CUDA, OpenMPI, OpenMP Basics

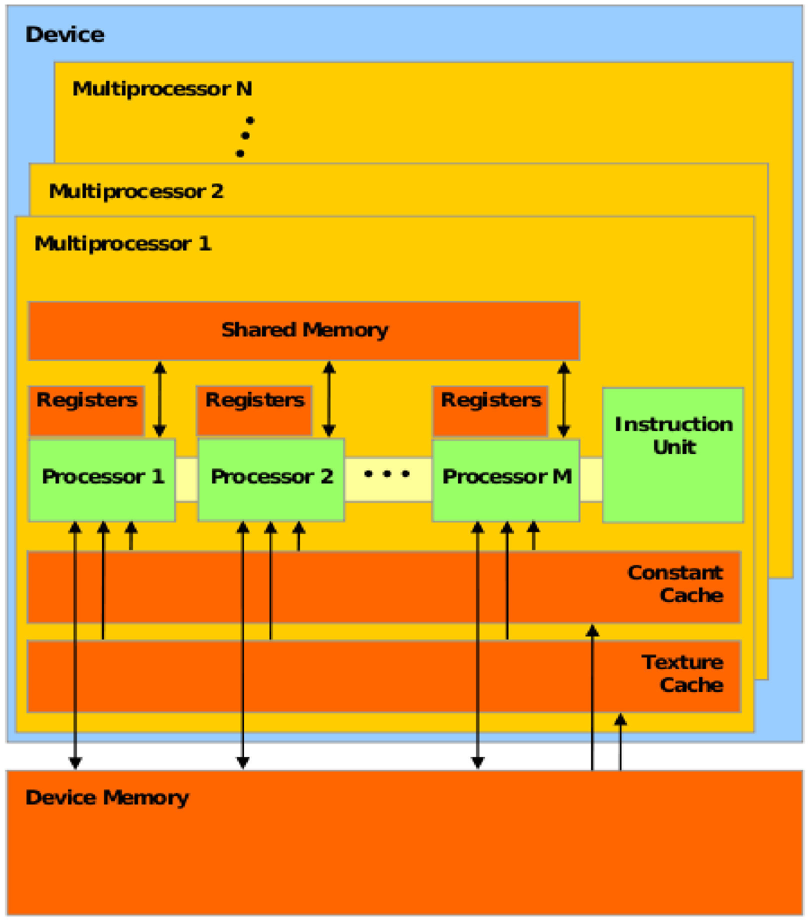
Matt Heavner
9/15/2009

CUDA: Where does it fit into a problem?

- A “SIMD” architecture
- Works well when a similar operation is applied to a large dataset
  - Can also branch off, though, so not strictly SIMD
- Provides a small amount of additional syntax to C or C++ which allows parallel “kernels” to be run on the device

CUDA Physical Architecture

- Build around SMPs
  - Each S1070 has 4 CUDA devices, each with 30 SMPs
- Each SMP has 8 SP cores
- Each thread mapped to one SP
- Threads managed in groups of 32 – warps (basically, a SIMD group)
- Warp elements free to branch, though device will then serialize



A set of SIMT multiprocessors with on-chip shared memory.

Figure 4-2. Hardware Model

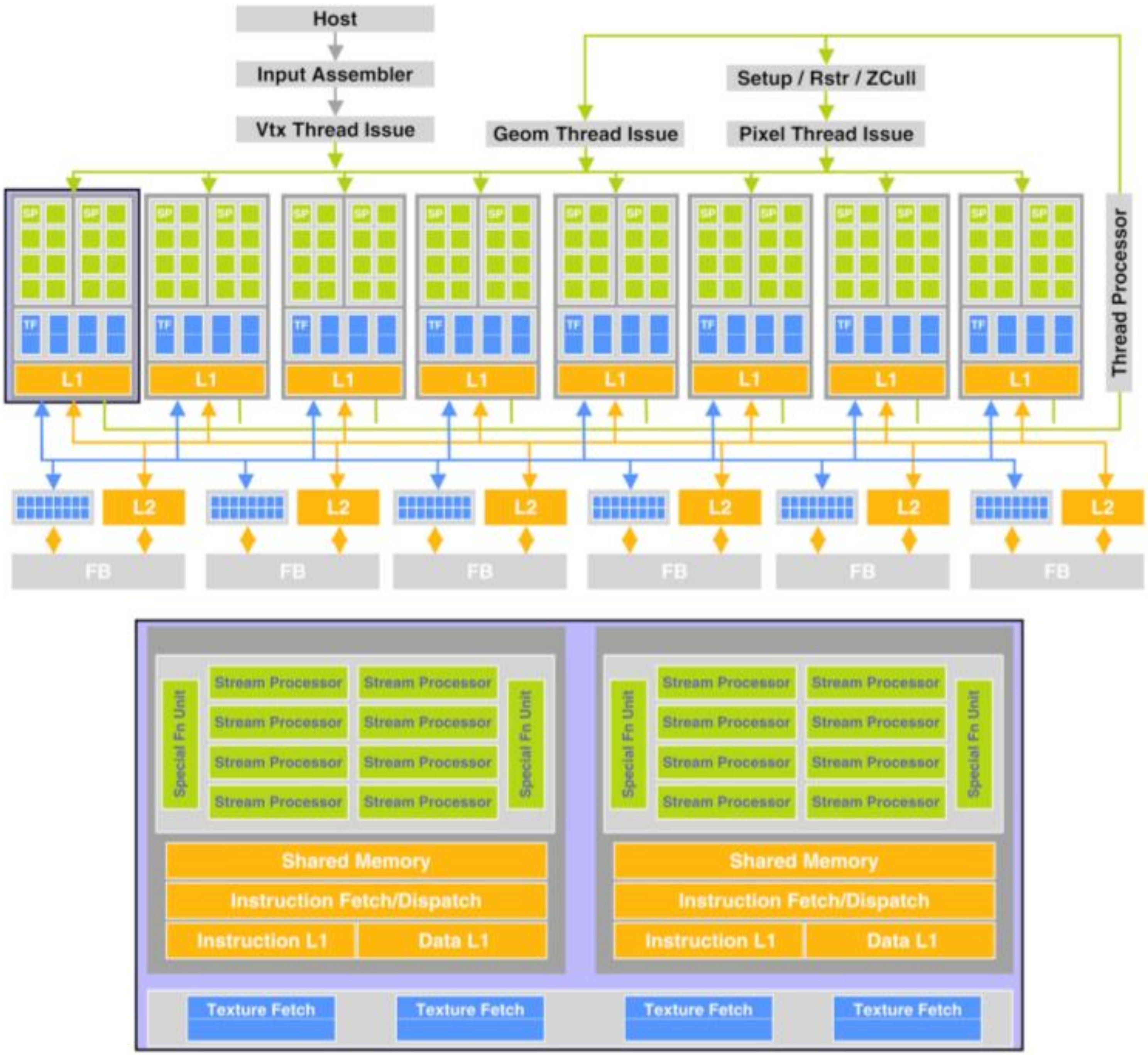


Fig. 1. ***Today, both AMD and NVIDIA build architectures with unified, massively parallel programmable units at their cores. (a) The NVIDIA GeForce 8800 GTX (top) features 16 streaming multiprocessors of 8 thread (stream) processors each. One pair of streaming multiprocessors is shown below; each contains shared instruction and data caches, control logic, a 16 kB shared memory, eight stream processors, and***

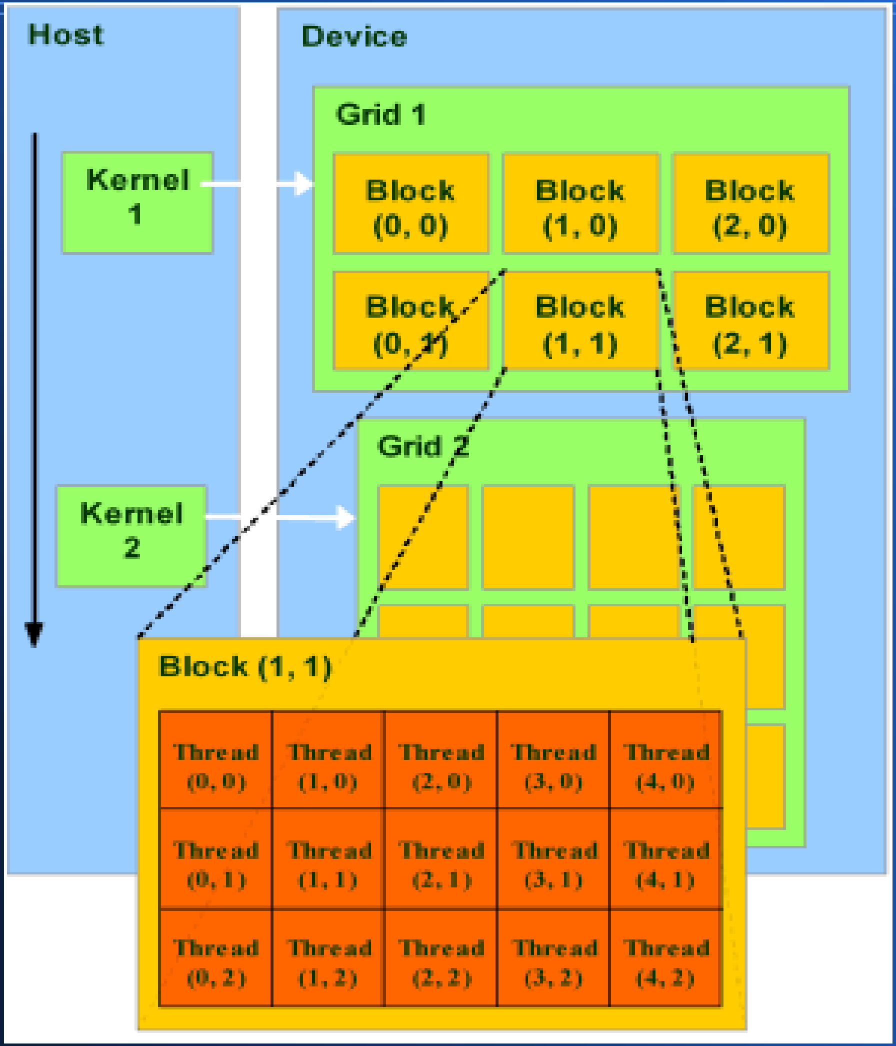
CUDA Compute Capability

- CUDA Products are divided into compute capability 1.0, 1.1, 1.2 and 1.3
- The architecture present on magic has compute capability 1.3
  - 1.2 / 1.3 adds support for double precision floating point ops
  - Max active warps / multiprocessor = 32
  - Max active threads / multiprocessor = 1024
  - Most flexibility

CUDA Kernels

- A kernel is the piece of code executed on the CUDA device by a single CUDA thread.
- Each kernel is run in a thread.
- Threads are grouped into warps of 32 threads. Warps are grouped into thread blocks. Thread blocks are grouped into grids.
- Blocks and grids may be 1d, 2d, or 3d
- Each kernel has access to certain variables that define its position – gridDim, blockIdx, blockDim, threadIdx. Useful for a dataset index
- While host code may be C++, this must be C along with CUDA syntax extensions

CUDA Logical Architecture



Kernel Call Syntax

- Kernels are called with the <<<>>> syntax
- <<<Dg, Db, Ns, S>>>
- Where:

  Dg = dimensions of the grid (type dim3)

  Db = dimensions of the block (type dim3)

  Ns = number of bytes shared memory dynamically allocated / block (type size_t). 0 default

  S = associated cudaStream_t. 0 default

Example CUDA Kernel

- Example syntax:
  - Kernel definition:

```
__global__ void kernel(int* dOut, int a, int b){
        dOut[blockDim.x*threadIdx.y + threadIdx.x] =
            a+b;
}
```

  - Kernel call:

```
kernel<<<1,dim3(2,2)>>>(arr,1,2);
```

Function Type Qualifiers

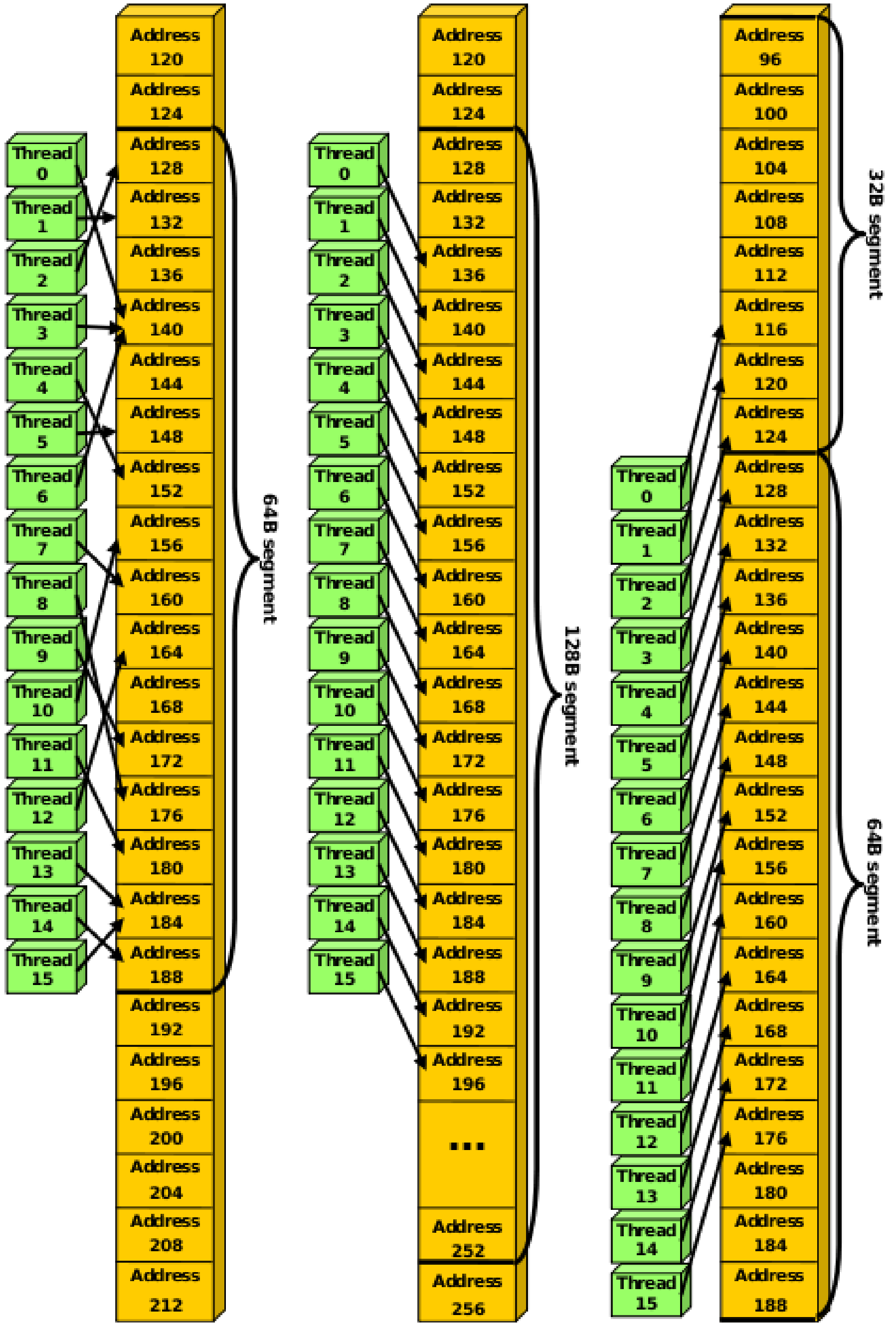
- The kernel was defined as __global__. This specifies that the function runs on the device and is callable from the host only
- __device__ and __host__ are other available qualifiers.
  - __device__ - executed on device, callable only from device
  - __host__ - default if not specified. Executed on host, callable from host only.

CUDA Memory Types

- Access to device memory (slowest, not cached), shared memory (faster) and thread registers (fastest)
- Only device memory is directly-accessible from the host
  - A typical approach is to copy a large dataset to device memory. From there it can be brought into faster shared memory and processed.
  - For all threads of a warp, a shared instruction requires 4 cycles
  - Contrast to 400-600 cycles required for device memory read instruction

Memory Access Coalescing

- Requirements necessary to group parallel memory operations into one call
- Compute capability 1.2+ is the least-strict: allows for any type of memory access pattern to be grouped
- Transaction coalesced as soon as accessed memory lies in the same segment size:
  - 32 bytes if all threads access 8-bit words
  - 64 bytes if all threads access 16-bit words
  - 128 bytes if all threads access 32-bit / 64-bit words



Left: random **`float`** memory access within a 64B segment, resulting in one memory transaction.

Center: misaligned **`float`** memory access, resulting in one transaction.

Right: misaligned **`float`** memory access, resulting in two transactions.

Figure 5-4. Examples of Global Memory Access by Devices with Compute Capability 1.2 and Higher

Syntax for using CUDA Device Memory

- cudaError_t cudaMalloc(void** devPtr, size_t size)
  - Allocates size_t bytes of device memory pointed to by *devPtr
  - Returns cudaSuccess for no error
- cudaError_t cudaMempy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)
  - Dst = destination memory address
  - Src = source memory address
  - Count = bytes to copy
  - Kind = type of transfer
    - cudaMemcpyHostToHost
    - “HostToDevice
    - “DeviceToHost
    - “DeviceToDevice

Syntax for CUDA Device Memory Cont.

- cudaError_t cudaFree(void* devPtr)
  - Frees memory allocated with cudaMalloc

CUDA Shared Memory

- The __shared__ qualifier declares a variable that:
  - Resides in shared memory of a thread block
  - Has lifetime of the block
  - Is only accessible from all threads in the block
- Ex:
  - Declared as: extern __shared__ float shared[];, size specified in kernel call
  - All shared memory uses the same beginning offset, so if one wanted the equivalent of: short array0[128];
  - float array1[64];
  - int array2[256];
  - It would be accessed on the device in shared memory as follows:

```
extern __shared__ char array[];
__device__ void func() // __device__ or __global__
{
  short* array0 = (short*)array;
  float* array1 = (float*)&array0[128];
  int* array2 = (int*)&array1[64];
}
```

Some Extra CUDA Syntax

- #include <cuda_runtime.h>
- cudaSetDevice must be called before executing kernels.
- When finished with the device, cudaThreadFinalize should be called

Compiling CUDA Code

- Done with the nvcc compiler
- nvcc invokes different tools at different stages
- Workflow:
  - Device code is separated from host code
  - Device code compiled into binary (cubin object)
  - Host compiler (gcc) is invoked on host code
  - The two are linked

Compiling CUDA Code Contd.

- On magic, the following flags are needed to compile CUDA code:
  - -I/usr/local/cuda/include
  - -L/usr/local/cuda/lib
  - -lcudart

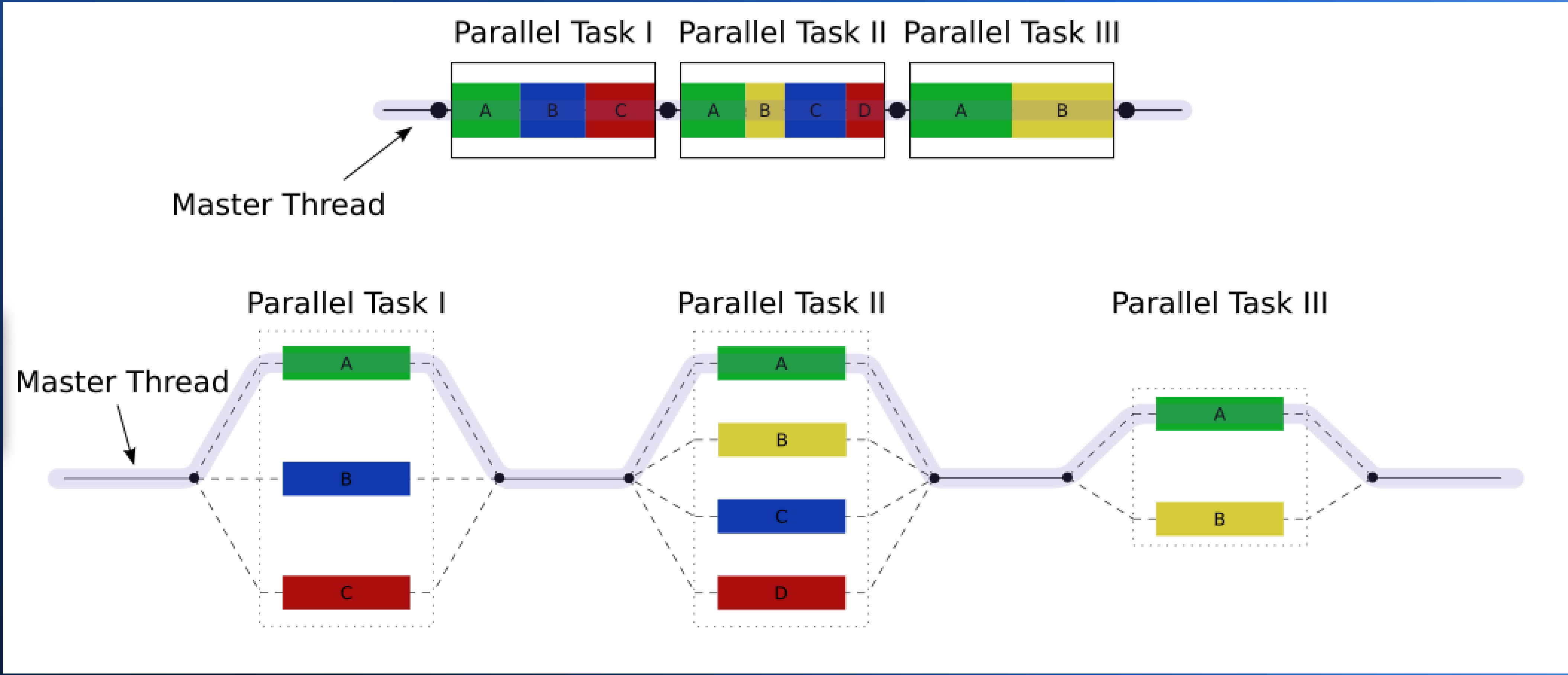
More info about CUDA

- lib, include, bin directories for CUDA are located at /usr/local/cuda on magic
- Much more information available within the CUDA programming guide:
  - http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf
- API reference is also available:
  - http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/CudaReferenceManual_2.1.pdf

OpenMP Overview

- A pthreads alternative
- Used to create multi-threaded shared-memory programs
- Add notation to specify parallel regions of code
  - Uses fork / join model – spawns threads for parallel regions
  - Threads given IDs, root = 0
- Done with #pragma statements

OpenMP Overview



http://en.wikipedia.org/wiki/File:Fork_join.svg

OpenMP Basic Syntax

- #pragma omp …

- # pragma omp parallel for

```
for ( x = 0; x<25; x++)
   printf(“%d\n”,x);
```

- omp_get_thread_num() returns thread number

Example with CUDA

```
bool initDevice() {
   return (cudaSetDevice(omp_get_thread_num()) == cudaSuccess);
}

#pragma omp parallel num_threads(devCount)
if (initDevice())
{
   dim3 dimBlock(32,16);
   dim3 dimGrid(65535,65535);
   kernel<<<dimGrid,dimBlock>>>();
   cudaThreadExit();
}
```

Compiling with OpenMP

- With gcc:
  - #include omp.h
  - Add -fompenmp flag
    - With nvcc, this should be –Xcompiler –fopenmp as this needs to be passed directly to gcc
    - -Xcompiler passes flags directly to host compiler
  - Add -lgomp flag

More OpenMP Information

- More comprehensive introduction from CCR:
  - http://www.ccr.buffalo.edu/download/attachments/65681/Omp-I-handout-2x2.pdf?version=2
- Advanced topics related to OpenMP from CCR:
  - http://www.ccr.buffalo.edu/download/attachments/65681/Omp-II-handout-2x2.pdf?version=2
- Another presentation I found informative:
  - http://pages.cs.wisc.edu/~gibson/filelib/openmp.ppt

MPI Overview

- Message Passing Interface
- Spawn processes across physically-different nodes in a cluster environment
- Groups processes into groups called communicators – default global communicator is MPI_COMM_WORLD

Running an MPI job

- mpirun –mca btl ^openib,udapl –np 9 –hostfile machinefile ./hello
  - Runs MPI-enabled program ‘hello’ on 9 nodes, taken from a list specified in ‘machinefile’
  - -mca btl ^openib,udapl suppresses some errors related to default networking methods

Program Outline Using MPI

Include MPI header files

Declare variables / data structures

Initialize MPI

.

Main program – MPI enabled

.

Terminate MPI

End program

Basic MPI Syntax

int MPI_Init(int* argc, char*** argv**)** – initialize MPI

int MPI_Comm_rank(MPI_Comm comm, int* rank) – get rank of process

int MPI_Comm_size(MPI_Comm comm, int* size) **–** get size of communicator

int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) – send data

int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm) – receive data

int MPI_Finalize() - shut down MPI

Example with OpenMPI, OpenMP, CUDA

```
// A simple "hello world" using CUDA, OpenMP, OpenMPI
// Matt Heavner

using namespace std;

#include <stdio.h>
#include <cuda_runtime.h>
#include <stdlib.h>
#include <omp.h>
#include <mpi.h>
#include "kernel.cu"

char processor_name[MPI_MAX_PROCESSOR_NAME];
bool initDevice();
extern "C" void kernel();

bool initDevice()
{
   printf("Init device %d on %s\n",omp_get_thread_num(),processor_name);
   return (cudaSetDevice(omp_get_thread_num()) == cudaSuccess);
}

int main(int argc, char* argv[])
{
   int numprocs,namelen,rank,devCount;

   int val = 0;
   MPI_Status stat;

// Initialize MPI
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);
  printf("Hello from %d on %s out of
      %d\n",(rank+1),processor_name,numprocs);

  if (cudaGetDeviceCount(&devCount) != cudaSuccess)
  {
     printf("Device error on %s\n!",processor_name);
     MPI_Finalize();
     return 1;
  }
 // Test MPI message passing
  if (rank == 0){
     val = 3;
     for (int i=0; i<numprocs; i++)
        MPI_Send(&val,1,MPI_INT,i,0,MPI_COMM_WORLD);
  }
  MPI_Recv(&val,1,MPI_INT,0,0,MPI_COMM_WORLD,&stat);
  if (val == 3)
     cout << rank << " properly received via MPI!" << endl;
  else
     cout << rank << " had an error receiving over MPI!" << endl;
```

Example with OpenMPI, OpenMP, CUDA cont.

```
  // Run one OpenMP thread per device per MPI node
  #pragma omp parallel num_threads(devCount)
  if (initDevice())
  {
    // Block and grid dimensions
    dim3 dimBlock(12,12);

    kernel<<<1,dimBlock>>>();
    cudaThreadExit();
  }
  else
  {
    printf("Device error on %s\n",processor_name);
  }
  MPI_Finalize();
  return 0;
}
```

Example with OpenMPI, OpenMP, CUDA kernel

```
// kernel.cu
//
// An arbitrary kernel

#ifndef _BURN_KERNEL_H_
#define _BURN_KERNEL_H_

extern "C"
{
   __global__ void kernel()
   {
      __shared__ float shared[512];
      float a = 3.0 * 5.0;
      float b = (a * 50) / 4;
      int pos = threadIdx.y*blockDim.x+threadIdx.x;
      shared[pos] = b;
   }

}

#endif
```

Example with OpenMPI, OpenMP, CUDA Makefile

```
CC=/usr/local/cuda/bin/nvcc
CFLAGS= -I/usr/lib64/openmpi/1.2.7-gcc/include -I/usr/local/cuda/include -Xcompiler -fopenmp
LDFLAGS= -L/usr/lib64/openmpi/1.2.7-gcc/lib -L/usr/local/cuda/lib
LIB= -lgomp -lcudart -lmpi
SOURCES= helloworld.cu
EXECNAME= hello

all:
	$(CC) -o $(EXECNAME) $(SOURCES) $(LIB) $(LDFLAGS) $(CFLAGS)

clean:
	rm *.o *.linkinfo
```

Example with OpenMPI, OpenMP, CUDA Run Command

run.sh

```
# Command to submit "hello world" example

#!/bin/sh

mpirun -mca btl ^openib,udapl -np 9 -hostfile machinefile ./hello
```

machinefile

```
ci-xeon-2
ci-xeon-3
ci-xeon-4
ci-xeon-5
ci-xeon-6
ci-xeon-7
ci-xeon-8
ci-xeon-9
ci-xeon-10
```

More MPI Information

- This was just a barebones example of combining / compiling CUDA, OpenMP, OpenMPI together
- Much more comprehensive tutorials:
  - CCR MPI Quick Reference
    - http://www.ccr.buffalo.edu/download/attachments/65681/mpi-quickref-handout-2x2.pdf?version=1
  - Intermediate MPI
    - http://www.ccr.buffalo.edu/download/attachments/65681/Mpi-intermed-handout-2x2.pdf?version=2
  - Advanced MPI
    - http://www.ccr.buffalo.edu/download/attachments/65681/Mpi-advanced-handout-2x2.pdf?version=2