Parallel Computing Notes
Topic: Notes on Hybrid MPI + GPU Programming

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Last Update: 11/01/17

Table of Contents

Reading List for GPU/CUDA

- Reading: CUDA API: Sanders & Kandrot
  - Intro to CUDA, Ch3
  - Block Parallelism, Ch 4
  - Thread Parallelism, Ch 5
- Tutorials
  - CUDA Tutorial: `https://developer.nvidia.com/cuda-training#1`
  - CUDA API: `http://docs.nvidia.com/cuda/cuda-runtime-api/index.html`
  - CUDA SDK: `https://developer.nvidia.com/gpu-computing-sdk`
  - CUDA example files on tuckoo in /COMP605/cuda
- GPU Architectures:
  - References: NVIDIA online documents
  - and lecture notes by S.Weiss `http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci360/lecture_notes/gpus.pdf`

Reading List for GPU/CUDA Reading

- Matrix Multiplication with CUDA — A basic introduction to the CUDA programming model. Robert Hochberg, August 11, 2012
- K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication," Graphics Hardware (2004)
- http://www.hpcwire.com/hpcwire/2008-10-08/compilers_and_more_programming_gpus_today.html
- http://www.hpcwire.com/hpcwire/2008-10-30/compilers_and_more_optimizing_gpu_kernels.html
- http://www.admin-magazine.com/HPC/Articles/Parallel-Programming-with-OpenMP
- http://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf

GPU/CUDA Env on tuckoo

Check whether your computer has a capable GPU

- Identify the model name of your GPU.
- On Windows, use GPU-Z found here.
  Note: The listed capabilities of the card may be inaccurate on multi GPU systems.
- On linux, in a console use: lspci — grep VGA
- On Macintosh, Select About this Mac from the Apple menu, then click More Info. Under Hardware select Graphics/Displays.
- tuckoo is not a GPU node:

```
[mthomas@tuckoo ~]$  lspci | grep VGA
[mthomas@tuckoo]$ lspci | grep VGA
02:00.0 VGA compatible controller: Matrox Electronics Systems Ltd. MGA G200e [Pilot]
      ServerEngines (SEP1) (rev 02)
```

- Check a GPU node:

```
[mthomas@tuckoo]$ ssh node9 "/sbin/lspci | grep VGA"
01:00.0 VGA compatible controller: NVIDIA Corp.. NV44 [GeForce 6200 LE] (rev a1)
02:00.0 VGA compatible controller: NVIDIA Corp.. GF100 [GeForce GTX 480] (rev a3)
03:00.0 VGA compatible controller: NVIDIA Corp.. GF100 [GeForce GTX 480] (rev a3)
```

Download CUDA SDK for OS X (Fall'12)

- CUDA SDK: https://developer.nvidia.com/gpu-computing-sdk
- Instructions:
  `http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-mac-os-x/index.html`
- Check that system has CUDA GPU system
- MacBookPro link:
  `http://www.geforce.com/hardware/notebook-gpus/geforce-gt-650m`
- Install on OS X:

```
Set ENV variables (csh):
# for CUDA
set path = (/Developer/NVIDIA/CUDA-5.0/bin $path)
set DYLD_LIBRARY_PATH = /Developer/NVIDIA/CUDA-5.0/lib
```

Check for location of the compiler:

```
[gidget:~] mthomas% which nvcc
/Developer/NVIDIA/CUDA-5.0/bin/nvcc

[gidget:~] mthomas% nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2012 NVIDIA Corporation
Built on Fri_Sep_28_16:10:16_PDT_2012
Cuda compilation tools, release 5.0, V0.2.1221
```

See if the CUDA Driver is installed correctly on OS X:

```
[gidget:~] mthomas% kextstat | grep -i cuda
  123    0 0xffffff7f825b9000 0x2000
  0x2000     com.nvidia.CUDA (1.1.0) <4 1>
```

Check for tuckoo GPU nodes:

```
[mthomas@tuckoo cudatests]$ cat /etc/motd
. . .
  GPUs
   ---------------------------------------------
   node9  has 2     GTX 480 gpu cards (1.6GB dev ram ea.)
   node8  has 2     C2075   gpu cards (  6GB dev ram ea.)
   node7  has 2     C1060   gpu cards (  4GB dev ram ea.)
   node11 has 1     K40     gpu card  (                 )

   see files /examples/cuda/nodeX.SPECS (X=7,8,9,11)*
         and /examples/cuda/GPU.SPECS

. . .
```

Check tuckoo compiler and ENV

```
[mthomas@tuckoo ~]$  lspci | grep VGA
02:00.0 VGA compatible controller: Matrox Graphics, Inc. MGA G200e [Pilot] ServerEngines (SEP1) (rev 02)

[mthomas@tuckoo ~]$ which nvcc
/usr/local/cuda/bin/nvcc

[mthomas@tuckoo ~]$ nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2012 NVIDIA Corporation
Built on Thu_Apr__5_00:24:31_PDT_2012
Cuda compilation tools, release 4.2, V0.2.1221

[mthomas@tuckoo ~]$ cat /proc/version
Linux version 2.6.32-220.el6.x86_64
(mockbuild@c6b18n3.bsys.dev.centos.org)
(gcc version 4.4.6 20110731 (Red Hat 4.4.6-3) (GCC) )
#1 SMP Tue Dec 6 19:48:22 GMT 2011
```

Check tuckoo ENV on one of the nodes

```
[mthomas@tuckoo cudatests]$ cat env.bat
#!/bin/sh
# this example batch script requests many processors...
# for more info on requesting specific nodes see
# "man pbs_resources"
#PBS -V
#PBS -l nodes=1:csrc-gpu
#PBS -N env
#PBS -j oe
#PBS -r n
#PBS -q batch
cd $PBS_O_WORKDIR

echo -------------------------------------------------------
echo -n 'Job is running on node '; cat $PBS_NODEFILE
echo -------------------------------------------------------
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo -------------------------------------------------------
echo -------------------------------------------------------

mpiexec -np 1 -hostfile $PBS_NODEFILE /usr/local/cuda/bin/nvcc -V
echo -------------------------------------------------------
echo -------------------------------------------------------
mpiexec -np 1 -hostfile $PBS_NODEFILE /bin/cat  /proc/version
```

Check tuckoo ENV on one of the nodes

```
[mthomas@tuckoo cudatests]$ cat env.o33799
-------------------------------------------------------
Job is running on node node10
-------------------------------------------------------
PBS: qsub is running on tuckoo.sdsu.edu
PBS: originating queue is batch
PBS: executing queue is batch
PBS: working directory is /home/mthomas/pardev/cuda/cudatests
PBS: execution mode is PBS_BATCH
PBS: job identifier is 33799.tuckoo.sdsu.edu
PBS: job name is env
PBS: node file is /var/spool/torque/aux//33799.tuckoo.sdsu.edu
PBS: current home directory is /home/mthomas
PBS: PATH = /opt/pgi/linux86-64/2011/mpi/mpich/include/:/usr/lib64/qt-3.3/bin:/usr/local/bin:/bin:
/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/usr/lib64/openmpi/bin:/usr/local/torque/bin:
/usr/local/torque/sbin:/usr/local/cuda/bin:/usr/local/tau/x86_64/bin:
/usr/local/vampirtrace/bin:/opt/pgi/linux86-64/11.0/bin:/home/mthomas/bin
-------------------------------------------------------
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2012 NVIDIA Corporation
Built on Thu_Apr__5_00:24:31_PDT_2012
Cuda compilation tools, release 4.2, V0.2.1221
-------------------------------------------------------
Linux version 2.6.32-220.17.1.el6.x86_64 (mockbuild@c6b5.bsys.dev.centos.org) (gcc version 4.4.6 20110731 (Red Hat 4.4.6-3) (GC
```

obtaining device information: enum_gpu.cu (1)

```
#include "../common/book.h"
int main( void ) {
    cudaDeviceProp  prop;
    int count;
    HANDLE_ERROR( cudaGetDeviceCount( &count ) );
    for (int i=0; i< count; i++) {
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
        printf( "   --- General Information for device %d ---\n", i );
        printf( "Name:  %s\n", prop.name );
        printf( "Compute capability:  %d.%d\n", prop.major, prop.minor );
        printf( "Clock rate:  %d\n", prop.clockRate );
        printf( "Device copy overlap:  " );
        if (prop.deviceOverlap)
            printf( "Enabled\n" );
        else
            printf( "Disabled\n");
        printf( "Kernel execution timeout :  " );
        if (prop.kernelExecTimeoutEnabled)
            printf( "Enabled\n" );
        else
            printf( "Disabled\n" );
```

obtaining device information: enum_gpu.cu (2)

```
        printf( "   --- Memory Information for device %d ---\n", i );
        printf( "Total global mem:  %ld\n", prop.totalGlobalMem );
        printf( "Total constant Mem:  %ld\n", prop.totalConstMem );
        printf( "Max mem pitch:  %ld\n", prop.memPitch );
        printf( "Texture Alignment:  %ld\n", prop.textureAlignment );
        printf( "   --- MP Information for device %d ---\n", i );
        printf( "Multiprocessor count:  %d\n",
                    prop.multiProcessorCount );
        printf( "Shared mem per mp:  %ld\n", prop.sharedMemPerBlock );
        printf( "Registers per mp:  %d\n", prop.regsPerBlock );
        printf( "Threads in warp:  %d\n", prop.warpSize );

        printf( "Max threads per block:  %d\n",
                    prop.maxThreadsPerBlock );
        printf( "Max thread dimensions:  (%d, %d, %d)\n",
                    prop.maxThreadsDim[0], prop.maxThreadsDim[1],
                    prop.maxThreadsDim[2] );
        printf( "Max grid dimensions:  (%d, %d, %d)\n",
                    prop.maxGridSize[0], prop.maxGridSize[1],
                    prop.maxGridSize[2] );
        printf( "\n" );
        }
}
```

```
-----------------------------------------------
--- General Information for device 0 ---
Name:  Tesla C1060
Compute capability:  1.3
Clock rate:  1296000
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
--- Memory Information for device 0 ---
Total global mem:  4294770688
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  256
--- MP Information for device 0 ---
Multiprocessor count:  30
Shared mem per mp:  16384
Registers per mp:  16384
Threads in warp:  32
Max threads per block:  512
Max thread dimensions:  (512, 512, 64)
Max grid dimensions:  (65535, 65535, 1)
-----------------------------------------------
--- General Information for device 2 ---
Name:  GeForce GT 240
Compute capability:  1.2
Clock rate:  1340000
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
--- Memory Information for device 2 ---
Total global mem:  1073020928
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  256
```

```
-----------------------------------------------
--- General Information for device 1 ---
Name:  Tesla C1060
Compute capability:  1.3
Clock rate:  1296000
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
   --- Memory Information for device 1 ---
Total global mem:  4294770688
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  256
   --- MP Information for device 1 ---
Multiprocessor count:  30
Shared mem per mp:  16384
Registers per mp:  16384
Threads in warp:  32
Max threads per block:  512
Max thread dimensions:  (512, 512, 64)
Max grid dimensions:  (65535, 65535, 1)
-----------------------------------------------
   --- MP Information for device 2 ---
Multiprocessor count:  12
Shared mem per mp:  16384
Registers per mp:  16384
Threads in warp:  32
Max threads per block:  512
Max thread dimensions:  (512, 512, 64)
Max grid dimensions:  (65535, 65535, 1)
```

Compile & run CUDA code on the login node

- you can install CUDA toolkit and compile code without a GPU installed
- you can RUN the code from the command line on some machines.

```
[mthomas@tuckoo hello]$ cat simple_hello.cu
/*
 * simple_hello.cu
 *
 * Copyright 1993-2010 NVIDIA Corporation.
 *     All rights reserved.
 *
  */
#include <stdio.h>
#include <stdlib.h>

int main( void ) {
   int deviceCount;

   cudaGetDeviceCount( &deviceCount );
   printf("Hello, World! You have %d devices\n",
        deviceCount );


    return 0;
}
```

```
[mthomas@tuckoo chapter03]$  ./hello_world
Hello, World! You have 0 devices
```

Running first Job: simple_kernel.cu

```
[mthomas@tuckoo hello]$ cat simple_kernel.cu
/*
 * Copyright 1993-2010 NVIDIA Corporation.
 *     All rights reserved.
 *
  */
#include <stdio.h>

__global__ void kernel( void ) {
}

int main( void ) {
    int deviceCount;

    kernel<<<1,1>>>();

    cudaGetDeviceCount( &deviceCount );
    printf("Hello, World! You have %d devices\n",
           deviceCount );

    return 0;
}
```

```
[mthomas@tuckoo hello]$ nvcc -o simple_kernel simple_kernel.cu
nvcc warning : The 'compute_10' and 'sm_10' architectures are d
[mthomas@tuckoo hello]$
[mthomas@tuckoo hello]$
[mthomas@tuckoo hello]$
[mthomas@tuckoo hello]$  /simple_kernel
Hello, World! You have 0 devices
```

CUDA Batch Script Example

```
[mthomas@tuckoo chapter03]$ cat hello_world.bat
#!/bin/sh
# this example batch script requests many processors...
# for more info on requesting specific nodes see
# "man pbs_resources"
#PBS -V
#PBS -l nodes=node9:ppn=1
#PBS -N simple_hello
#PBS -j oe
#PBS -q batch
cd $PBS_O_WORKDIR

echo -------------------------------------------------------
echo -n 'Job is running on node '; cat $PBS_NODEFILE
echo -------------------------------------------------------
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo -------------------------------------------------------
NCORES=`wc -w < $PBS_NODEFILE`
echo "many-test using $NCORES cores..."
./simple_hello
```

Run Job using Batch Queue: hello_world.bat - OUTPUT

```
[mthomas@tuckoo chapter03]$ cat hello_world.o18574
-------------------------------------------------------
Job is running on node node9
-------------------------------------------------------
PBS: qsub is running on tuckoo.sdsu.edu
PBS: originating queue is batch
PBS: executing queue is batch
PBS: working directory is /home/mthomas/pardev/cuda/cuda_by_example/chapter03
PBS: execution mode is PBS_BATCH
PBS: job identifier is 18574.tuckoo.sdsu.edu
PBS: job name is hello_world
PBS: node file is /var/spool/torque/aux//18574.tuckoo.sdsu.edu
PBS: current home directory is /home/mthomas
PBS: PATH = /usr/lib64/qt-3.3/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/usr/local/openmpi/bin:/usr/loca
-------------------------------------------------------
hello_world using 1 cores...
Hello, World! You have 2 devices
Hello, World! You have 2 devices
Hello, World! You have 2 devices
Hello, World! You have 2 devices
Hello, World! You have 2 devices
Hello, World! You have 2 devices
Hello, World! You have 2 devices
Hello, World! You have 2 devices
Hello, World! You have 2 devices
Hello, World! You have 2 devices
```

CUDA Hello World

```
#include <stdio.h>
__global__ void hello_kernel(float * x) {
// By Ingemar Ragnemalm 2010
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void hello(char *a, int *b)  {
        a[threadIdx.x] += b[threadIdx.x];
}

int main()  {
        char a[N] = "Hello \0\0\0\0\0\0";
        int b[N] = {15, 10, 6, 0, -11, 1, 0,
             0, 0, 0, 0, 0, 0, 0, 0, 0};

        char *ad;
        int *bd;
        const int csize = N*sizeof(char);
        const int isize = N*sizeof(int);

        printf("%s", a);

        cudaMalloc( (void**)&ad, csize );
        cudaMalloc( (void**)&bd, isize );
        cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
        cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );

}
```

```
dim3 dimBlock( blocksize, 1 );
dim3 dimGrid( 1, 1 );
hello<<<dimGrid, dimBlock>>>(ad, bd);
cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost );
cudaFree( ad );   cudaFree( bd );
printf("%s\n", a);
return EXIT_SUCCESS;
```

simple_kernel_params.cu

```
#include "../common/book.h"

__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}

int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );

    add<<<1,1>>>( 2, 7, dev_c );

    HANDLE_ERROR( cudaMemcpy( &c, dev_c, sizeof(int),
                              cudaMemcpyDeviceToHost ) );
    printf( "2 + 7 = %d\n", c );
    HANDLE_ERROR( cudaFree( dev_c ) );

    return 0;
}
```

simple_kernel_params.cu

```
[mthomas@tuckoo chapter03]$ cat simple_kernel_params.o3901
-------------------------------------------------------
Job is running on node csrc-gpu
-------------------------------------------------------
PBS: qsub is running on tuckoo.sdsu.edu
PBS: originating queue is batch
PBS: executing queue is batch
PBS: working directory is /home/mthomas/pardev/cuda/cuda_by_example/chapter03
PBS: execution mode is PBS_BATCH
PBS: job identifier is 3901.tuckoo.sdsu.edu
PBS: job name is simple_kernel_params
PBS: node file is /var/spool/torque/aux//3901.tuckoo.sdsu.edu
PBS: current home directory is /home/mthomas
PBS: PATH = /usr/lib64/qt-3.3/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/s
-------------------------------------------------------
simple_kernel_params using 1 cores...
2 + 7 = 9
2 + 7 = 9
...
2 + 7 = 9
2 + 7 = 9
2 + 7 = 9
```

Compute Unified Device Architecture (CUDA) Overview

Introduction to Compute Unified Device Architecture (CUDA, K&W Ch3; S&K, Ch3)

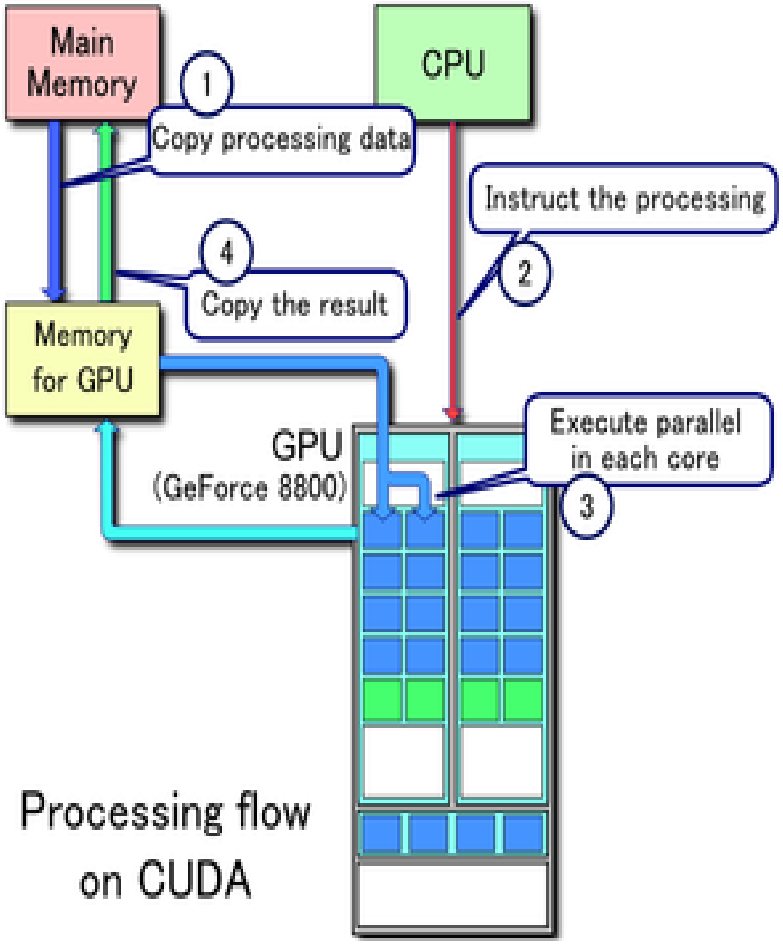
Outline:

- Basic Program Example
- The CUDA Kernel
- Passing Parameters
- Memory Management

CUDA (Compute Unified Device Architecture)

Example of CUDA processing flow:

1. CPU initializes, allocates, copies data from main memory to GPU memory
2. CPU sends instructions to GPU
3. GPU executes parallel code in each core
4. GPU Copies the result from GPU mem to main mem



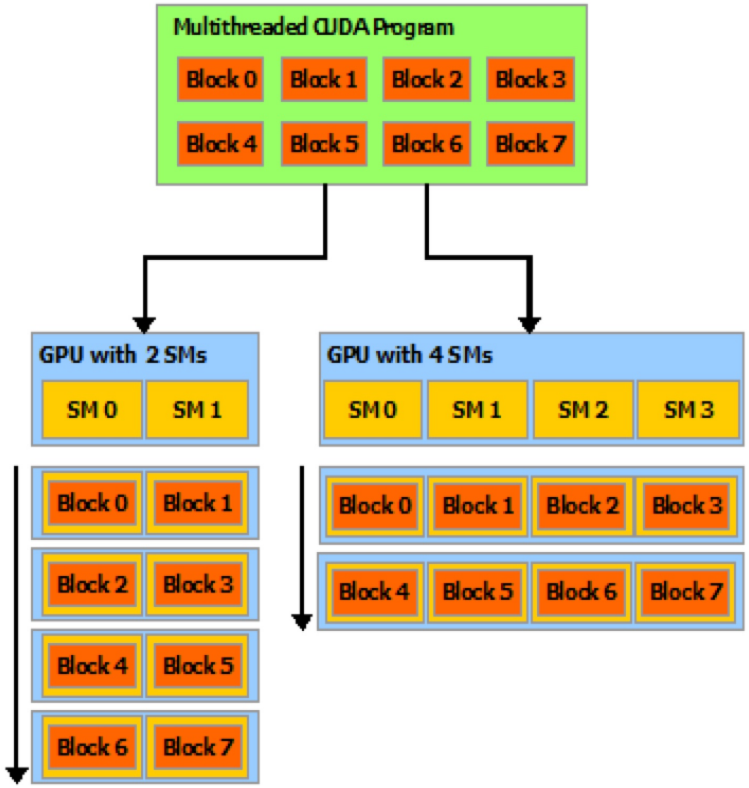
Source: http://en.wikipedia.org/wiki/CUDA

CUDA API (1)

- CUDA C is a variant of C with extensions to define:
  - where a function executes (host CPU or the GPU)
  - where a variable is located in the CPU or GPU address space
  - execution parallelism of kernel function distributed in terms of grids and blocks
  - defines variables for grid, block dimensions, indices for blocks and threads
- Requires the *nvcc* 64-bit compiler and the CUDA driver outputs PTX (Parallel Thread eXecution, NVIDIA pseudo-assembly language) , CUDA, standard C binaries
- CUDA run-time JIT compiler (optional);
  compiles PTX code into native operations
- math libraries, cuFFT, cuBLAS and cuDPP (optional)

CUDA Programming Model

- Mainstream processor chips are parallel systems: multicore CPUs and many core GPUs
- CUDA/GPU provides three key abstractions:
  - hierarchy of thread groups
  - shared memory
  - barrier synchronization
- fine-grained data & thread parallelism, nested within coarse-grained data & task parallelism
- partitions problem into coarse sub-probs solved with parallel independent blocks of threads
- sub-problems divided into finer pieces solved in parallel by all threads in block
- GPU has array of Streaming Multiprocs (SMs)
- Multithreaded program partitioned into blocks of threads that execute independently from each other
- Scales: GPU (more MPs) executes in less time than GPU (fewer MPs).



Source: NVIDIA cuda-c-programming-guide

CUDA Kernel Basics

CUDA Code Example: simple_hello.cu (K&S Ch3)

```
[mthomas@tuckoo hello]$ cat simple_hello.cu
/*
 * Copyright 1993-2010 NVIDIA
 *     Corporation.
 *     All rights reserved.
  */
#include <stdio.h>

__global__ void mykernel( void ) {
}

int main( void ) {
    mykernel<<<1,1>>>();
    printf( "Hello, GPU World!\n" );
    return 0;
}
```

CUDA code highlights:

- *mykernel* $<<<1,1>>>()$ directs the function to be run on the device
- *mykernel*() is an empty function
- *__global__* is a CUDA directive that tells system to run this function on the GPU device

CUDA API: Kernel

In its simplest form it looks like:

$$kernelRoutine <<< gridDim, blockDim >>> (args)$$

Kernel runs on the device. It is executed by threads, each of which knows about:

- variables passed as arguments
- pointers to arrays in device memory (also arguments)
- global constants in device memory
- shared memory and private registers/local variables
- some special variables:
  - *gridDim*: size (or dimensions) of grid of blocks
  - *blockIdx* : index (or 2D/3D indices)of block
  - *blockDim*: size (or dimensions) of each block
  - *threadIdx*: index (or 2D/3D indices) of thread

Function Type Qualifiers

Function type qualifiers specify whether a function executes on the host or on the device and whether it is callable from the host or from the device:

- __device__
  - Executed on GPU
  - Launched on GPU
- __global__
  - Executed on device
  - Callable from host
  - Callable from the device for devices of compute capability 3.x
- __host__ (optional)
  - Executed on host
  - Callable from host only

Source:

`http://docs.nvidia.com/cuda/cuda-c-programming-guide/#function-type-qualifiers`

Grids and Blocks

- A *Grid* is a collection of blocks:
  - *gridDim*: size (dimensions) of grid of blocks
  - *blockIdx* : index (2D/3D indices) of block
- A *Block* is a collection of threads (columns):
  - *blockDim*: size (dimensions) of each block
  - *threadIdx*: index (or 2D/3D indices) of thread
- *Threads* execute the *kernel* code on *device*:

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Source: *Cuda By Example (Ch 5)*

Two types of parallelism:

Block Parallelism

Launch N blocks with 1 thread each:

$$add <<< N, 1 >>> (dev\_a, \quad dev\_b, \quad dev\_c) >>>$$

Thread Parallelism

Launch 1 block with N threads:

$$add <<< 1, N >>> (dev\_a, \quad dev\_b, \quad dev\_c) >>>$$

We will look at examples for each type of parallel mechanisms.

Memory Allocation

- CPU: malloc, calloc, free, cudaMallocHost, cudaFreeHost
- GPU: cudaMalloc, cudaMallocPitch, cudaFree, cudaMallocArray, cudaFreeArray

Passing Parameters to the Kernel

simple_kernel_params.cu (part 1)

```
#include <iostream>
#include "book.h"

__global__ void add( int a, int b, int *c ) {
        *c = a + b;
    }
int main( void ) {
 int c;
 int *dev_c;

 /* allocate memory on the device for the variable */
 HANDLE_ERROR(
   cudaMalloc((void**)&dev_c, sizeof(int) ) );

 /* nothing to copy -- no call to cudaMemcpy */

 /* launch the kernel */
 add<<<1,1>>>( 2, 7, dev_c );

 /* copy results back from device to the host */
 HANDLE_ERROR(
   cudaMemcpy(&c,dev_c,sizeof(int),cudaMemcpyDeviceToHost)
 );

 printf( "2 + 7 = %d\n", c );

 cudaFree( dev_c );

 return 0;
}
```

- The Kernel: $add <<< 1, 1 >>> (2, 7, dev\_c)$ runs on the device.
- *__global__* is a CUDA directive that tells system to run this function on the GPU device
- Kernel passing variables that are modified on the device.
- using 1 block with 1 thread
- Result passed from the device back to the host
- Must use pointers

simple_kernel_params.cu (part 1)

```
[cuda_by_example/chapter03] nvcc -o simple_add   simple_add.cu

[cuda_by_example/chapter03] qsub simple_add.bat
7987.tuckoo.sdsu.edu
[cuda_by_example/chapter03]$ cat simple_device_call.o7987

simple_device_call using 1 cores...
2 + 7 = 9
```

```
#!/bin/bash
#
#
#PBS -V
#PBS -l nodes=node9:ppn=1
#PBS -N simple_add
#PBS -j oe
#PBS -r n
#PBS -q batch
cd $PBS_O_WORKDIR

echo "Running simple_add."
./simple_add
```

CUDA Block Parallelism (K&W Ch3; S&K, Ch4)

Block Parallelism

- Simple add: CPU host launched a simple kernel that ran serially on the GPU device.
- Blocks: fundamental way that CUDA exposes parallelism: data parallelism
- Block parallelism will launch a device kernel that performs its computations in parallel.
- We will look at array addition:
  $add <<< N, 1 >>> (dev\_a, dev\_b, dev\_c);$
- put multiple copies of the kernel onto the blocks

Serial CPU Code for Vector Add: add_loop_cpu.c

```
/*
 * Copyright 1993-2010 NVIDIA Corporation.  All rights reserved.
 *
 */
#include "../common/book.h"

#define N   10

void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}

int main( void ) {
    int a[N], b[N], c[N];

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    add( a, b, c );

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    return 0;
}
```

Pseudocode for CPU and GPU Vector Add

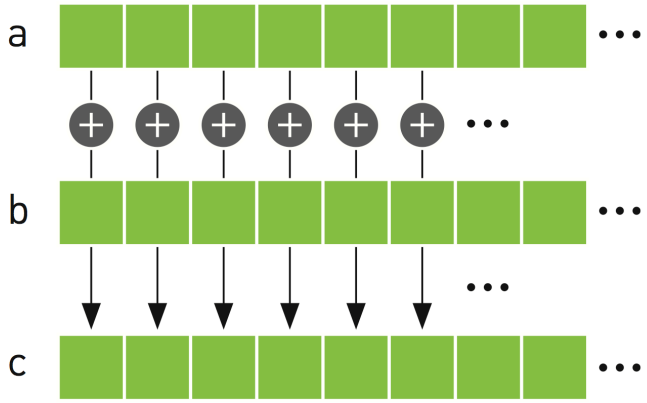
CPU

```
void add( int *a, int *b, int *c ) {
       int tid = 0;
       while (tid < N) {
           c[tid] = a[tid] + b[tid];
           tid += 2; }
}
. . . . . . . .
int main( void ) {
    . . . . . . . .
    add( a, b, c );
}
```

GPU

```
__global__ void add( int *a, int *b, int *c ) {
        int tid = blockIdx.x;
        if (tid < N)
            c[tid] = a[tid] + b[tid];
}
. . . . . . . .
int main( void ) {
    . . . . . . . .
    // set the number of parallel blocks
    // on device that will execute kernel
    // max number is 65,535 blocks
   add<<<N,1>>>( dev_a, dev_b, dev_c );
}
```

Summing two vectors



Recall: Grid and Block Assignment

kernelRoutine <<< *gridDim*, *blockDim* >>> (*args*)

- A *Grid* is a collection of blocks:
  - *gridDim*: size (dimensions) of grid of blocks
  - *blockIdx* : index (2D/3D indices) of block
- A *Block* is a collection of threads (columns):
  - *blockDim*: size (dimensions) of each block
  - *threadIdx*: index (or 2D/3D indices) of thread
- *Threads* execute the *kernel* code on *device*:

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Source: *Cuda By Example (Ch 5)*

GPU Code for the add kernel demonstrating how to obtain the block index ID.

```
#include "../common/book.h"

#define N   10

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Example above is for GPU code with N blocks. Thread ID (or rank) is obtained from the block index object

Source: *Cuda By Example*

GPU block assignment for add kernel for $N = 4$ blocks, after *int tid* $=$ *blockIdx.x* has been computed.

BLOCK 1

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 2

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 3

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 4

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

CUDA: Dynamic Variable Assignments

- The code below contains code for the simple add example from K&S
- It contains two key examples that show how to:
  - time your code using CUDA / *Event Timers*
  - pass variables from the command line and make them available to the device.
  - dynamically allocate memory on the host and device using dynamic variables.

```
/* File:   add_loop_gpu.cu
 *
 * Written By:   Mary Thomas (mthomas@mail.sdsu.edu)
 * Date:         Dec, 2014
 * Based on:     CUDA SDK code add_loop_gpu.cu
 * Description:  Reads number of threads from the command line
 *               and sets this as a global device variable.
 *
 * Copyright 1993-2010 NVIDIA Corporation.  All rights reserved.
 *
 * NVIDIA Corporation and its licensors retain all intellectual property and
 * proprietary rights in and to this software and related documentation.
 * Any use, reproduction, disclosure, or distribution of this software
 * and related documentation without an express license agreement from
 * NVIDIA Corporation is strictly prohibited.
 *
 * Please refer to the applicable NVIDIA end user license agreement (EULA)
 * associated with this source code for terms and conditions that govern
 * your use of this NVIDIA software.
 *
 */

#include <stdio.h>

//#define N   65535+10
__device__ int d_Nthds;
 __global__ void checkDeviceThdCount(int *t) { *t = d_Nthds; }

__global__ void add( int *a, int *b, int *c) {
    int tid = blockIdx.x;    // this thread handles the data at its thread id
   if (tid < d_Nthds)
        c[tid] = a[tid] + b[tid];
}
```

```
int main( int argc, char** argv ) {
    int h_N = atoi(argv[1]);
    int a[h_N], b[h_N], c[h_N];
    int h_tid[h_N], h_cpuid[h_N];
    int *dev_a, *dev_b, *dev_c;
    int *d_tid;
    int i,j,k;
    int h_N2 = h_N;

    float time;
    cudaEvent_t start, stop;

    cudaEventCreate(&start) ;
    cudaEventCreate(&stop) ;
    cudaEventRecord(start, 0) ;

    // Check some key device properties
    int devCount=0;
    cudaGetDeviceCount(&devCount);
    printf("There are %d CUDA devices.\n", devCount);
    for (int i = 0; i < devCount; ++i)
    {
        // Get device properties
        printf("\nCUDA Device #%d\n", i);
        cudaDeviceProp devProp;
        cudaGetDeviceProperties(&devProp, i);
        printf("Device Name: %s \n",  devProp.name);
        printf("Maximum threads per block:     %d\n",  devProp.maxThreadsPerBlock);
        printf("Maximum dimensions of block: blockDim[0,1,2]=[");
        for (int i = 0; i < 3; ++i)
            printf(" %d ", devProp.maxThreadsDim[i]);
        printf("] \n");
    }
```

```
// set the number of threads to the global variable d_Nthds
  int h_Ndevice;
  int *d_N;
  cudaMemcpyToSymbol(d_Nthds, &h_N, sizeof(int), 0, cudaMemcpyHostToDevice);
  cudaMalloc( (void**)&d_N, sizeof(int) );
  checkDeviceThdCount<<<1,1>>>(d_N);
   cudaMemcpy( &h_Ndevice, d_N, sizeof(int), cudaMemcpyDeviceToHost ) ;
  printf("h_N = %d, h_Ndevice=%d \n", h_N, h_Ndevice);
  cudaThreadSynchronize();


  // allocate the memory on the GPU
  cudaMalloc( (void**)&dev_a, h_N * sizeof(int) ) ;
  cudaMalloc( (void**)&dev_b, h_N * sizeof(int) ) ;
  cudaMalloc( (void**)&dev_c, h_N * sizeof(int) ) ;

  // fill the arrays 'a' and 'b' on the CPU
  for (i=0; i<h_N; i++) {
      a[i] = i+1;
      b[i] = (i+1) * (i+1);
  }

  // copy the arrays 'a' and 'b' to the GPU
   cudaMemcpy( dev_a, a, h_N * sizeof(int), cudaMemcpyHostToDevice ) ;
   cudaMemcpy( dev_b, b, h_N * sizeof(int), cudaMemcpyHostToDevice ) ;

  //getThreadInfo<<<h_N,1>>>( d_tid,d_cpuid );
  add<<<h_N,1>>>( dev_a, dev_b, dev_c);

  // copy the array 'c' back from the GPU to the CPU
  cudaMemcpy( c, dev_c, h_N * sizeof(int), cudaMemcpyDeviceToHost ) ;
```

```
    //print out small arrays
    if( h_N < 11 )
    {
        // display the results
        for (i=0; i<h_N; i++) {
            printf( "%d + %d = %d\n", a[i], b[i], c[i] );
        }

       i=0; j=h_N/2; k=h_N-1;
       printf( "Arr1[%d]: %d + %d = %d\n",i, a[i], b[i], c[i] );
       printf( "Arr1[%d]: %d + %d = %d\n",j, a[j], b[j], c[j] );
       printf( "Arr1[%d]: %d + %d = %d\n",k, a[k], b[k], c[k] );

       printf( "Arr2[%d]: %d + %d = %d\n",i, a[i], b[i], c[i] );
       printf( "Arr2[%d]: %d + %d = %d\n",j, a[j], b[j], c[j] );
       printf( "Arr2[%d]: %d + %d = %d\n",k, a[k], b[k], c[k] );
    }

    // free the memory allocated on the GPU
     cudaFree( dev_a ) ;
     cudaFree( dev_b ) ;
     cudaFree( dev_c ) ;

    //calculate elapsed time:
    cudaEventRecord(stop, 0) ;
    cudaEventSynchronize(stop) ;
    //Computes the elapsed time between two events (in milliseconds)
    cudaEventElapsedTime(&time, start, stop) ;
    printf("GPU Nthreads=%d, Telap(msec)= %26.16f\n",h_N,time);

    return 0;
}
```

Batch Script for Running CUDA add_loop_gpu.cu

```
#!/bin/sh
#
#  to run:
#        qsub -v T=10 bat.addloop
#
#PBS -V
#PBS -l nodes=1:node9:ppn=1
#PBS -N addloop
#PBS -j oe
#PBS -r n
#PBS -q batch
cd $PBS_O_WORKDIR

echo "running add_loop_gpu using $T threads"
./add_loop_gpu $T
```

add_loop_gpu.cu (output), also showing device information and timing diagnostics

```
[mthomas@tuckoo chapter04]$qsub -v NTHDS=10 bat.addloop
running add_loop_gpu using 10 threads
There are 2 CUDA devices.

CUDA Device #0
Device Name: GeForce GTX 480
Maximum threads per block:     1024
Maximum dimensions of block: blockDim[0,1,2]=[ 1024  1024  64 ]

CUDA Device #1
Device Name: GeForce GTX 480
Maximum threads per block:     1024
Maximum dimensions of block: blockDim[0,1,2]=[ 1024  1024  64 ]
h_N = 10, h_Ndevice=10
1 + 1 = 2
2 + 4 = 6
3 + 9 = 12
4 + 16 = 20
5 + 25 = 30
6 + 36 = 42
7 + 49 = 56
8 + 64 = 72
9 + 81 = 90
10 + 100 = 110
Arr1[0]: 1 + 1 = 2
Arr1[5]: 6 + 36 = 42
Arr1[9]: 10 + 100 = 110
Arr2[0]: 1 + 1 = 2
Arr2[5]: 6 + 36 = 42
Arr2[9]: 10 + 100 = 110
GPU Nthreads=10, Telap(msec)=         0.5985599756240845
```

Timing CUDA code CudaEvent Timers (output)

```
int main( int argc, char** argv ) {
    . . .
    float time;
    cudaEvent_t start, stop;

    cudaEventCreate(&start) ;
    cudaEventCreate(&stop) ;
    . . .
    cudaEventRecord(start, 0) ;
    . . .
    . . .
    //calculate elapsed time:
    cudaEventRecord(stop, 0) ;
    cudaEventSynchronize(stop) ;
    //Computes the elapsed time between two events (in milliseconds)
    cudaEventElapsedTime(&time, start, stop) ;
    printf("GPU Nthreads=%d, Telap(msec)= %26.16f\n",h_N,time);
```

See S&K, Chapter 6

Timing Results for add_vector output using CudaEvent Timers (output)

```
serial: Nthreads=10000,       Telap(msec) =       184.0
serial: Nthreads=1000000,     Telap(msec) =     15143.0
serial: Nthreads=100000000,   Telap(msec) =    181107.0


   GPU: Nthreads=10000,      Telap(msec) =         1.1845
   GPU: Nthreads=1000000,    Telap(msec) =        11.185
   GPU: Nthreads=100000000,  Telap(msec) =        661.78
```

What happens to global variables?

```
Set up test code to see results for very large values of N:

    i=0;  j=N/2;  k=N;
    printf( "Arr1[%d]: %d + %d = %d\n",i, a[i], b[i], c[i] );
    printf( "Arr1[%d]: %d + %d = %d\n",j, a[j], b[j], c[j] );
    printf( "Arr1[%d]: %d + %d = %d\n",k, a[k], b[k], c[k] );

    i=0;  j=N2/2;  k=N2;
    printf( "Arr2[%d]: %d + %d = %d\n",i, a[i], b[i], c[i] );
    printf( "Arr2[%d]: %d + %d = %d\n",j, a[j], b[j], c[j] );
    printf( "Arr2[%d]: %d + %d = %d\n",k, a[k], b[k], c[k] );
```

```
Arr1[0]: 0 + 0 = 32896
Arr1[65540]: 65540 + 524304 = 0
Arr1[65545]: 11028 + 0 = 0
Arr2[0]: 0 + 0 = 32896
Arr2[32772]: 32772 + 1074003984 = 0
Arr2[65545]: 11028 + 0 = 0
```

What happens when the number of threads is >> number of blocks?

- Need to distribute the threads
- Cannot exceed *maxThreadsPerBlock*, typically 512
- Need a combination of threads and blocks
- Algorithm to convert from 2D space (multiple blocks and multiple threads per block) to 1D:

  $int \quad tid = threadIdx.x + blockIdx.x * blockDim.x;$
- Note: *blockDim* is constant

add.cu

```
#include "../common/book.h"

#define N   (33 * 1024)

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}

int main( void ) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the CPU
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );

    // allocate the memory on the GPU
    HANDLE_ERROR(cudaMalloc((void**)&dev_a,N*sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_b,N*sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_c,N*sizeof(int)));

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }
```

```
    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );

    add<<<128,128>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                              cudaMemcpyDeviceToHost ) );

    // verify that the GPU did the work we requested
    bool success = true;
    for (int i=0; i<N; i++) {
        if ((a[i] + b[i]) != c[i]) {
            printf( "Error:  %d + %d != %d\n", a[i], b[i], c[i]
            success = false;
        }
    }
    if (success)    printf( "We did it!\n" );

    // free the memory we allocated on the GPU
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_c ) );

    // free the memory we allocated on the CPU
    free( a );   free( b );    free( c );

    return 0;
    }
```

Two dimensional arrangement of a collection of blocks and threads

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Source: *Cuda By Example, Figure 5.1*

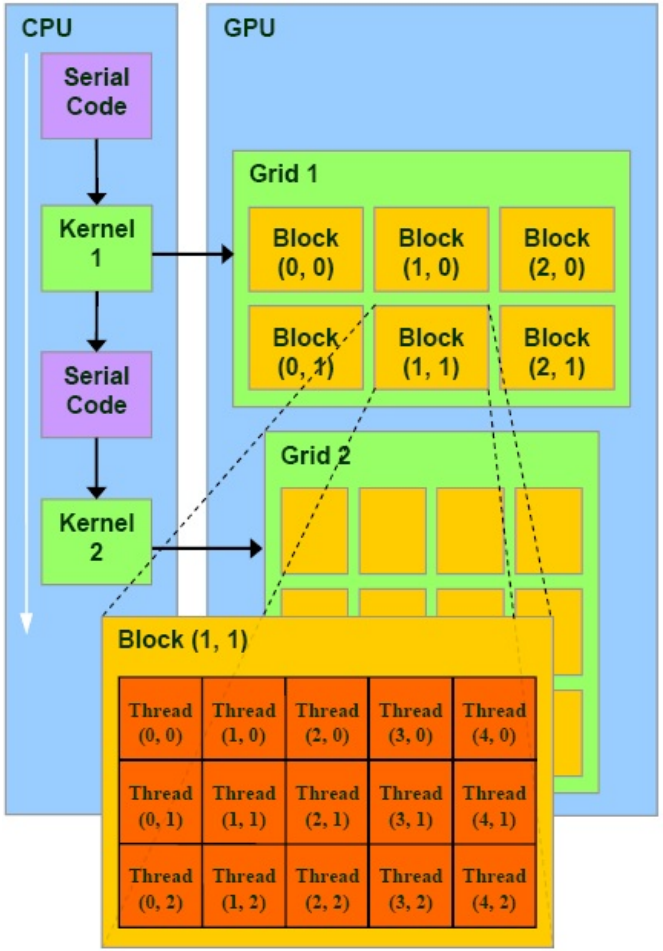
CUDA Hello World: Using dimGrid and dimBlock

```
#include <stdio.h>
__global__ void hello_kernel(float * x) {
// By Ingemar Ragnemalm 2010
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void hello(char *a, int *b)  {
        a[threadIdx.x] += b[threadIdx.x];
}

int main()  {
        char a[N] = "Hello \0\0\0\0\0\0";
        int b[N] = {15, 10, 6, 0, -11, 1, 0,
             0, 0, 0, 0, 0, 0, 0, 0, 0};

        char *ad;
        int *bd;
        const int csize = N*sizeof(char);
        const int isize = N*sizeof(int);

        printf("%s", a);

        cudaMalloc( (void**)&ad, csize );
        cudaMalloc( (void**)&bd, isize );

        cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
        cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );

}
```

```
        dim3 dimBlock( blocksize, 1 );

        dim3 dimGrid( 1, 1 );

        hello<<<dimGrid, dimBlock>>>(ad, bd);

        cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost );

        cudaFree( ad );   cudaFree( bd );

        printf("%s\n", a);
        return EXIT_SUCCESS;
```

Recall: Defining GPU Threads and Blocks

- Looking at Device: Nvidia Tesla C1060
- Kernels run on GPU threads
- Grid: organized as 2D array of blocks:
  - Maximum sizes of each dimension: $[gridDim.x \times gridDim.y \times gridDim.z]$ $= (65,536 \times 65,536 \times 1)$ blocks
- Block: $3D$ collection of threads
  - Max threads per block: 512
  - Max thread dimensions: (512, 512, 64) $[blockDim.x * blockDim.y * blockDim.z]$ $MaxThds/Block \leq 1024$
- threads composing a thread block must:
  - execute the same kernel
  - share data: issued to the same core
  - *Warp*: group of 32 threads; min size of data processed in SIMD fashion by CUDA multiprocessor.



Source: http://hothardware.com/Articles/NVIDIA-GF100-Architecture-and-Feature-Preview

Thread Parallelism

- We split the blocks into threads
- Threads can communicate with each other
- You can share information between blocks (using global memory and atomics, for example), but not global synchronization.
- Threads can be synchronized using *syncthreads*().
- Block parallelism: call kernel with N blocks, 1 thread per block
  add<<<N,1>>>(dev_a, dev_b, dev_c);
  N blocks x 1 Thread/block = N parallel threads
- Thread parallelism: call kernel with 1 block, N threads per block
  add<<<1,N>>>(dev_a, dev_b, dev_c);
  1 block x N Thread/block = N parallel threads
- Ultimately, we combine both models.

add_loop.cu, using 1 block and N threads

```
#include "../common/book.h"

#define N   10

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc((void**)&dev_a,
            N*sizeof(int)));
    HANDLE_ERROR( cudaMalloc((void**)&dev_b,
            N*sizeof(int)));
    HANDLE_ERROR( cudaMalloc((void**)&dev_c,
            N*sizeof(int)));

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i * i;
    }
```

```
// copy the arrays 'a' and 'b' to the GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
        cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
        cudaMemcpyHostToDevice ) );


/* call kernel with 1 block, N threads per block */
add<<<1,N>>>( dev_a, dev_b, dev_c );


// copy the array 'c' back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
        cudaMemcpyDeviceToHost ) );

// display the results
for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}

// free the memory allocated on the GPU
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_c ) );

return 0;
}
```

```
 /* Thread Parallelism: Using dynamic number of threads
 * Modified By:   Mary Thomas (mthomas@mail.sdsu.edu)
 * Based on:     CUDA SDK code add_loop_gpu.cu
 */
#include <stdio.h>
//#define N   65535+10
__device__ int d_Nthds;
__global__ void checkDeviceThdCount(int *t) {*t = d_Nthds;}
__global__ void add( int *a, int *b, int *c) {
   int tid = blockIdx.x;
   if (tid < d_Nthds) {
        c[tid] = a[tid] + b[tid];
    }
}
int main( int argc, char** argv ) {
if(argc != 2) {
    printf("Usage Error: %s <N> \n",argv[0]);}
    int h_N = atoi(argv[1]);

    int a[h_N], b[h_N], c[h_N];
    int *dev_a, *dev_b, *dev_c;
    int i,j,k;    int *d_N, d_Ntmp;
    float time;
    cudaEvent_t start, stop;
    cudaEventCreate(&start) ;
    cudaEventCreate(&stop) ;
    cudaEventRecord(start, 0) ;

// set #threads to device variable d_Nthds
cudaMemcpyToSymbol(d_Nthds, &h_N, sizeof(int),
        0, cudaMemcpyHostToDevice);
cudaMalloc( (void**)&d_N, sizeof(int) );
checkDeviceThdCount<<<1,1>>>(d_N);
cudaMemcpy( &d_Ntmp, d_N, sizeof(int),
        cudaMemcpyDeviceToHost ) ;
cudaThreadSynchronize();
```

```
// fill the arrays 'a' and 'b' on the CPU
for (i=0; i<h_N; i++) {
        a[i] = i+1;
        b[i] = (i+1) * (i+1);
}
// allocate the memory on the GPU
cudaMalloc( (void**)&dev_a, h_N * sizeof(int) ) ;
cudaMalloc( (void**)&dev_b, h_N * sizeof(int) ) ;
cudaMalloc( (void**)&dev_c, h_N * sizeof(int) ) ;

// copy the arrays 'a' and 'b' to the GPU
cudaMemcpy( dev_a, a, h_N * sizeof(int),
        cudaMemcpyHostToDevice ) ;
cudaMemcpy( dev_b, b, h_N * sizeof(int),
        cudaMemcpyHostToDevice ) ;

add<<<1,h_N>>>( dev_a, dev_b, dev_c);

// copy the array 'c' back from the GPU to the CPU
cudaMemcpy( c, dev_c, h_N * sizeof(int),
        cudaMemcpyDeviceToHost ) ;

// free the memory allocated on the GPU
cudaFree( dev_a ) ;
cudaFree( dev_b ) ;
cudaFree( dev_c ) ;

//calculate elapsed time:
cudaEventRecord(stop, 0) ;
cudaEventSynchronize(stop) ;
//Compute elapsed time  (in milliseconds)
cudaEventElapsedTime(&time, start, stop) ;
printf("Nthreads=%ld, Telapsed(msec)= %26.16f\n",
        h_N,time);

return 0; }
```

Thread Parallelism:

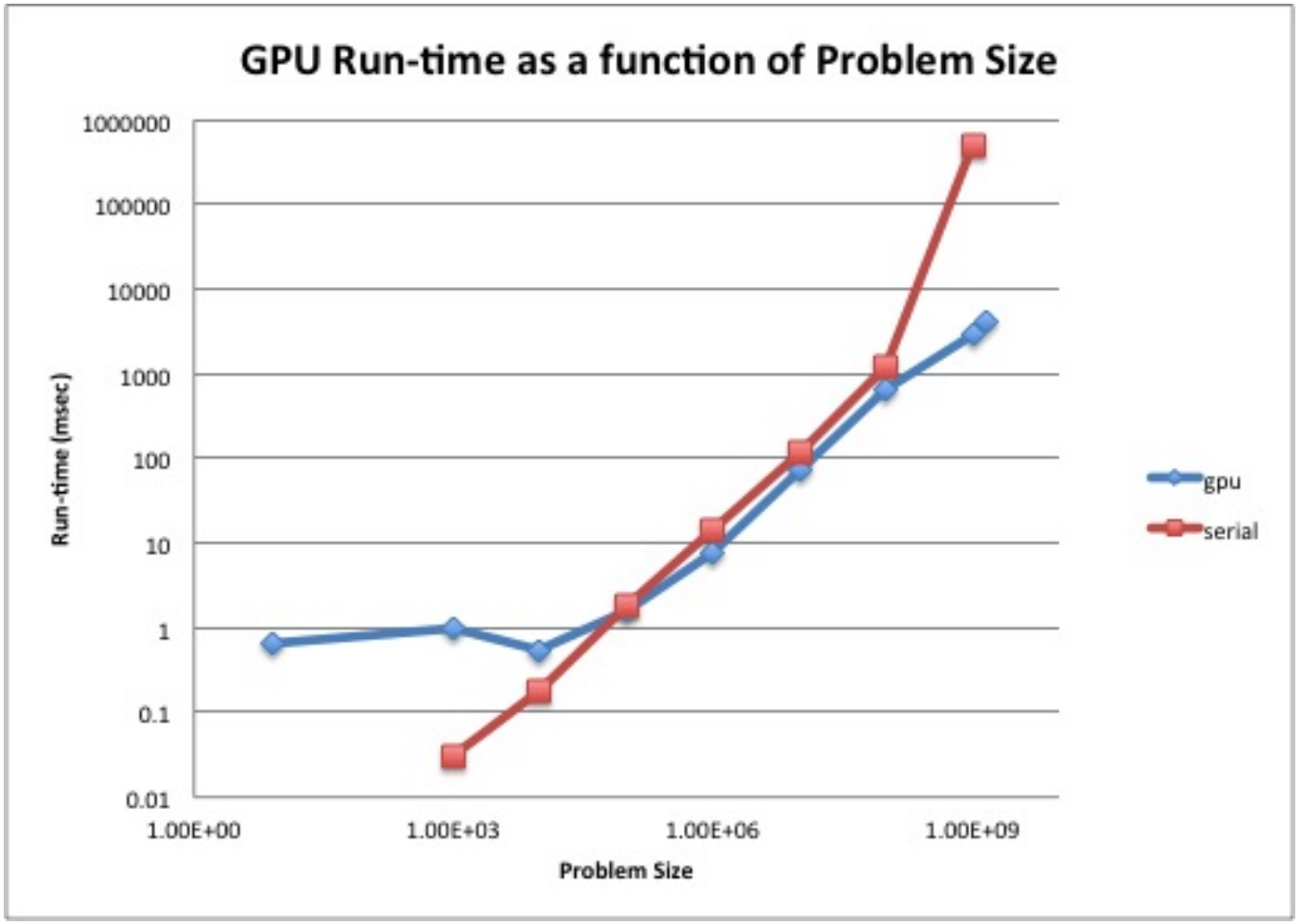
```
[mthomas@tuckoo:cuda/add_loop] nvcc -arch=sm_20 -o add_loop_gpu add_loop_gpu.cu
[mthomas@tuckoo:cuda/add_loop] qsub -v T=10 bat.addloop
8709.tuckoo.sdsu.edu
[mthomas@tuckoo:cuda/add_loop] cat addloop.o8709
running add_loop_gpu using 10 threads
There are 2 CUDA devices.
CUDA Device #0
Device Name: GeForce GTX 480
Maximum threads per block:     1024
Maximum dimensions of block: blockDim[0,1,2]=[ 1024  1024  64 ]
CUDA Device #1
Device Name: GeForce GTX 480
Maximum threads per block:     1024
Maximum dimensions of block: blockDim[0,1,2]=[ 1024  1024  64 ]
h_N = 10, d_N=1048576, d_Ntmp=10
1 + 1 = 2
2 + 4 = 6
3 + 9 = 12
4 + 16 = 20
5 + 25 = 30
6 + 36 = 42
7 + 49 = 56
8 + 64 = 72
9 + 81 = 90
10 + 100 = 110
Arr1[0]: 1 + 1 = 2
Arr1[5]: 6 + 36 = 42
Arr1[9]: 10 + 100 = 110
Arr2[0]: 1 + 1 = 2
Arr2[5]: 6 + 36 = 42
Arr2[9]: 10 + 100 = 110
GPU Nthreads=10, Telap(msec)=         0.4095999896526337
```

Thread Parallelism: add_loop_blocks.cu (output)

```
[mthomas@tuckoo]$ cat addloopser.o* | grep Telap
serial: Nthreads=10000, Telapsed(millisec) =                    184.0
serial: Nthreads=1000000, Telapsed(millisec) =                  15143.0
serial: Nthreads=100000000, Telapsed(millisec) =                 181107.0

[mthomas@tuckoo]$ cat addloopgpu.o* | grep Telap
GPU Nthreads=10000, Telap(msec)=        1.1845120191574097
GPU Nthreads=1000000, Telap(msec)=       11.1852159500122070
GPU Nthreads=100000000, Telap(msec)=      661.7844238281250000
GPU Nthreads=1410065408, Telap(msec)=     4061.6052246093750000
```

Loss of scaling when number of threads exceeds max threads per block (1024)

Performance comparison of serial vs GPU runtimes



Note, for small N, the GPU performance degrades after 10^3 but then improves for very large N.

What happens #threads is larger than #blocks*thds requested?

- You cannot exceed maxThreadsPerBlock
- Use device query to find out the max (1024 on tuckoo).
- You will loose parallel efficiency
- tuckoo.sdsu.edu (Spring 2014):
  - Max threads per block: 512 or 1024
  - Max thread dimensions: (512, 512, 64) or (1024x1024/64)
  - Max grid dimensions: (65535, 65535, 1)
- For large N, need 2D combination of threads and blocks
- Thread Rank: convert the 2D $[block, thread]$ space to a 1D indexing scheme

$$tid = threadIdx.x + blockIdx.x * blockDim.x;$$

Determining what the block & thread dimensions are on the device.

```
[mthomas@tuckoo:cuda/enum] cat /etc/motd
[snip]
GPUs
   ---------------------------------------------
   node9  has 2     GTX 480 gpu cards (1.6GB dev ram ea.)
   node8  has 2     C2075   gpu cards (  6GB dev ram ea.)
   node7  has 2     C1060   gpu cards (  4GB dev ram ea.)
   node11 has 1     K40     gpu card  (                 )
[snip]
[mthomas@tuckoo:cuda/enum] cat enum_gpu.bat
#!/bin/sh
###PBS -l nodes=node9:ppn=1
#PBS -l nodes=node7:ppn=1
#PBS -N enum_gpu
#PBS -j oe
#PBS -q batch
cd $PBS_O_WORKDIR

./enum_gpu
```

```
-----------------------------------------
NODE 7:   C1060
Name:  GeForce GT 240
   --- MP Information for device 2 ---
Multiprocessor count:  12
Shared mem per mp:  16384
Registers per mp:  16384
Threads in warp:  32
Max threads per block:  512
Max thread dimensions:  (512, 512, 64)
Max grid dimensions:  (65535, 65535, 1)
-----------------------------------------
NODE 9:  GTX 480
Name:  GeForce GTX 480
   --- MP Information for device 1 ---
Multiprocessor count:  15
Shared mem per mp:  49152
Registers per mp:  32768
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (65535, 65535, 65535)
```

Translating thread row & column locations into unique thread IDs.

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Threads represent columns, blocks represent rows.

blockDim.x	**tid=threadIdx.x + blockIdx.x*blockDim.x**	**Th0**	**Th1**	**Th2**	**Th3**
0	$0 + 0 * 4 = 0$	0	1	2	3
1	$0 + 1 * 4 = 4$	4	5	6	7
2	$0 + 2 * 4 = 8$	8	9	10	11
3	$0 + 3 * 4 = 12$	12	13	14	15

Map to a vector of Thread ID's:

Elem(B,TID) =	[0,0]	[0,1]	[0,2]	[0,3]	[1,0]	[1,1]	[1,2]	[1,3]	[2,0]	[2,1]	[2,2]	[2,3]	[3,0]	[3,1]	[3,2]	[3,3]
Vector =	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- GPU hardware limits the number of blocks per grid and the number of threads per block
- Larger problems require use of both grid and blocks
- Need to control the number of threads, since they are smaller
- Fix number of threads and distributed chunks along the blocks:

```
add<<<128,128>>>( dev_a, dev_b, dev_c);
add<<<h_N,h_N>>>( dev_a, dev_b, dev_c);
add<<<ceil(h_N/128),128>>>( dev_a, dev_b, dev_c);
add<<<(h_N+127)/128,128>>>( dev_a, dev_b, dev_c);
```

- if maxTh == maximum number of threads per block:

```
add<<<(h_N+(maxTh-1))/maxTh, maxTh>>>( dev_a, dev_b, dev_c);
```

- Compute thread index as:
  $tid = threadIdx.x + blockIdx.x * blockDim.x;$

```
$tid = threadIdx.x + blockIdx.x * blockDim.x;$
```

```
/* CODE: add_loop_gpu for large # threads.
 */
#include <stdio.h>
//#define N   65535+10
__device__ int d_Nthds;
__global__ void checkDeviceThdCount(int *t) {*t = d_Nthds;}
__global__ void add( int *a, int *b, int *c) {
   tid = threadIdx.x + blockIdx.x * blockDim.x;
   if (tid < d_Nthds)
        c[tid] = a[tid] + b[tid];
}

int main( int argc, char** argv ) {
    /* get #threads from the command line */
    int h_N = atoi(argv[1]);
    int a[h_N], b[h_N], c[h_N];
    int *dev_a, *dev_b, *dev_c;
    int i,j,k,  *d_N, d_Ntmp;
    float time;
    cudaEvent_t start, stop;

    cudaEventCreate(&start) ;
    cudaEventCreate(&stop) ;
    cudaEventRecord(start, 0) ;

    // set the number of threads to device: d_Nthds
    cudaMemcpyToSymbol(d_Nthds, &h_N, sizeof(int),
        0, cudaMemcpyHostToDevice);

    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, h_N * sizeof(int) ) ;
    cudaMalloc( (void**)&dev_b, h_N * sizeof(int) ) ;
    cudaMalloc( (void**)&dev_c, h_N * sizeof(int) ) ;
```

```
    // fill the arrays 'a' and 'b' on the CPU
    for (i=0; i<h_N; i++) {
        a[i] = i+1;     b[i] = (i+1) * (i+1);
    }

    // copy the arrays 'a' and 'b' to the GPU
     cudaMemcpy( dev_a, a, h_N * sizeof(int),
                cudaMemcpyHostToDevice ) ;
     cudaMemcpy( dev_b, b, h_N * sizeof(int),
                cudaMemcpyHostToDevice ) ;

    //add<<<128,128>>>( dev_a, dev_b, dev_c);
    //add<<<h_N,h_N>>>( dev_a, dev_b, dev_c);
    add<<<ceil(h_N/128),128>>>( dev_a, dev_b, dev_c);
    add<<<(h_N+127)/128,128>>>( dev_a, dev_b, dev_c);

    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( c, dev_c, h_N * sizeof(int),
                cudaMemcpyDeviceToHost ) ;

     // free the memory allocated on the GPU
     cudaFree( dev_a ) ; cudaFree( dev_b ) ;
     cudaFree( dev_c ) ;

    //Compute  elapsed time (in milliseconds)
    cudaEventRecord(stop, 0) ;
    cudaEventSynchronize(stop) ;
    cudaEventElapsedTime(&time, start, stop) ;
    printf("GPU Nthreads=%d, Telap(msec)= %26.16f\n",h_N,time);

    return 0;
}
```

Generalized kernel launch parameters dimGrid, dimBlock

- Distribute threads by thread blocks
- Kernel passes $<< \#blocks, \#threads >>$
- These are 3 dimensional objects, of type dim3 (C type)
- To distribute *h_N* threads, using the maximum number of threads per block, use:

```
int threadsperblock=maxthds;
blocksPerGridimin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
add<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b, dev_c);
```

OR, using the *dim3* object:

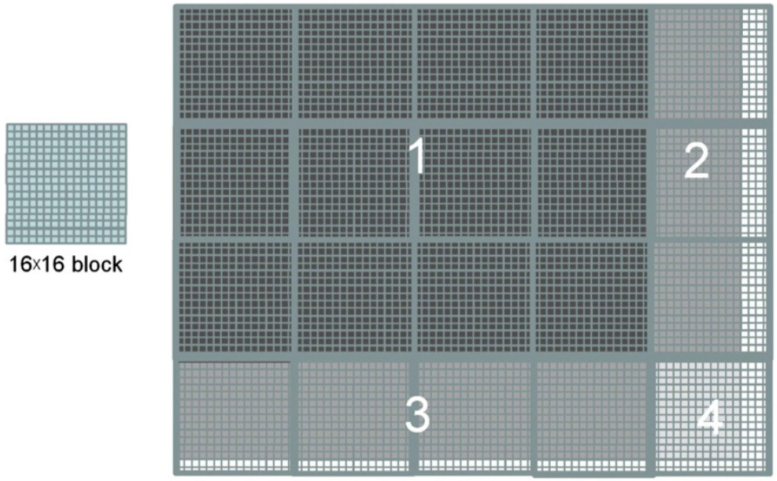
```
dim3 dimBlock(threadsperblock,1,1);
dim3 dimGrid(blocksPerGrid, 1, 1);
add<<<dimGrid, dimBlock>>>( dev_a, dev_b, dev_c);
```

- Calling a kernel for a 2D $m \, x \, n$ matrix M[m][n], where $m < maxthds$ and $n < maxblocks$

```
dim3 dimGrid(n,1,1);
dim3 dimBlock(m, 1, 1);
add<<<dimGrid, dimBlock>>>( dev_a, dev_b, dev_c);
```

Mapping threads to multidimensional data

Example:
Covering a 76x62 picture with 16x16 blocks
m =76 horiz (x)
n = 62 vert (y) pixels



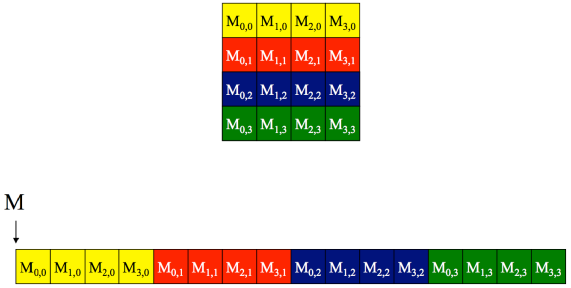

```
__global__ void kernel( unsigned char *ptr, int ticks )
{
// map from threadIdx/BlockIdx to pixel position
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int offset = x + y * blockDim.x * gridDim.x;
. . .

dim3 dimBlock(16, 16, 1);
dim3 dimGrid(ceil(n/16.0), ceil(m/16.0), 1);
pictureKernel<<<dimGrid, dimBlock>>>(d_Pin, d_Pout, n, m);
```

Row-major layout for 2D-C array

- The pixel data will have dynamic number of pixels
  - CUDA does not allow run-time allocation of a 2D matrix
  - Not allowed by the version of ANSI C used by CUDA (according to Kirk & Hu), but this may have changed by now).
- Need to linearize the array in *row − major* order, into a vector which can be dynamic.
- 1 D array, where Element[row][col] is element [row*width+col]
- Thread mapping:

int x = threadIdx.x + blockIdx.x * blockDim.x;

Memory Layout of a Matrix in C



Hybrid MPI + GPU Programming: Simple Example

```
[mthomas@tuckoo:~/pardev/cuda/sdk/samples/0_Simple/simpleMPI] cat readme.txt
Sample: simpleMPI
Minimum spec: GeForce 8

Simple example demonstrating how to use MPI in combination with CUDA.  This executable is not pre-built with the SDK installe

Key concepts:
CUDA Basic Concepts
MPI
Multithreading

===============
[mthomas@tuckoo:~/pardev/cuda/sdk/samples/0_Simple/simpleMPI] ls
total 396

-rw-------  1 mthomas mthomas    225 Feb  8  2017 readme.txt
-rw-------  1 mthomas mthomas 327884 Feb  8  2017 simpleMPI
-rw-------  1 mthomas mthomas   3272 Feb  8  2017 simpleMPI.cpp
-rw-------  1 mthomas mthomas   2518 Feb  8  2017 simpleMPI.cu
-rw-------  1 mthomas mthomas    893 Feb  8  2017 simpleMPI.h
-rw-------  1 mthomas mthomas  13756 Feb  8  2017 simpleMPI.o
```

Hybrid MPI + GPU Programming: Simple Example

```
[mthomas@tuckoo:~/pardev/cuda/sdk/samples/0_Simple/simpleMPI] cat simpleMPI.cpp
/*
 * Copyright 1993-2010 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */


/* Simple example demonstrating how to use MPI with CUDA
*
*  Generate some random numbers on one node.
*  Dispatch them to all nodes.
*  Compute their square root on each node's GPU.
*  Compute the average of the results using MPI.
*
*  simpleMPI.cpp: main program, compiled with mpicxx on linux/Mac platforms
*                 on Windows, please download the Microsoft HPC Pack SDK 2008
*/

// System includes
#include <iostream>

using std::cout;
using std::cerr;
using std::endl;

// MPI include
#include <mpi.h>

// User include
#include "simpleMPI.h"

// Error handling macros
#define MPI_CHECK(call) \
    if((call) != MPI_SUCCESS) { \
        cerr << "MPI error calling \""#call"\"\n"; \
```

Hybrid MPI + GPU Programming: Simple Example

```
[mthomas@tuckoo:~/pardev/cuda/sdk/samples/0_Simple/simpleMPI] cat simpleMPI.cu
/*
 * Copyright 1993-2010 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */


/* Simple example demonstrating how to use MPI with CUDA
*
*  Generate some random numbers on one node.
*  Dispatch them to all nodes.
*  Compute their square root on each node's GPU.
*  Compute the average of the results using MPI.
*
*  simpleMPI.cu: GPU part, compiled with nvcc
*/

#include <iostream>
using std::cerr;
using std::endl;

#include "simpleMPI.h"

// Error handling macro
#define CUDA_CHECK(call) \
    if((call) != cudaSuccess) { \
        cudaError_t err = cudaGetLastError(); \
        cerr << "CUDA error calling \""#call"\", code is " << err << endl; \
        my_abort(err); }


// Device code
// Very simple GPU Kernel that computes square roots of input numbers
__global__ void simpleMPIKernel(float *input, float *output)
{
```

```
[mthomas@tuckoo:~/pardev/cuda/sdk/samples/0_Simple/simpleMPI] cat simpleMPI.h
/*
 * Copyright 1993-2010 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

/* Simple example demonstrating how to use MPI with CUDA
*
*  Generate some random numbers on one node.
*  Dispatch them to all nodes.
*  Compute their square root on each node's GPU.
*  Compute the average of the results using MPI.
*
*  simpleMPI.h: common header file
*/

// Forward declarations
extern "C" {
    void initData(float *data, int dataSize);
    void computeGPU(float *hostData, int blockSize, int gridSize);
    float sum(float *data, int size);
    void my_abort(int err);
}
```