Hybrid Programming in CUDA, OpenMP and MPI

James E. McClure

Advanced Research Computing

22 October 2012

Course Contents

This is a talk about concurrency:

- Concurrency within individual GPU
- Concurrency within multiple GPU
- Concurrency between GPU and CPU
- Concurrency using shared memory CPU
- Concurrency across many nodes in distributed memory

Course Contents

Three programming models for achieving concurrency:

- CUDA: Single Instruction Multiple Data (SIMD) programming for GPU
- OpenMP: Fork-and-join parallelism for shared memory programming
- MPI: message passing interface for distributed memory programs

Useful links

ARC website:
http://www.arc.vt.edu/

CUDA Programming Guide:
http://docs.nvidia.com/cuda/index.html

CUDA SDK code examples:
http://docs.nvidia.com/cuda/cuda-samples/index.html

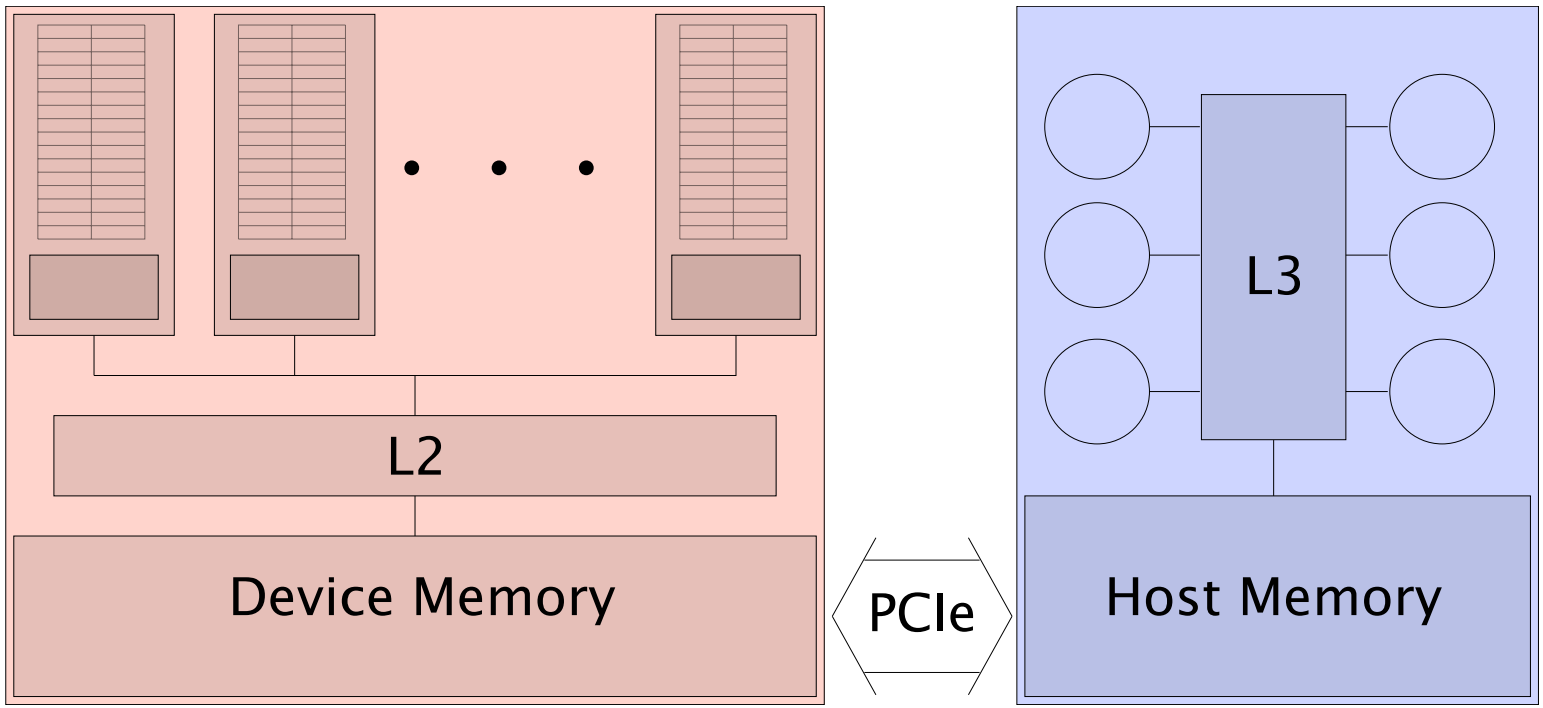
OpenMP website:
http://www.openmp.org

LLNL MPI tutorial:
https://computing.llnl.gov/tutorials/mpi/

Hardware Overview

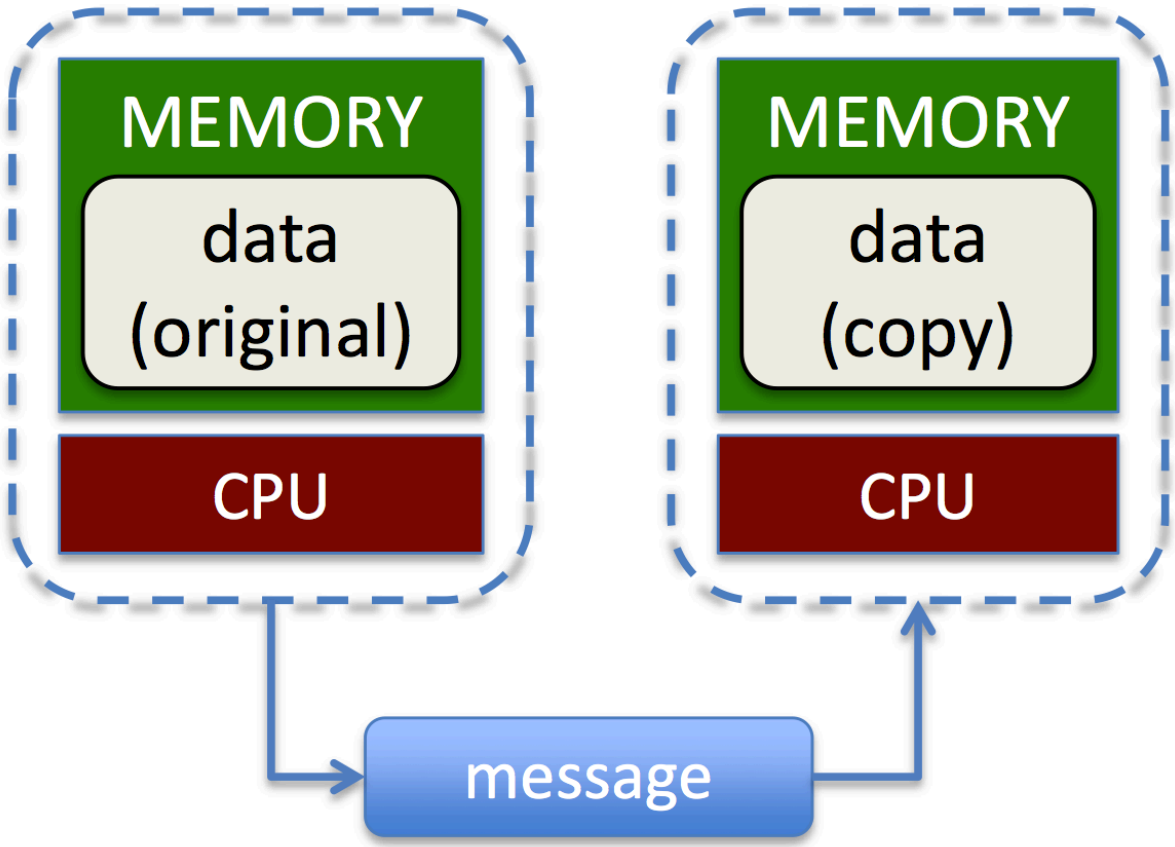
Modern supercomputing nodes are heterogeneous:

- Multiple CPU cores that share memory
- Multiple GPU or other accelerators



Hardware Overview

- GPU and shared memory CPU cores will be programmed with CUDA and OpenMP
- MPI will be used to pass messages between nodes



Examples to Download

ARC HokieSpeed Examples:

`www.arc.vt.edu/resources/hpc/hokiespeed.php`

- `wget <copy link>`
- Simple matrix-matrix multiplication example code
- Example code for OpenMP and MPI
- Example code for CUDA and MPI
- Makefiles for example cases
- Example submission script for HokieSpeed

Compiling with CUDA

To view with the modules you have loaded:

- `module list`

To see a list of available modules:

- `module avail`

Load the compiler by typing:

- `module swap intel gcc`
- `module load cuda`

Compiling with CUDA

The CUDA compiler `nvcc` compiles by:

- identifying device (ie. gpu) functions and compiling them
- passing host (ie. cpu) functions to gcc/g++

To compile, type:

- `nvcc -o runme program.cu`

To compile with *double precision* support, type:

- `nvcc -o runme -arch sm_13 program.cu`

Compiling CUDA with OpenMP and/or MPI

Compiling with OpenMP:

- `nvcc -Xcompiler -fopenmp -lcuda -lcudart -lgomp -o runme program.cu`

Compiling with MPI:

- Identify the path of the MPI library and include directories
- `module show openmpi`
- `-I$(CUDA_INC) -I$(OMPI_INC)`
- `-L$(OMP_LIB) -lmpi -L$(CUDA_LIB64) -lcuda -lcudart`

Running on HokieSpeed

Use the example runscript to submit a job to the queue. Request 4 nodes and 1 mpi process for each node:

- `#PBS -l nodes=4:ppn=12`
- `mpiexec -npernode 1 ./run-cuda-mpi`

Submit the job to the queue:

- `qsub hokiespeed_qsub_example.sh`

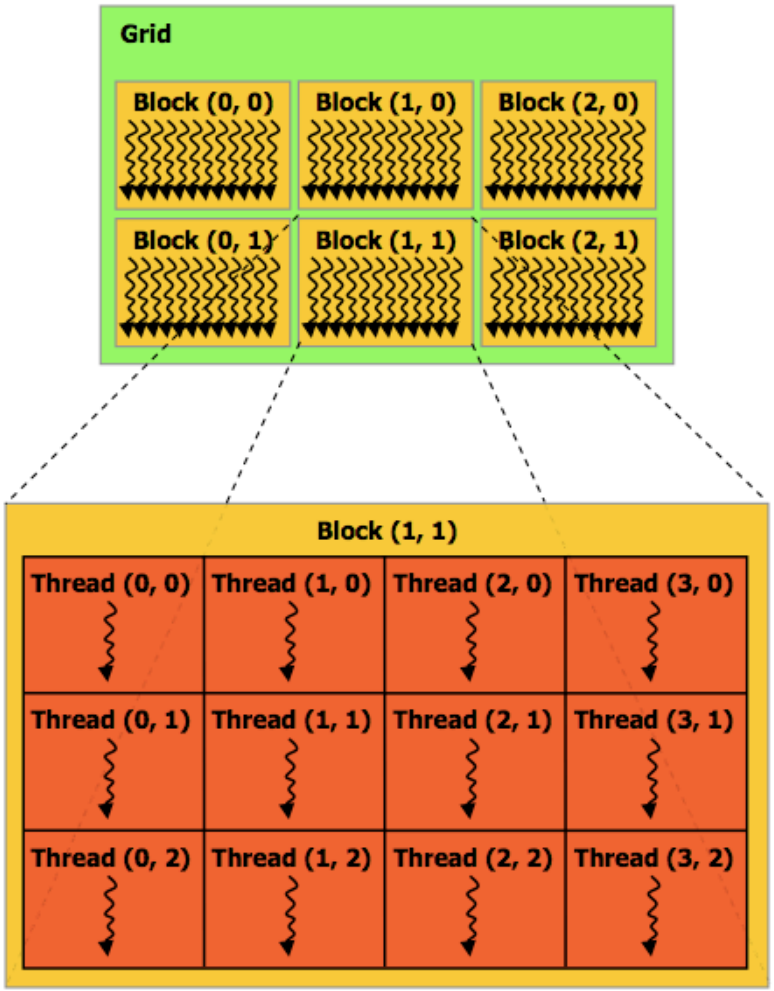
Monitor the job:

- `qstat 1234` (monitor job with id 1234)
- `qdel 1234` (kill job with id 1234)

View the output:

- `more hokiespeed_qsub_example.sh.o1234`

The CUDA Programming Model



Memory Management in CUDA

Example

```
// Multiplication for an NxN matrix
int N = K*BLOCK_SIZE;
// amount of memory in bytes
int size = N*N*sizeof(float);
float *hA,*hB,*hC; //host (cpu)
float *dA,*dB,*dC; //device (gpu)
hA = new float [N*N];
cudaMalloc(&dA,size);
// .... Initialize arrays on the host
cudaMemcpy(dA,hA,size,
           cudaMemcpyHostToDevice);
```

Thread Hierarchy

Example

```
// Set up threadblocks
dim3 threadBlock(BLOCK_SIZE,BLOCK_SIZE);
dim3 numBlocks(K,K);
//Launch the matrix multiplication kernel
gpuMM<<<grid,threadBlock>>>(dA,dB,dC,N);
// . . .
// Copy the result back to the CPU
cudaMemcpy(C,dC,size,
           cudaMemcpyDeviceToHost);
```

CUDA kernels

Parts of your program that run on GPU must be provided as CUDA kernels:

Example

```
__global__ void gpuMM(float*A,float*B,...)
{
 int row,col;
 row = blockIdx.x*blockDim.x+threadIdx.x;
 col = blockIdx.y*blockDim.y+threadIdx.y;
 float sum = 0.f;
 for (int n=0; n<N; ++n)
     sum += A[row*N+n]*B[n*N+col];
 C[row*N+col] = sum;
}
```

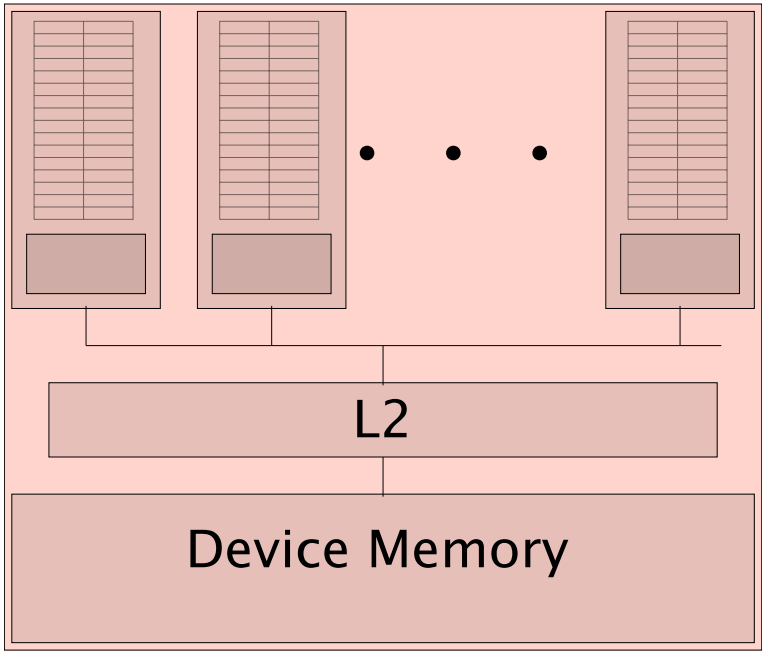
Measuring GPU performance

Example

```
float time;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,stream);
/* Do some GPU work and time it */
cudaEventRecord(stop,stream);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time,start,stop);
```

Occupancy Considerations for GPU

- Fermi GPU can have up to 48 active warps per SM
- Instructions are issued per warp
- If a warp is not ready, the hardware switches warps (context switching)
- Shared memory can limit occupancy!
- Goal: always have enough active warps to saturate the memory bandwidth of the device



Increasing Occupancy with Multiple Kernels

Suppose you are going to perform multiple matrix-matrix multiplications

Example

```
gpuMM<<<grid,nthreads>>>(dA1,dB1,dC1,N);
gpuMM<<<grid,nthreads>>>(dA2,dB2,dC2,N);
```

However, kernels launched from the same stream (in this case the *default* stream) will execute serially.

Kernels launched from different streams can execute concurrently on a single device, if there is room.

CUDA Streams and Events

The CUDA driver API provides streams and events as a way to manage GPU synchronization:

- Synchronization is implied for events within a stream (including default stream)
- Streams belong to a particular GPU
- More than one stream can be associated with a GPU
- Streams are required if you want to perform asynchronous communication
- Streams are critical if you want concurrency with *multiple* GPU or multiple kernels on any single GPU.

CUDA Streams

Example

```
// Create a pair of streams
cudaStream_t stream[2];
for (int i=0; i<2; ++i)
     cudaStreamCreate(&stream[i]);
// Launch a Kernel from each stream
KernelOne<<<100,512,0,stream[0]>>>(..)
KernelTwo<<<100,512,0,stream[1]>>>(..)
// Destroy the streams
for (int i=0; i<2; ++i)
   cudaStreamDestroy(stream[i]);
```

Synchronization of Streams and Events

Streams can be synchronized explicitly:

- `cudaDeviceSynchronize()`: wait for all preceding commands in all streams for a device to complete.
- `cudaStreamSynchronize()`: wait for all preceding events in a specified stream to complete
- `cudaStreamWaitEvent()`: synchronize a stream with an event (both must be specified)
- `cudaStreamQuery()`: Ask the system if preceding commands in a stream have completed

CUDA Streams

Two streams will be synchronized implicitly if any of the following operations are issued in between them

- a page-locked memory allocation (using `cudaMallocHost`)
- a device memory allocation (using `cudaMalloc`)
- a memory copy between two devices
- any CUDA command to the default stream

Using Multiple GPU

Example

```
int deviceCount;
// How many devices?
cudaGetDeviceCount(&deviceCount);
//Get the properties of all devices
for (int dvc = 0; dvc < deviceCount; ++dvc)
  cudaDeviceProp dvcProp;
 cudaGetDeviceProperties(&dvcProp, dvc);
}
```

Using Multiple GPU

GPU can be controlled by:

- a single CPU thread
- multiple CPU threads belonging to the same process
- multiple CPU threads belonging to different processes

All CUDA calls are issued to the *current* GPU:

Example

```
cudaSetDevice(0);
gpuMM<<<grid,threadBlock>>>(dA1,dB1,dC1,N);
cudaSetDevice(1);
gpuMM<<<grid,threadBlock>>>(dA2,dB2,dC2,N);
```

Streams and Multiple GPU

CUDA streams belong to a device:

- Each device has its own default stream
- Streams belong to the GPU that was current when it was created
- You cannot issue calls to a stream if the associated GPU is not active

Example

```
cudaSetDevice(0);
cudaStreamCreate(&streamA);
cudaSetDevice(1);
cudaStreamCreate(&streamB);
// Launch kernels
gpuMM<<...,streamA>>(dA1,dB1,dC1,N);
gpuMM<<...,streamB>>(dA2,dB2,dC2,N);
```

Peer to Peer Memory Copies

Suppose you want to use multiple GPU to work together and solve *the same problem*

You'll probably need to transfer some data between GPU to do this:

Example

```
cudaMemcpyPeerAsync(void *dst, int dst_dev,
        void *src,int src_dev,size_t size,
        cudaStream_t stream)
```

- Copies data between two devices
- `stream` must belong to *source* gpu
- blocking version exists too!

Exercises: GPU Concurrency

- Implement a GPU timer using streams
- Study the performance of `gpuMM` as a function of the matrix size `N`
- Increase the occupancy for small matrices by launching multiple kernels simultaneously
- Perform simultaneous matrix-matrix multiplication using two GPU
- How big do the matrices have to be for using multiple GPU to provide a significant advantage?

Concurrency using CPU and GPU

Kernel launches are *asynchronous* with respect to the CPU, even within the default stream

Example

```
gpuMM<<<grid,threadBlocks>>>(dA,dB,dC,N);
for (row=0; row<N; row++){
  for (col=0; col<N; col++){
      sum = 0.f;
      for (n=0; n<N; n++){
          sum+= hA[row*N+n]*hB[n*N+col];
      }
      hC[row*N+col] = sum;
   } // matrix multiplication on the CPU
}
cudaDeviceSynchronize();
```

Concurrency using CPU and GPU

We don't just pay for the computation, we also pay for data CPU $\leftrightarrow$ GPU transfers

Example

```
cudaMemcpy(dA,hA,size,
           cudaMemcpyHostToDevice);
cudaMemcpy(dB,hB,size,
           cudaMemcpyHostToDevice);
gpuMM<<<grid,threadBlock>>>(dA,dB,dC,N);
cudaMemcpy(C,dC,size,
           cudaMemcpyDeviceToHost);
```

Asynchronous Memory Transfers in CUDA

- We know that CPU $\leftrightarrow$GPU memory transfers are expensive
- We also know that PCIe can perform simultaneous, bi-directional transfers:
  - One cudaMemcpy for Host $\rightarrow$ Device
  - One cudaMemcpy for Device $\rightarrow$ Host
- If the memory transfers belong to the same stream they will be synchronized
- We need a way to asynchronously perform transfers to get the full advantage of PCIe

Asynchronous Memory Transfers in CUDA

Example

```
// Host data MUST be pinned!!!
cudaMallocHost(&cpuData,size);
cudaMalloc(&gpuData,size);
// Bi-directional memory transfer
cudaMemcpyAsync(gpuData,cpuData,size,
        cudaMemcpyHostToDevice,stream[0]);
cudaMemcpyAsync(cpuData,gpuData,size,
        cudaMemcpyDeviceToHost,stream[1]);
// Clean up
```

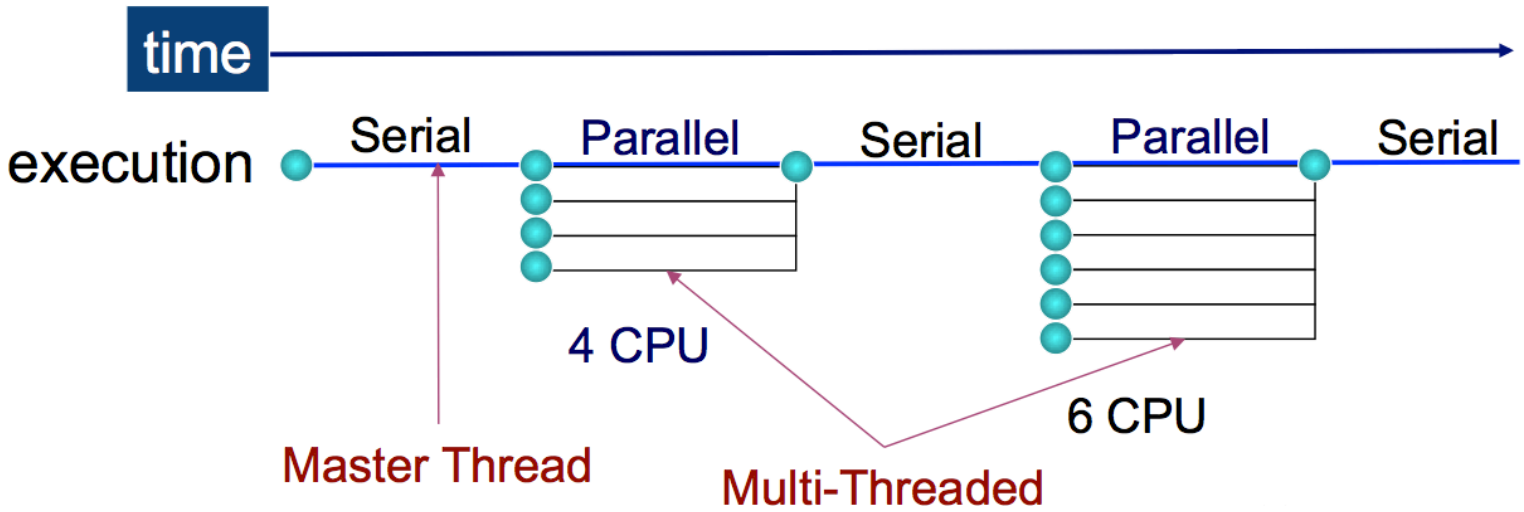
Exercises: CPU+GPU Concurrency

- How does simultaneously executing CPU and GPU matrix-multiplication effect performance?
- How does the performance of each depend on the problem size?
- How does the performance change if you include memory transfers in the timings?
- Is it possible to overlap memory copies for multiple GPU kernels?

Adding Multiple CPU Cores Using OpenMP

OpenMP uses a fork-and-join model of parallelism to target multi-core CPU

- Master thread initiated at run-time and persists throughout
- Worker threads are created within parallel regions



Adding Multiple CPU Cores Using OpenMP

Example

```
#include <omp.h>
int main (void){
  int omp_threads = 12;
  omp_set_num_threads(omp_threads);
  double starttime = omp_get_wtime();
  // . . .
#pragma omp parallel
{
// CPU & GPU work within a parallel region
}
  cudaDeviceSynchronize();
  double stoptime = omp_get_wtime();
  double CPU_time = stoptime-starttime;
}
```

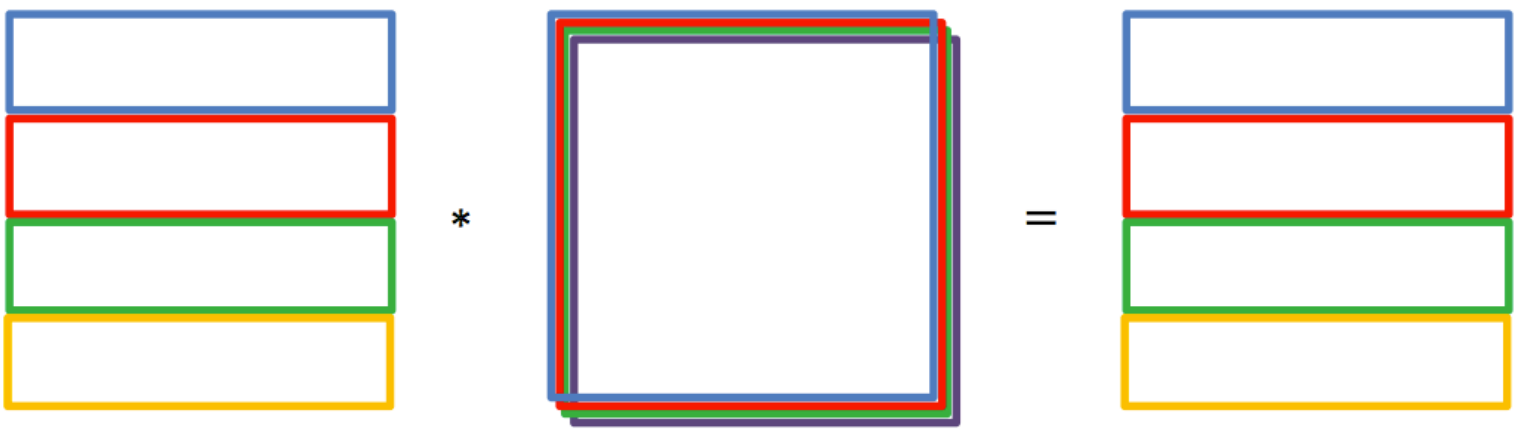
Adding Multiple CPU Cores Using OpenMP

Example

```
#pragma omp parallel
{ // Work inside a parallel region
#pragma omp master
 { // Manage GPU from master thread
  cudaMemcpy(...);
  gpuMM<<<grid,threadBlocks>>>(dA,dB,dC,N);
 }
 // Use all threads for CPU work
}
```

Adding Multiple CPU Cores Using OpenMP

- Need to break up the work between threads
- Partition by rows:



Adding Multiple CPU Cores Using OpenMP

Example

```
int partRow = N/omp_threads;
#pragma omp parallel
{ // work within a parallel region
// . . . GPU calls from master thread
int row,col,n;// iterators for each thread
#pragma omp for schedule(guided,partRow)
  for (row=0; row<N; row++){
    // . . . matrix multiply on CPU
  }
}
```

Exercises: OpenMP and CUDA

- Implement the matrix-matrix multiplication using OpenMP
- Show that the OpenMP implementation scales (ie. set `omp_threads=1,2,...12` and measure wall time)
- Does parallel initialization of `hA` effect the performance?
- What happens if you make CUDA calls from worker threads?

Using CUDA with MPI

Example

```
#include <mpi.h>
#include "cuda.h"
int main(int argc, char**argv){
 int np,rank;
 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&np);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 // . . .
 // Each MPI process assings work to CPU
 // and GPU using CUDA and/or OpenMP
 // . . .
 MPI_Finalize();
}
```

Using CUDA with MPI

Whether or not MPI calls can reference GPU memory depends on CUDA version and hardware compute capability:

Example

```
//pointers to host memory work always
float *buf;
buf = new float[N];
MPI_Send(&A,N,MPI_FLOAT,recvID,tag,
         MPI_COMM_WORLD);
//pointers to device memory only work
//with newest hardware/ CUDA
size = N*sizeof(float);
cudaMalloc(buf,size);
MPI_Send(&buf,N,MPI_FLOAT,recvID,tag,
         MPI_COMM_WORLD);
```

Exercises: MPI and CUDA

Compare the performance of the following for bi-directional communication between two nodes:

1. Use `MPI_Sendrecv` to send data from a source process to a destination process
2. The following sequence:
   - Copy data to be sent from the device to the host at the source process
   - Use `MPI_Sendrecv` to send data from the source process to a destination process
   - Copy received data from host to device at destination process

Modify the example code available from:
http://mpi.deino.net/mpi_functions/MPI_Sendrecv.html

Questions?

Be sure to fill out the FDI evaluation forms:
http://www.fdi.vt.edu/training/evals/index.html