AC290: Extreme Computing
Intro to GPU Computing

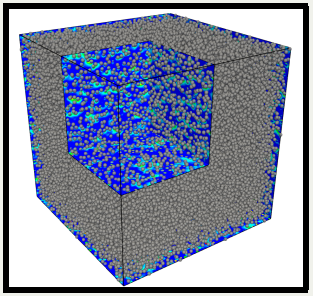
Daniel Willen

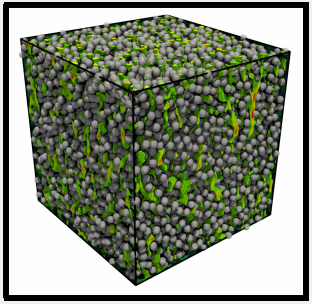
The MITRE Corporation

Lecture 1 Recap

- Compiling and running GPU code on a cluster
- Cuda kernel configuration and thread management
- Memory management

Any questions? (save HW-specific questions for the end)





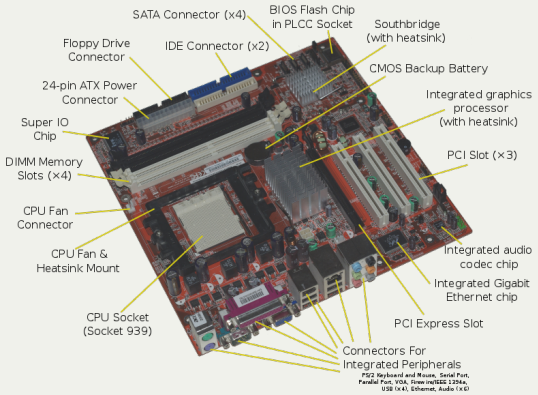
Overview

1. GPU Architecture
2. GPU Memory Types
3. Cuda-aware MPI
4. Homework Help

Objectives:

1. Become familiar with GPU architecture terms
2. Become familiar with what intermediate and advanced Cuda constructs exist

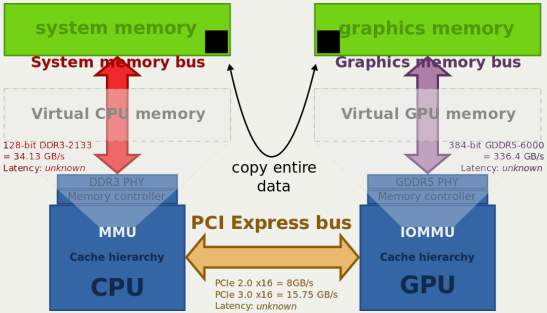
GPUs and Your Computer



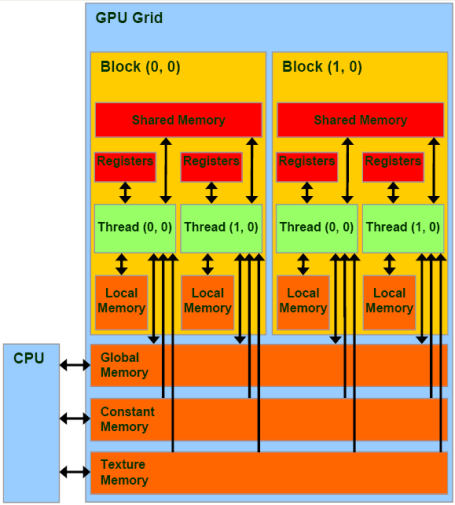
- Motherboard connects most of the components in a computer
  - CPU, RAM, GPU, sound card, ...
- CPU and GPU are connected via the Peripheral Component Interface Express bus
  - PCI Express or PCIe
  - Moving data from one to the other requires communication!

PCIe Bus

- Processors are fast:
  - NVIDIA V100: 900 GB/s
  - Intel Cascade Lake: 128 GB/s
- CPU-GPU communication is slow:
  - PCIe Gen3: 32 GB/s
  - Infiniband HDR: 50 GB/s
  - NVLink: 300 GB/s
- Communication is a bottleneck!
  - Avoid it (algorithmic, GPU-centric)
  - Hide it (async)



Memory Layout



Several different types of GPU memory...

- Red – on-chip
- Orange – off-chip
- `cudaMemcpy` copies to global memory
- Other types need to be set explicitly

http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

Nvidia V100 Architecture

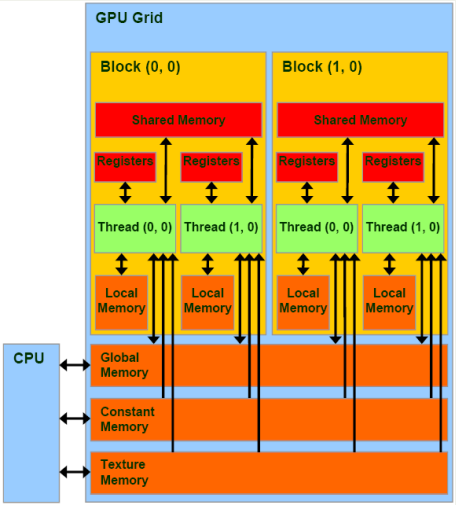
- Streaming Multiprocessor (SM): 64 FP32 cores, 64 INT32 cores, 32 FP64 cores

https://devblogs.nvidia.com/inside-volta/

GPU Architecture

- Recall:
  - Threads are organized into thread blocks
  - Thread blocks are organized into a grid
- Compiler allocates thread blocks to SMs (32 FP64 cores/SM)
- Thread blocks are allocated to 'warps' of threads (32/warp)
- There are hardware limits on...
  1. Threads per warp (32) (threads can be executed in half-warps)
  2. Threads per block
  3. Active blocks
  4. Dimension of thread blocks
  5. ... and many more
- This is important for optimizations
- Further reading: see References

Memory Layout



1. Register: On-chip, per-thread
2. Shared: On-chip, all threads in block
3. Constant: Off-chip, always available, read-only
4. Texture: Read-only, interpolation!
5. Local: Off-chip, per-thread
6. Global: Off-chip, always available, default

http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

Memory Types

- Access latency (in clock cycles) can high depend on the type of access
  - If all threads access the same memory address, the value can be broadcast
  - If not, accesses are serialized
- Penalty – roughly how much longer an access takes compared to register

Type	Location	Lifetime	Penalty	Latency (cycles)	Size
Register (L1)	On-chip	Thread	1×	28	up to 128KB/SM
Shared	On-chip	Block	1×	20-80	up to 96KB/SM
Constant	Off-chip	App	1×	10-300	2KB
Local (L2)	Off-chip	Thread	100×	193	6124KB
Global	Off-chip	App	100×	$\sim$ 235	$\sim$ 16-32GB

https://arxiv.org/pdf/1804.06826.pdf

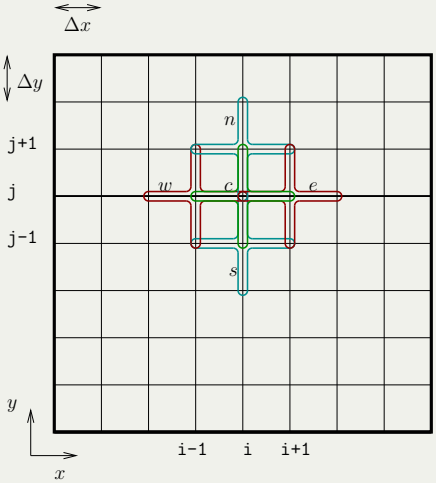
Device Constant Memory


```
/* Set up device constant memory */
int var = 1;            // Host
__constant__ int _var; // Device Constant
cudaMemcpyToSymbol(_var, &var, sizeof(int));

/* Use in a kernel */
__global__ void k1(void) {

  int twice_var = 2*_var;

}
```

- `cudaMemcpyToSymbol(const void* symbol, const void* src, ...)`
- '`symbol` is a device symbol address and is a variable that resides in global or constant memory space'

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1g9bcf02b53644eee2bef9983d807084c7

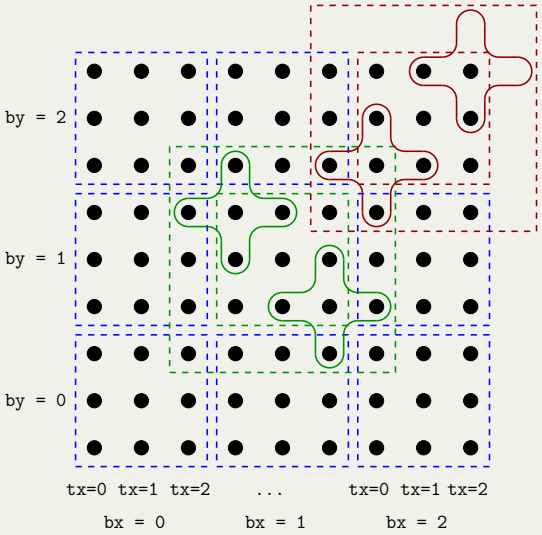
Shared Memory



Finite-difference stencils reuse grid points:

- Central difference: 5× in 2-D, 7× in 3-D
- Each thread reads the same value from global memory to register. This is slow!
- Solution: Store the value in shared memory that can be accessed by all threads in a block
  - Read once, use many times
  - Should not write back to shared memory – write elsewhere!

Shared Memory



- Each thread block needs data from one more thread on each size
- Thus, each thread block size is 2 greater in each dimension than the number of points to compute
- We need to take care not to calculate the value at the edge of the stencil!

Shared Memory Example


```
// Set up kernel execution. Assume we're using the max number of threads in
//  each dimension
int threads_x = MAX_THREADS_DIM;
int threads_y = MAX_THREADS_DIM;

// Each block has 2 extra threads in our example
int blocks_x = ceil(nx / (threads_x - 2)); // Should cast division to double!
int blocks_y = ceil(ny / (threads_y - 2)); // Should cast division to double!

dim3 num_blocks(blocks_x, blocks_y);
dim3 dim_blocks(threads_x, thread_y);

// Launch the kernel as usual
my_kernel<<<num_blocks, dim_blocks>>>(_u, nx, ny);
```

Shared Memory Example


```
my_kernel(double *u, int nx, int ny) {
  // Declare shared memory
  __shared__ double s_u[MAX_THREADS_DIM*MAX_THREADS_DIM];

  // Shared memory index in shared memory grid
  int si = threadIdx.x;  int sj = threadIdx.y;
  int sc = si + sj*blockDim.x;

  // Global memory index in regular grid
  // Subtraction accounts for shared memory overlap
  int ti = blockIdx.x*blockDim.x + threadIdx.x - 2*blockIdx.x;
  int tj = blockIdx.y*blockDim.y + threadIdx.y - 2*blockIdx.y;
  int cc = ti + nx*tj;

  // Load shared memory
  if (ti < nx && tj < ny) { // Only load data in computational grid
    s_u[sc] = u[cc];        // Shared memory <- global memory
  }
```

Shared Memory Example


```
  // Ensure initial load is done
  __syncthreads();

  // Perform Jacobi; avoid shared memory 'shell'
  if ((ti >= 1 && ti < (nx - 1)) &&     // Inside global domain (x)
      (tj >= 1 && tj < (ny - 1)) &&     // Inside global domain (y)
      (si > 0 && si < (blockDim.x - 1)) && // Off shared memory shell (x)
      (sj > 0 && sj < (blockDim.y - 1))) { // Off shared memory shell (y)

    // Center: u[i  + j*nx] = s_u[si + sj*blockDim.x]
    // East:   u[i+1 + j*nx] = s_u[si+1 + sj*blockDim.x]
    // North:  u[i + (j+1)*nx] = s_u[si + (sj+1)*blockDim.x]
  }
} // end of kernel
```

Harder, Better, Faster, Stronger

- Most HPC applications want to run larger simulations
  - Weak scaling: domain size scales with number of processors
  - Computational work per processor is constant
- Or, run faster
  - Strong scaling: Domain size is constant
  - Computational work per processor decreases
- High-end GPUs only have 32GB of device memory
  - Fluids have four fields: $u,\ v,\ w,\ p$
  - `sizeof(double) = 8` bytes
  - Total domain memory $= 32\text{GB} = (4 \text{ fields}) \left(\frac{(8n_x)^3 \text{bytes}}{\text{field}} \right)$
  - Domain size: $\rightarrow n_x \approx 250$: This is small!
    - Current state-of-the-art is $10,000^3$

Communication Breakdown

- In order to run larger simulations, we need to include more GPUs
- This results in memory distributed among the GPUs
- This requires communication for:
  - Boundary condition updates at new time step
  - Moving objects from one subdomain to another (e.g., particles in a fluid)
  - Global reductions (scalar products, min and max, sums, ...)
    - Most systems of linear equations are solved iteratively
    - Several reductions per iteration!
- Recall that communication to/from the GPU is $O(10)$ slower than the bandwidth!

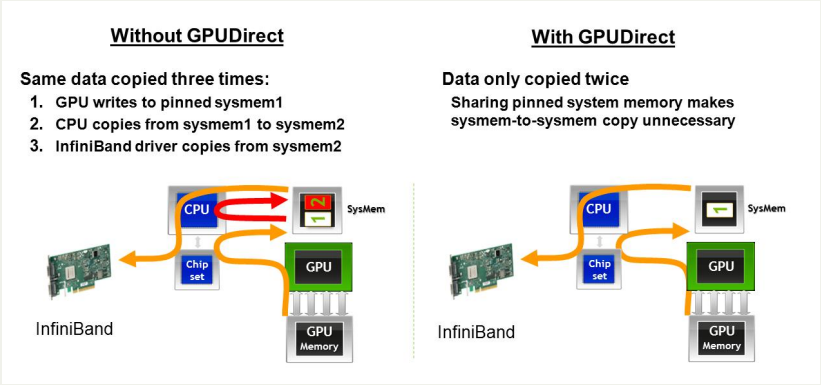
Cuda-aware MPI

- MPI defines standard by which programs running in distributed memory spaces can communicate
- Originally written for CPU-CPU communication.
- Old GPU-GPU communication:
  1. Copy memory from GPU to CPU
  2. MPI: Copy memory from CPU to CPU (inter- or intra-compute node)
  3. Copy memory from CPU to GPU
- Recent (2013) advances allow MPI to recognize GPU memory spaces
  - Abstracts and optimizes memory copying
  - Allows for transparent use of communication acceleration technologies
- Nvidia is very aware that inter-GPU communication is an issue
  - GPUDirect
  - NVLink and NVSwitch
  - Many-GPU implementation is *not* necessarily inefficient!

GPUDirect

GPUDirect (2010)

- Inter-node
- `cudaMallocHost` uses this

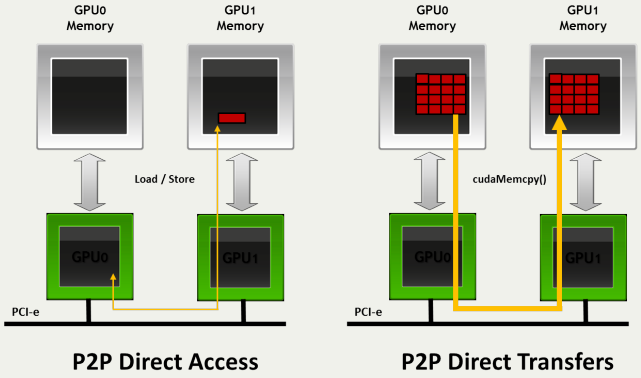


https://developer.nvidia.com/gpudirect

GPUDirect

GPUDirect Peer-to-Peer (P2P) (2011)

- Intra-node
- `cudaMemcpy(..., ..., ..., cudaMemcpyDeviceToDevice)`

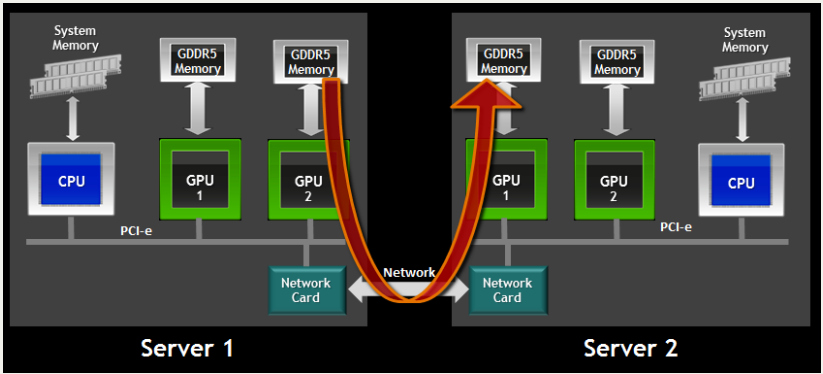


https://developer.nvidia.com/gpudirect

GPUDirect

GPUDirect Remote-Direct Memory Access (RDMA) (2012)

- Inter-node
- Exposed via MPI



https://developer.nvidia.com/gpudirect

Cuda-Aware MPI Example


```
// Init MPI
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

// Init data
int local = rank; // Send this
int remote = -1;  // Recv here

int *_local, *_remote;
cudaMalloc(&_local, sizeof(int));
cudaMalloc(&_remote, sizeof(int));
// Copy host->device
cudaMemcpy(_local, local, sizeof(int),...);
cudaMemcpy(_remote,remote,sizeof(int),...);
```

- Set up MPI as usual
- Local data set to rank number
- Remote data set to `-1`
- Allocate and copy memory to GPU

Cuda-Aware MPI Example


```
MPI_win win;
MPI_Win_create(_remote, 1*sizeof(int),
            sizeof(int), MPI_INFO_NULL,
            MPI_COMM_WORLD, &win);

MPI_Win_fence(0, win);

int target = (rank + 1) % nprocs;
MPI_Put(_local, 1, MPI_INT,
       target, 0, 1, MPI_INT, win);

MPI_Win_fence(0, win);

cudaMemcpy(local, _local, sizeof(int));
cudaMemcpy(remote,_remote,sizeof(int));
```

- `MPI_win` is one-sided
  - Does not require `MPI_Send` or `MPI_Recv`
  - Processes put data directly in remote processes' memory (RDMA)
- Need to declare which memory is accessible
- Need to declare when it is accessible
- Put my info in the target's window
- Use GPU memory exactly like I would CPU
  - Note that `_remote` and `_local` are passed to the MPI calls!

LBM & GPUs

The basic LBM scheme reads

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

and does not look too different from a matrix problem.

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

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

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.

References

1. https://devblogs.nvidia.com/inside-volta/
2. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model
3. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html
4. http://www.ce.jhu.edu/dalrymple/classes/602/Class13.pdf
5. https://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture5.pdf
6. https://www.anandtech.com/show/3809/nvidias-geforce-gtx-460-the-200-king/2
7. https://devblogs.nvidia.com
8. https://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf
9. https://stackoverflow.com/questions/3606636/cuda-model-what-is-warp-size
10. https://stackoverflow.com/questions/10460742/how-do-cuda-blocks-warps-threads-map-onto-cuda-cores
11. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#occupancy
12. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking