7-12K each

Intel Intel® Xeon® Platinum 8180 CPU: ~$10K per socket

Super ridiculous top end CPU

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

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

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

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

http://www.nvidia.com/object/tesla-p100.html
https://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2_40-GHz
Myth #4.0: OpenACC is magic

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

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

It takes more than 3 hours to master GPUs…

… but we can discuss some of the basics …

… background to the Kokkos & Raja talks on easier GPU computing …

… there are many web resources …

… and nothing beats hands on.
Part 1: From CPU to GPU
Original design goals for CPUs:

- Make single threads very fast.
- Reduce latency through large caches.
- Predict, speculate.
Modern “CPU-Style” core design emphasizes individual thread performance.

Adapted from presentations by Andreas Klöckner and Kayvon Fatahalian

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

Fallout 4 Screenshot

Design goals for GPUs:

- Throughput matters and single threads do not.
- Hide memory latency through parallelism.
- Let programmer deal with “raw” storage hierarchy.
- Avoid high frequency clock speed:
  - Desirable for portable devices, consoles, laptops…

GPU: early example

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

Comparison of block diagram of vintage GPU and CPU

http://www.anandtech.com/show/2556/8
16 Maxwell cores each have four SIMD clusters with 32 ALUs. Data streams at ~56 GFLOAT/s and peak 4.6 TFLOP/s (SP)
CPU v GPU: fundamental difference #1

Each CPU core executes scalar or vector operations. Each GPU core only executes vector instructions.

CPU: Single Instruction Multiple Data (SIMD) parallelism through ILP & vector execution units.

GPU: SIMD parallel execution of all operations

http://en.wikichip.org/wiki/intel/microarchitectures/skylake

Compilers may need to be coaxed into generating vector instructions for CPU. Recall: “Performance, SIMD, Vectorization and Performance Tuning” talk by James Reindeer.
Compilers may need to be coaxed into generating vector instructions for CPU.

CPU v GPU: fundamental difference #2

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

Skylake core: 180 Integer registers and 168 floating point registers

Maxwell core: 16K registers

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

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

Nice summary:

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

NVIDIA’s Compute Unified Device Architecture
GPU programming model
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: Kepler GPU

GK110: 15 cores that cluster 192 FPU each.

Each Kepler core (SMX) has six SIMD clusters of 32 ALUs.
Data streams at ~70 GFLOAT/s and peak 4+ TFLOP/s (SP)

http://www.tomshardware.com/reviews/geforce-gtx-titan-gk110-review,3438.html
16 Maxwell cores each have four SIMD clusters with 32 ALUs. Data streams at ~56 GFLOAT/s and peak 4.6 TFLOP/s (SP)
GP100 was built to be the highest performing parallel computing processor in the world to address the needs of the GPU accelerated computing markets serviced by our Tesla P100 accelerator platform. Like previous Tesla-class GPUs, GP100 is composed of an array of Graphics Processing Clusters (GPCs), Texture Processing Clusters (TPCs), Streaming Multiprocessors (SMs), and memory controllers. A full GP100 consists of six GPCs, 60 Pascal SMs, 30 TPCs (each including two SMs), and eight 512-bit memory controllers (4096 bits total).

Each GPC inside GP100 has ten SMs. Each SM has 64 CUDA Cores and four texture units. With 60 SMs, GP100 has a total of 3840 single precision CUDA Cores and 240 texture units. Each memory controller is attached to 512 KB of L2 cache, and each HBM2 DRAM stack is controlled by a pair of memory controllers. The full GPU includes a total of 4096 KB of L2 cache.

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

Professional NVIDIA Pascal GV100 architecture with 84 cores using 12nm fab size

HBM2 memory with **900 GB/s bandwidth** and 5376 ALUs with peak 15.7 TFLOP/s (SP) **7.8 TFLOP/s (DP)**

84 Volta GV100 cores each with four SIMD clusters of 16 ALUs (5120 Total).
GPU: trends in FPU Clusters

The FPU clusters ("core") in 4 NVIDIA generations

2007: G80

2008: Tesla

2010: Fermi

Q4 2012: Kepler GK110

SMX:


The FPU cluster sizes have ballooned: 16 - 24 - 32 - 192
but the shared memory and register file have not grown accordingly.
Trend: smaller die process yields more space for shared memory and registers.

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

GPU: Kepler to Maxwell to Pascal

http://www.ubergizmo.com/2014/02/nvidia-maxwell-gpu-for-geforce-cards/
Trend: smaller die process yields space for additional half precision “tensor-cores”

The FPU clusters (“core”) in 2 latest NVIDIA processor FP64 heavy architectures

GPU: Pascal to Volta

http://www.ubergizmo.com/2014/02/nvidia-maxwell-gpu-for-geforce-cards/
GPU: a decade of core architectures

In 10 years the NVIDIA core count & core architecture has scaled remarkably well ...

... their roadmap has been signposted and CUDA codes have scaled ...
... yet somehow many major HPC codes have not adapted.
Concurrent Cloud Computing: installing occaBench for V100
February 4, 2018 | Tim Wielandt

Overview: This week we have been experimenting with instances on Amazon AWS and PaperSpace that come equipped with NVIDIA V100 GPUs. These GPUs are hot properties and not widely available, so we had to request special access to V100 equipped instances on both systems. Both AWS and PaperSpace responded quickly to our request. The PaperSpace support team was also noted by their responsive, patient, and helpful getting through some minor technical issues.

Note: this article is not an endorsement of these companies or their products, we are just providing an insight into our experience getting, started on their systems. Your mileage may vary. In our experience both systems were very similar once the instances were provisioned.

Configuration: On AWS we set up a p2.2xlarge instance and paperSpace we set up a V100 machine. In both cases we chose Ubuntu 16.04, for no other reason than familiarity with Ubuntu/Linux.

On the PaperSpace system I was able to get back a dual x86, NVIDIA driver, the NVIDIA CUDA SDK, and some basic tools installed with:

```bash
# basics
sudo apt-get update
sudo apt-get install -y build-essential gcc
sudo apt-get install emake
```

http://paranumal.com/blog
Part 2b: GPU programming with CUDA Threading

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

CUDA

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

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

The programmer explicitly moves data between HOST and DEVICE

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

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

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

The GPU architecture admits a natural parallel threading model

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

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

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

```c
#include "cuda.h"

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

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

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

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

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

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

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

Note the `.cu` file extension.
We use NVIDIA's CUDA Compiler `nvcc` to compile `.cu` files.
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`
Before jumping into how to write a CUDA kernel we consider first a serial function that fills an array with entries 0:N-1.

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

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

We can think of splitting the n-loop into tiles of size 4: n=t+4b.
Here: block dimension = 4 and grid dimension = 5.
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: $N \leq gridDim*blockDim$. 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: $N \leq gridDim.x \times blockDim.x$
**CUDA: multi-dimensional thread rank**

Each thread can determine its (multi-dimensional) rank with respect to both its rank in the thread-block and the rank of the thread-block itself.

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

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

*Best practice: avoid frequent branching based on threadIdx or blockIdx.*

* three dimensional grid of thread-blocks supported as of CUDA 2.*
## CUDA: limitations

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

<table>
<thead>
<tr>
<th>Technical specifications</th>
<th>Compute capability (version)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td>2</td>
</tr>
<tr>
<td>Maximum x-, y-, or z-dimension of a grid of thread blocks</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum dimensionality of thread block</td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td></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></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td></td>
</tr>
</tbody>
</table>

Table credit: CUDA wikipedia page (http://en.wikipedia.org/wiki/CUDA)
void tiledSerialSimpleKernel(int N, float *d_a){

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

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

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

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

    }

}

}

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

We also assume that: N <= gridDim.x*blockDim.x
CUDA: simple array operation kernel

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

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

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

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

Key observation: the loops are implicitly executed by thread parallelism and do not appear in the CUDA kernel code.
\[ c_n = a_n + b_n \text{ for } n = 0, \ldots, N - 1 \]
CUDA: offload model

The programmer explicitly moves data between HOST and DEVICE

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

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

https://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-780
Code Along: diving straight into CUDA

Code along demo:

CUDA code to add two vectors together from scratch !!!

You can find a pre-made version here:

https://github.com/tcew/ATPESC18/tree/master/examples/cuda/addVectors

Wacky CUDA syntax used:

Thread rank and size info: threadIdx.x, blockIdx.x, blockDim.x
DEVICE function (kernel) annotation: __global__
Allocating/freeing a DEVICE array: cudaMalloc, cudaFree
Copy data between DEVICE and HOST: cudaMemcpy
Copy direction: cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost

Kernel launch : addVectorsKernel <<< dimGrid, dimBlock >>> (N, d_a, d_b, d_c);
Adapt the CUDA code to reverse the entries in an array:

\[ b_n = a_{(N-1-n)} \] for \( n = 0, \ldots, N - 1 \)

Start with your code along code, or use the pre-baked version:

https://github.com/tcew/ATPESC18/tree/master/examples/cuda/addVectors

Things to pay attention to:

1. Make sure you copy back the correct CUDA DEVICE array to the HOST.
2. How many threads should you use to avoid read-write race conflicts?
3. Change the number of threads.
4. Print the whole b array after the kernel.

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

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

# clone the examples on the login node:
git clone https://github.com/tcew/ATPESC18

# if you haven’t already done so, queue an interactive job request:
qsub -A ATPESC2018 -I -n 1 -t 120 -q training

# find the source
cd ATPESC18/examples/cuda/simple

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

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

Make sure you can complete this exercise now if possible!
Source code: https://github.com/tcew/ATPESC18/examples/cuda/simple
CUDA: multi-dimensional tiled serial function

CUDA supports up to 3 nested outer “block” loops, with a sequence of 3 nested inner “thread” loops

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

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

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

                // Perform action based on thread-ranks
                ...
            }
        }
    }
}
```

Key observation: the iterations in each iteration are assumed to be independent
// HOST code to queue kernel
dim3 dimGrid(GX, GY, GZ), dimBlock(BX, BY, BZ);
multidKernel <<< dimGrid, dimBlock >>> (N, d_a);

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

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

    operations based on thread ranks;
}

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

We consider a more substantial example: solving the Poisson problem.

Elliptic Poisson problem:

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

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

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

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

Discrete Poisson problem (assuming Cartesian grid):

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

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

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

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

\[u_{ji} = 0 \quad \text{for } i = 0, N + 1 \text{ or } j = 0, N + 1\]
Jacobi iteration for discrete Poisson problem:

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

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

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:

\[\varepsilon := \sqrt{\sum_{i=1}^{i=N} \sum_{j=1}^{j=N} \left( u_{ji}^{k+1} - u_{ji}^k \right)^2} > tol\]
For the iterate step we note:
each node can update independently for maximum parallelism.

Iterate:

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

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

**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]);
        }
    }
}
```

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 NxN array of threads and change leave the edge nodes unchanged.

At the start we set: \( rhs = -\delta^2 \delta f \)
CUDA: parallel Jacobi iteration

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

**Iterate:**

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

Note: we use an NxN 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: \( rhs = -\delta^2\delta f \)

[https://github.com/tcew/ATPESC18/tree/master/examples/cuda/jacobi](https://github.com/tcew/ATPESC18/tree/master/examples/cuda/jacobi)
To make this more parallel we need to split the termination into CUDA thread-blocks:

Reduction:

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

Block reduction (B blocks)

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

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

Next we need to distribute the inner sum work over the threads in each of the B thread-blocks.
CUDA: parallel reduction

Standard tree reduction at the thread-block level!!

CUDA partial reduction kernel:

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

__shared__ datafloat s_blocksum[BDIM];

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

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

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

Standard tree reduction at the thread-block level!!

CUDA partial reduction kernel:

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

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

Here the __shared__ array is read/writeable only by threads in the same thread-block. All threads in the thread-block have to enter the __syncthreads() before any of them can continue.
The .cu file contains both the partialSum reduction DEVICE kernel and the HOST code. Note: the HOST code includes event based timing of the kernel execution.
Hands On #2: CUDA Mandelbrot Area

Converting the Mandelbrot example from Tim Mattson’s talk to CUDA.

#1. Retrieve the files:

git clone https://github.com/tcew/ATPESC18

#2. Complete the skeleton code

ATPESC18/handsOn/mandelbrot/mandelbrot.cu

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

#3. Hints:

To compile (on cooley.alcf.anl.gov):

    nvcc -arch=sm_30  -o mandelbrot mandelbrot.cu -lm

To run (on a cooley compute node):

    ./mandelbrot

Useful CUDA keywords (google for details):

   thread rank: threadIdx.x, threadIdx.y, blockIdx.x, blockIdx.y, blockDim
   keywords:    __device__, __global__

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

Dark Arts Indeed
Classic Definition of “Supercomputer”

This is a well known definition of a “supercomputer”

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

Ken Batcher*

*Attribution is a little cloudy: *possibly Seymour Cray
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
### CUDA: Limitations

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

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

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

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

**Impact of Varying Block Size**

![Graph showing impact of varying block size on occupancy](image)

**Impact of Varying Register Count Per Thread**

![Graph showing impact of varying register count on occupancy](image)

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

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

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

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

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

BAD: all threads in the SIMD group access the same bank resulting in serialization.
High end NVIDIA GPUs either have 256 or 384 bit wide memory bus to device memory.

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

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

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

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

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

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

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

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

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

Work-items

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

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

Data Transfer

- Low individual bandwidth and high latency.
- Coalesced memory access on contiguous and aligned work-items.
Exposing vectorization / SIMD parallelism are vital in both architectures.
Part 4: Portable programming models
Blinky was a MacBook Pro with discrete NVIDIA GPU and OpenMP, OpenCL, CUDA …

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

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

Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Part 4a: Open Compute Language (OpenCL)
Quick-reference-card for OpenCL 2.0: (link)
OpenCL: standard for multicore

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

- **Diverse industry participation**
  - Processor vendors, system OEMs, middleware vendors, application developers

- **Many industry-leading experts involved in OpenCL’s design**
  - A healthy diversity of industry perspectives

- **Apple made initial proposal and is very active in the working group**
  - Serving as specification editor
OpenCL: why?

Processor Parallelism

CPUs
Multiple cores driving performance increases

Emerging Intersection

Heterogeneous Computing

GPUs
Increasingly general purpose data-parallel 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
CUDA and OpenCL are competing standards for GPGPU programming.
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: thread indexing

OpenCL is **very** closely related to CUDA

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

OpenCL is **very** closely related to CUDA

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong> function</td>
<td>__kernel function</td>
</tr>
<tr>
<td><strong>device</strong> function</td>
<td>function</td>
</tr>
<tr>
<td><strong>constant</strong> variable</td>
<td>__constant variable</td>
</tr>
<tr>
<td><strong>device</strong> variable</td>
<td>__global variable</td>
</tr>
<tr>
<td><strong>shared</strong> variable</td>
<td>__local variable</td>
</tr>
</tbody>
</table>
The rapid development of OpenCL helps explain the similarities between CUDA and OpenCL. Again, the memory model for CUDA and OpenCL are very similar.

CUDA memory model:
- Global Memory
- Constant Memory
- Texture Memory
- Registers
- Shared Memory
- Thread (0,0) and Thread (1,0)

OpenCL memory model:
- Private Memory
- Work-Item
- Local Memory
- Global/Constant Memory
- Host Memory

Image system not shown in 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

With flexibility can come complexity.
The include files ...

CUDA

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

OpenCL

```c
#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.

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);
...

... 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.

... 

// not necessary although you may wish to use cudaStreamCreate  
...

// make compute context on device (pfn_notify is an error callback function)  
cl_command_queue queue =  
   clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &err);
I wasn’t kidding about flexibility.

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:

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

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

   • `clCreateProgramWithSource`

3. Compile and build the “program”:

   • `clBuildProgram`

4. Check for compilation errors:

   • `clGetProgramBuildInfo`

5. Build executable kernel:

   • `clCreateKernel`
OpenCL: building a kernel

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

...  

// not necessary  
// nvcc compiles the kernel code when you compile the executable  
...

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

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

/* omitted error checking */  
...

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

CUDA uses block sizes + number of blocks.
OpenCL uses block sizes and global number of threads.
The kernel programming languages are similar:

**CUDA**

```c
__global__ void simpleKernel(int N, 
    float *a)
{
    /* get thread coordinates */
    int i = threadIdx.x + 
    blockIdx.x*blockDim.x;
    /* do simple task */
    if(i<N)
        a[i] = i;
}
```

**OpenCL**

```c
__kernel void simpleKernel(int N, 
    __global float *a)
{
    /* get thread coordinates */
    int i = get_global_id(0);
    /* do simple task */
    if(i<N)
        a[i] = i;
}
```

Some minor differences in syntax & identifiers
OpenCL: summary

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

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

- On the other hand:
  - Runtime compilation adds several optimization opportunities without templating.
  - OpenCL is library based, so no special compilers are required.
  - Vendor independence is important.
OpenCL: comparing Jacobi kernels

Iterate:

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

Serial kernel:

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

CUDA kernel:

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

OpenCL kernel:

```c
?? void jacobi( ?? const int N,
               ?? const double *rhs,
               ?? const double *u,
               ?? double *newu) {
    // Get thread indices
    const int i = ??;
    const int j = ??;
    if((i < N) && (j < N)) {
        ??;
    }
}
```
OpenCL: comparing Jacobi kernels

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

Iterate:

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

Serial kernel:

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

CUDA kernel:

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

OpenCL kernel:

```c
__kernel void jacobi(const int N,
                     __global const double *rhs,
                     __global const double *u,
                     __global double *newu){
    // Get thread indices
    const int i = get_global_id(0);
    const int j = get_global_id(1);
    if((i < N) && (j < N)){
        // Get linear index into (N+2)x(N+2) grid
        const int id = (j + 1)*(N + 2) + (i + 1);
        newu[id] = 0.25f*(rhs[id]
            + u[id - (N+2)]
            + u[id + (N+2)]
            + u[id - 1]
            + u[id + 1]);
    }
}
```

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

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

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

OpenCL partial reduction kernel:

```c
__kernel void partialReduce( const int entries,
                            ?? const double *u,
                            ?? const double *newu,
                            ?? double *blocksum){

  __local double s_blocksum[BDIM];
  const int id = get_global_id();
  int alive = ??;
  int t = ??;
  s_blocksum[t] = 0;
  // load global data into local memory if in range
  if(id < entries){
    const double diff = u[id] - newu[id];
    s_blocksum[t] = diff*diff;
  }
  while(alive>1){
    barrier(CLK_LOCAL_MEMFENCE); // barrier (make sure s_blocksum is ready)
    alive /= 2;
    if(t < alive) s_blocksum[t] += s_blocksum[t+alive];
  }
  if(t==0) blocksum[get_group_id(0)] = s_blocksum[0];
}
```
Part 4b: Portability alternatives to OpenCL
Need an efficient, durable, portable, open-source, vendor-independent approach for many-core programming
Existing CUDA code can be ported to other frameworks.

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

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

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

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

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

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

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

```cpp
double a[100];
#pragma acc enter data copyin(a)
// OpenACC code
#pragma acc exit data copyout(a)

class Matrix {
  double *v;
  int len
  Matrix(int n) {
    len = n;
    v = new double[len];
    #pragma acc enter data create(v[0:len])
  }
  ~Matrix() {
    #pragma acc exit data delete(v[0:len])
    delete[] v;
  }
};
```

Code taken from:
WHAT'S NEW IN OPENACC 2.0 AND OPENMP 4.0, GTC '14
Portability: ease of use

My opinion on “Maturity” balanced against “Ease of use” for portable many-core programming

Still need an easy, efficient, durable, portable, open-source, vendor-independent approach for many-core programming
### Step Back: MPI + X?

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI +</td>
<td>MPI</td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI +</td>
<td>OpenMP</td>
<td>pThreads</td>
</tr>
<tr>
<td>CUDA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI +</td>
<td>OpenCL</td>
<td></td>
</tr>
<tr>
<td>TBB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI +</td>
<td>Cilk Plus</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Part 4c: OCCA
Open Concurrent Compute Abstraction
OCCA: easy portability

What is OCCA?

In a nutshell, OCCA (like oca-rina) is an open-source library which aims to

- Make it easy to program different types of devices (e.g. CPU, GPU, FPGA)
- Provide a unified API for interacting with backend device APIs (e.g. OpenMP, CUDA, OpenCL)
- Use just-in-time compilation to build backend kernels
- Provide a kernel language, a minor extension to C, to abstract programming for each backend

Quick Navigation

github.com/libocca/occa
libocca.org
GPU: Marmite® of the HPC world

People love or hate GPUs & the source is messy
Open Concurrent Compute Abstraction (OCCA)

Goals:

• Portability.
• Native code performance.
• Insulate simulation codes from HPC churn.

• Reduce the Marmite-ness of GPUs.

Design Principles:

• Simplicity.
• Unified interface.
• Limited dependencies.

• Explicit offload compute model.
• Kernel language: lightly annotated C.

Codes Exploring OCCA:

• libParanumal, ESDGSEM, NUMA, GNuMe, Nek5K*, libCEED, MFEM, laghos…

OCCA: OKL: a unified language for parallel architectures, David Medina PhD. 
http://libocca.org
What does OCCA \textbf{not} do?

Open Concurrent Compute Architecture, no magic unicorns.

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

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

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

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

Low-level code:
  • We do not circumvent the vendor compilers.
Building the OCCA library:

```
git clone https://github.com/libocca/occa -b 0.2
cd occa
export OCCA_DIR=`pwd`
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OCCA_DIR/lib
make -j
```

Building example:

```
cd examples/addVector/cpp
make
./main
```

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

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

#### Numerous approaches to portability

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

*OCCA emphasis: lightweight and extensible.*

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

Description

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

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

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

The concept of iterating over groups and items is simple
Outer-loops

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

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

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

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

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

OKL
```
Outer-loops

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

```c
kernel void kernelName(...){
    for(outer){
        for(inner){
        }
    }
    for(outer){
        for(inner){
        }
    }
    for(outer){
        for(inner){
        }
    }
    for(outer){
        for(inner){
        }
    }
    for(outer){
        for(inner){
        }
    }
}
```
**OCCA:OKL kernel language**

### Shared memory

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

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

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

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

### Exclusive memory

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

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

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

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

Local barriers are auto-inserted
Shared memory

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

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

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

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

Exclusive memory (similar to threadPrivate)

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

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

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

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

Description

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

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

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

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

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

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

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

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

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

    addVectors(N, o_a, o_b, o_ab);

    o_ab.copyTo(ab);

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

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

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

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

    device.setup("OpenCL", 0, 0); // (Platform, Device) = (0, 0)
    o_a = device.malloc(N*sizeof(float));
    o_b = device.malloc(N*sizeof(float));
    o_ab = device.malloc(N*sizeof(float));

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

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

    for (int i = 0; i < 5; ++i)
    {
        std::cout << i << " " << ab[i] << 'n';
    }
}
All the HOST codes use the same kernel.

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

Source-to-Source Compilation

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

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

Source File \(\rightarrow\) “Content” \(\rightarrow\) OKL/OFL Parser \(\rightarrow\) Source File 2 \(\rightarrow\) Run-time Compilation

Custom compilation tools tailored for code manipulation and analysis
Behind the Scenes: caching and hashing

**Source-to-Source Compilation**

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

```
#define N 10

int i = N;
```

```
int i = 10;
```

Custom compilation tools tailored for code manipulation and analysis
OCCA Infrastructure

Source-to-Source Compilation

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

#define N 10

int i = N;

int i = 10;

int i = N;

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

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

Custom compilation tools tailored for code manipulation and analysis
OCCA Infrastructure

Source-to-Source Compilation

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

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

Custom compilation tools tailored for code manipulation and analysis
OCCA Infrastructure

Source-to-Source Compilation

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

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

Custom compilation tools tailored for code manipulation and analysis
OCCA Infrastructure

Source-to-Source Compilation

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

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

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

# login node: clone the OCCA repo

git clone https://github.com/libocca/occa -b 0.2

# compute node: cd to the OCCA directory

cd occa

# build OCCA

make -j

# add OCCA_DIR to env and add dynamic library path
export OCCA_DIR=`pwd`
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OCCA_DIR/lib

#2. build the OCCA LBM code:

# login node: clone the ATPESC18 repo

git clone https://github.com/tcew/ATPESC18

# compute node: cd to the lbm directory

cd ATPESC18/handsOn/lbm

# build OCCA lbm solver

make -f makefile.occa

#3. login node: save png image with white background to the lbm directory:

#4. run the lbm code with your png image

# (using 400 as a flow volume threshold)

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

Details: a D2Q9 lattice Boltzmann method is implemented using an OCCA update kernel that uses a single thread to updates the 9 particle densities at each lattice node.
**OCCA Flow Simulation: instructions**

#5. the lbm code generates bah##.png image files:

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

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

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

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

#7. Find out what compute modes are available:

$OCCA_DIR/bin/occainfo

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

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

#9. re-make the executable:

```bash
make -f makefile.occa
```

#10. rerun

```bash
./occaLBM yourImageName.png 400
```

# Do you notice a speed change?

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

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

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

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

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

Discontinuous Galerkin for RTM

![Discontinuous Galerkin for RTM](image)

Est. GFLOPS vs Polynomial Order for different platforms:
- OpenCL (AMD Tahiti)
- OpenCL (NVIDIA Titan)
- CUDA (NVIDIA Titan)
- OpenCL-AMD
- OpenCL-Intel
- OpenMP (icpc)
- OpenMP (g++)
### OCCA: apps & benchmarks

Lattice Boltzmann Method in Core Sample Analysis

Comparison across platforms (Normalized with original code)

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

Comparison across platforms (Normalized with OCCA::OpenMP)

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

Discontinuous Galerkin for shallow water equations

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

Algebraic multigrid for elliptic problems

Setup Time

Solve Time

M. unknowns per sec

# of unknowns

CUDA on Titan
OpenCL on Titan
OpenCL on Tahiti
OpenCL on Intel i7
OpenMP on Intel i7
Monte Carlo for neutronics
Collaborations with Argonne National Lab

XSBench

RSBench

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

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

Backprop

BFS

Needleman

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

https://github.com/dmed256/OCCA-Benchmarks
Concurrent Cloud Computing: installing occaBench for V100
February 6, 2018 / Tim Warburton

Overview: This week we have been experimenting with instances on Amazon AWS and Paperspace that come equipped with NVIDIA V100 GPUs. These GPUs are hot properties and not widely available, so we had to request special access to V100 equipped instances on both systems. Both AWS and Paperspace responded quickly to our requests. The Paperspace support team was also incredibly responsive, patient, and helpful getting through some minor technical issues.

Note: this article is not an endorsement of these companies or their products, we are just providing an insight into our experience getting started on their systems. Your mileage may vary. In our experience both systems were very similar once the instances were provisioned.

Configuration: On AWS we set up a p3.2xlarge instance and on Paperspace we set up a V100 machine. In both cases we chose Ubuntu 16.04, for no other reason than familiarity with Ubuntu/Linux.

For my own development needs I have switched to use GPU cloud servers including: gpueater (AMD GPUs), Amazon AWS (NVIDIA GPUs incl. V100), paperspace (NVIDIA GPUs incl. V100).
tim.warburton@vt.edu