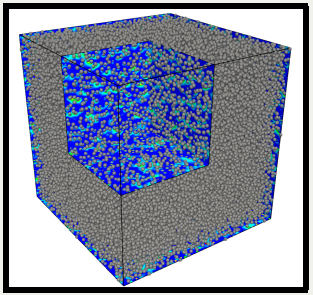
AC290: Extreme Computing
Intro to GPU Computing

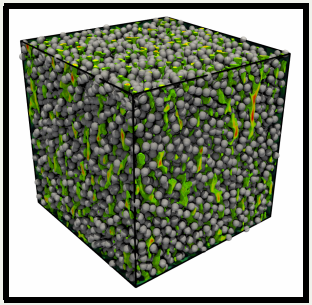
Daniel Willen

The MITRE Corporation

About Me

- Ph.D., Mechanical Engineering, Johns Hopkins University 2018
  - Resolved-particle simulations of disperse multiphase flows
    - Formation of rain in the atmosphere
    - Erosion and transport of material in atmosphere/waterways
    - Fluidized beds (chemical reactors)
    - Thermal spray coating





About Me

- Thesis work extended single-GPU-centric CFD code to use many-GPUs
  - GPU-centric: GPU is used as the primary compute unit
  - many-GPUs: Use more than one to speed-up/scale-up; requires MPI
  - Original code was $\sim 70\times$ faster than legacy CPU code
- Several methods for disperse flows:
  - Two-fluid (continuum): Particles are small; treat as a separate continuous phase
  - Point-particle/DEM: Particles are point-forces in the flow
  - Resolved: The particle-fluid interaction is fully modeled
- Results: Up to one million particles on 216 K-40 GPUs at MARCC
  - State-of-the-art $O(100,000)$ using CPUs on entire supercomputer

Overview

- Brief GPU Introduction
- Cuda Basics – `Hello world`
  - Compiling and running, program flow control, threading and parallelization
- Cuda Basics – Arrays
  - Memory management, race conditions and reductions
- Simple Jacobi example
  - Bringing it all together
- Next time: GPU architecture, shared memory, Cuda-aware MPI

Objectives

1. Learn the basics of GPU computing
   - We'll use a simple example so we don't have the overhead of LBM
   - Hands-on – I'll be skipping technical details today
2. Write a simple GPU code to solve the heat equation
3. Gain an appreciation of the programming complexity that GPUs add

Brief GPU Introduction

GPUs are composed of **threads**

- A thread performs some operation ($+, -, *, /$, read, write, etc.)

Compared to CPUs...

- Many more cores (1000s vs 10s) – potential to do a lot of work
- GPU cores are less independent (branching and logical operations are bad)
- 'Brawn' vs. 'brains'
- Massive parallelization of floating point operations
- Small onboard memory (10s of GB) (Due to type memory, bandwidth, history...)

GPU parallelization paradigms:

1. Host-device (CPU-GPU) communication is very slow
2. Avoid thread divergence, even at the expense of doing more work
3. Data locality is key

What is Cuda?

- Language used for GPU programming (`C`, `C++`)
- Adds extra programming overhead → codes are longer, more complicated

- Log into Odyssey
- Questions so far?

Hello world Example

hello-world/simple

```
/* Cuda 'hello world' pseudo-code */

// GPU kernel definition
__global__ void hello_world_kernel(int
    dev) {
  printf("Hello world from device %d!\n",
      dev);
}

// Main code
int main(void) {
  int dev = 0;
  hello_world_kernel<<<1,1>>>(dev);
  cudaDeviceSynchronize();
}
```

- `__global__`: Called from CPU
- `__device__`: Called from GPU
- `dev`: Sets the specific GPU to use based on compute node
- `<<<1,1>>>`: Kernel execution configuration. More on this later!
- `cudaDeviceSynchronize()`: Ensures the kernel finishes executing before moving to the next step

Compiling and Running

- Compile with Nvidia CUDA Compiler `nvcc` (similar to `gcc`):
  `nvcc hello_world.cu -o hello_world`
  - Need to `module load cuda`
- Running directly on a compute node
  - Only in interactive mode – otherwise no GPU!
  - `srun --pty -p gpu -t 6:00 --mem 8000 --gres=gpu:1 /bin/bash`
  - This sets environment variable `CUDA_VISIBLE_DEVICES`
  - If `CUDA_VISIBLE_DEVICES=1,2`, then `cudaSetDevice(0);` sets device `1`
  - `./hello_world`
- Running with Slurm
  - `CUDA_VISIBLE_DEVICES` is set automatically

Cuda and Makefile

Makefile

```
all:
	nvcc hello_world.cu -o hello_world

debug:
	nvcc -g -G hello_world.cu -o hello_world

clean:
	rm -f hello_world
```

Cuda and Slurm

submit.sh

```
#!/bin/sh
# Run this with ‘sbatch submit.sh’

#SBATCH --partition=gpu
#SBATCH --time=1:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=4
#SBATCH --gres=gpu:1
#SBATCH --job-name=test
#SBATCH --output=test.out

# Can run commands here, or use ‘echo’ to look at CUDA_VISIBLE_DEVICES
hostname
./hello_world
```

Compiling and Running

1. **What happens if we remove `cudaDeviceSynchronize()`?**
2. What happens if we change `<<<1,1>>>`?

1) Synchronization


```
1 cudaMemcpy(...);              // Blocking
2 GPU_kernel_1<<<nx,ny>>>();    // Non-blocking
3 CPU_function_1();             // Blocking
4 GPU_kernel_2<<<nx,ny>>>();    // Non-blocking*
5 cudaDeviceSynchronize();      // Blocking
6 CPU_function_2();
```

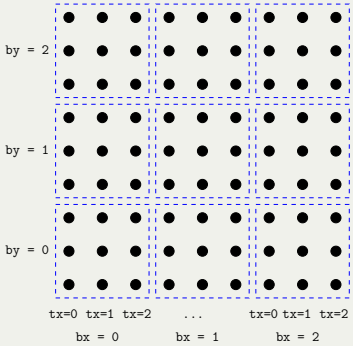
- Program control returns to CPU after kernel is launched
- Exceptions for memory management
- This allows asynchronous execution, but need to be careful!
  - Cuda streams

Compiling and Running

1. What happens if we remove `cudaDeviceSynchronize`?
2. **What happens if we change `<<<1,1>>>`?**
   - `<<<num_blocks, dim_blocks>>>`

2) Threads – Kernel Config

- GPU architecture lends itself to organizing threads in a grid
- The grid is decomposed into thread blocks
- Each thread block is composed of threads
- Individual threads execute operations on the data
- Data index is:
  - `ti = blockIdx.x*blockDim.x + threadIdx.x`
  - `tj = blockIdx.y*blockDim.y + threadIdx.y`
- Here, `blockDim.x = blockDim.y = 3`
- Note that the grid size does not have to be a multiple of the block dimensions!
- `<<<num_blocks, dim_blocks>>>` = `<<<(3,3), (3,3)>>>`



Threading Demonstration

hello-world/kernel_config

```
// GPU kernel definition
__global__ void hello_world_kernel(int nx,
    int ny) {
  // Index of the current thread
  int ti = blockDim.x*blockIdx.x + threadIdx.x;
  int tj = blockDim.y*blockIdx.y + threadIdx.y;

  if (ti < nx && tj < ny) {
    printf("(%d, %d) >> Hello world from
        device %d!\n", ti, tj, dev);
  }
}
```

```
// Main code
int main(void) {
  int nx = 4; int ny = 6;
  int tx = 2; int ty = 3;
  // Calculate these!
  int bx = 2; int by = 2;

  dim3 dim_blocks(tx, ty);
  dim3 num_blocks(bx, by);

  hello_world_kernel<<<num_blocks,
      dim_blocks>>>(nx, ny);
}
```

Memory Management

- Motivation: Program reads in data from file and operates on it with GPU
- GPUs cannot* read directly from disk
- Need to:
  1. Initialize data on CPU
  2. Allocate memory on GPU
  3. Copy memory from CPU to GPU (this is $$$!)
- When we write to file:
  4. Copy memory from GPU to CPU (this is $$$!)
- Note: there are *many* ways to deal with memory in Cuda (unified memory, `cudaMallocHost`, ...)

* There is active research in this area

Memory Management


```
// Declare pointers to CPU (host) and GPU (device) arrays
double *array, *_array; // the '_' denotes an array on the device

// Allocate and initialize memory on CPU
array = (double*) malloc(length * sizeof(double));
// Initialize data -- I/O, initial condition, etc.

// Allocate and copy memory on GPU
cudaMalloc(&_array, length * sizeof(double));
cudaMemcpy(_array, array, length * sizeof(double), cudaMemcpyHostToDevice);

// Do something crazy with the data, then copy it back
cudaMemcpy(array, _array, length * sizeof(double), cudaMemcpyDeviceToHost);

// Free
free(array);
cudaFree(_array);
```

Memory Demonstration

- Example: Parallel printing
- Notice that the results differ each time! This is known as a race condition

Race Condition

- Notice that printing in parallel doesn't return the same order each time
- This is because threads are not guaranteed to execute in any particular order
- What would happen when we try to find the sum of an array?

Serial – CPU

```
// a[i] = 1; i < N
double sum = 0;
for (int i = 0; i < N; i++) {
  sum += array[i];
}
printf("sum = %d"; sum); // What is this?
```

Parallel – GPU

```
// a[i] = 1; i < N
double *sum = 0;
array_sum<<<N, 1>>>(array, N, &sum);
printf("sum = %d"; sum); // What is this?

// Where the array_sum is
int ti = blockDim.x*blockIdx.x +
    threadIdx.x;
sum += array[ti];
```

Race Conditions

This behavior is undefined:

1. Thread #1 reads value from `sum`
2. Thread #1 increments value
3. Thread #2 reads value from `sum`
4. Thread #1 writes value of `sum`
5. Thread #2 increments value
6. Thread #2 writes value of `sum`

→ Thread #2 hasn't accounted for Thread #1!
→ Parallel reduction (e.g., sum, max, min) are complicated

Motivation

- 2-D heat equation:

$$\frac{\partial u}{\partial t} = D\nabla^2 u$$

  on the domain $0 \leq x, y \leq \pi$. Note that u is a function of x, y, and t.

- Homogeneous Dirichlet boundary conditions:

$$u(0, y) = u(\pi, y) = u(x, 0) = u(x, \pi) = 0$$

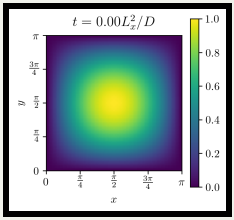
- Initial condition:

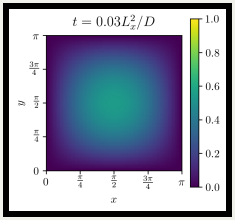
$$u(x, y, t = 0) = \sin(x) \sin(y)$$

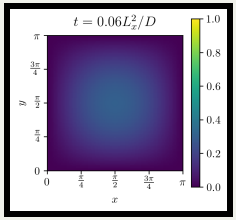
Analytic Solution

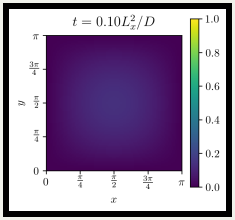
- Expand in a Fourier series, take a Laplace transform, use Green's function...

$$u(x, y) = \sin(x) \sin(y) \exp(-2Dt)$$









Increasing time $\rightarrow$

Numerical Solution

Finite differences:

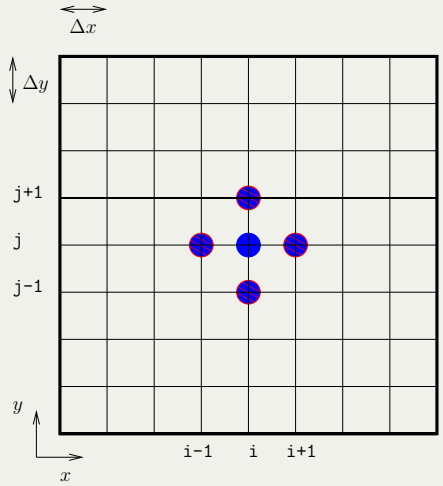
- Second-order central difference in space

$$\nabla^2 u^{(n)} = \frac{u^{(n)}_{i+1,j} + u^{(n)}_{i-1,j} - 2u^{(n)}_{i,j}}{\Delta x^2} + \frac{u^{(n)}_{i,j+1} + u^{(n)}_{i,j-1} - 2u^{(n)}_{i,j}}{\Delta y^2} + O(\Delta x^2, \Delta y^2)$$

- First-order forward difference in time (explicit)

$$\frac{\partial u}{\partial t} = \frac{u^{(n+1)}_{i,j} - u^{(n)}_{i,j}}{\Delta t} + O(\Delta t)$$

- Solve for $u^{(n)}_{i,j} \equiv u(i\Delta x, j\Delta y, n\Delta t)$



Numerical Solution

Rearranging (and assuming $\Delta x = \Delta y$) ...

$$u_{i,j}^{(n+1)} = u_{i,j}^{(n)} + \frac{D\Delta t}{\Delta x^2}\left[u_{i+1,j}^{(n)} + u_{i-1,j}^{(n)} + u_{i,j+1}^{(n)} + u_{i,j-1}^{(n)} - 4u_{i,j}^{(n)}\right]$$

Recall:

- Solving the Laplace problem $\nabla^2 u = 0$ can be done by "relaxing" the unsteady heat equation $\frac{\partial u}{\partial t} = \nabla^2 u$ to its steady state, $\frac{\partial u}{\partial t} = 0$
- This is like solving the linear system $\mathbf{Ax} = \mathbf{0}$ using the stencil above (introducing an "artificial" time)
  - Diffusive stability condition: $\Delta t \leq \Delta x^2/4D$
- This is an example of the Jacobi method
  - The Jacobi method is a method for solving systems of linear equations
  - Slow, but improve with Gauss-Siedel, Schedule Over-Relaxation (SOR), ...

Serial Solution


```
/* Psuedo-code for serial Jacobi solver */
// 1) Specify domain size
// 2) Initialize variables, set initial condition
// 3) pref = D*dt/(dx*dx)

for (t = 0; t < t_end; t += dt) {    // Loop over time
  for (i = 1; i < (nx - 1); i++) {   // Loop over x (no boundaries)
    for (j = 1; j < (ny - 1); j++) { // Loop over y (no boundaries)
      u_new[i,j] = pref*(u[i+1,j] + u[i-1,j] + u[i,j+1] + u[i,j-1]) +
                         (1. - 4*pref)*u[i,j];
    }
  }
  // Store u_new->u
}
```

Parallel Solution

$$u_{i,j}^{(n+1)} = f(u_{i,j}^{(n)}, u_{i\pm1,j\pm1}^{(n)})$$

- Notice that the Jacobi method is explicit, so each grid point can be calculated independently – easy to parallelize!
  - Assuming that the entire memory space is shared
  - Otherwise, need to distribute the memory
- How? *Many* different methods:
  - AVX/SIMD – Vector operations on a CPU
  - OpenMP – Shared memory between CPUs
  - MPI – Distributed memory between CPUs (multiple programs running)
  - **Cuda – Vector operations on GPU (will explore more later!)**
  - MPI+x – Combination of MPI and one of the above. Allows larger problems
- Best method depends on the problem

Simple Jacobi on a GPU

- Similar domain set-up as for serial
- Need to declare GPU device memory, mem copy
- Kernel configuration: based on domain size
- Device `__constant__` memory: Good for domain parameters
- Improvements? Shared memory: next lesson!
- Let's time it

Timing


```c
#include <time.h>
#include <sys/time.h>

int main(void) {
  struct timeval ts, te;
  gettimeofday(&ts, 0);
  // Do something
  gettimeofday(&te, 0);

  double elapsed_time = (te.tv_sec - ts.tv_sec) +
                        (te.tv_usec - ts.tv_usec)*1e-6;
}
```

LBM & GPUs

The basic LBM scheme reads

$$f_p\left(x + c_p, t + 1\right) = f_p\left(x, t\right) + \omega h \left[f_p^{\mathrm{eq}}\left(\rho, \mathbf{u}\right) - f_p\right]\left(x, t\right)$$

and it does not look too different from a matrix problem.

In fact, let's consider a rectangular mesh. Then,

$$f_p^{(n+1)}\left[i', j'\right] = f_p^{(n)}\left[i, j\right] + \omega h \left[f_p^{\mathrm{eq}}\left[i, j\right] - f_p\left[i, j\right]\right]$$

where i' and j' are destination (post-streaming) nodes.

Beyond similarity, what really matters is to take into account *data locality* that is critical for efficient GPU computing.