COMP 696: Advanced Parallel Computing
Lecture 21: Hybrid MPI + GPU Programming: GPU Overview

Mary Thomas

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

Posted: 11/10/15
Last Update: 11/10/15

Table of Contents

1. PHybrid MPI + GPU Programming: GPU Overview
2. Introduction to GPU/CUDA Computing
   - GPU/CUDA Architecture
3. GPU Architecture
   - GPU Memory Model
   - GPU/CUDA Job Execution
     - GPU/CUDA Env on tuckoo
     - GPU/CUDA Env on local host: OS X
     - GPU/CUDA Env on Student Cluster
     - Running CUDA Code on tuckoo
   - CUDA Overview
   - The Kernel
   - Passing Parameters
   - CUDA Block Parallelism
   - Block Parallelism
   - CUDA Thread Parallelism (S&K, Ch5)
   - Thread Parallelism
   - CUDA SHMEM & Syncrhronization
4. Hybrid MPI + GPU Programming: Simple Example

Reading List for GPU/CUDA Topic

- Kirk & Hwu, *Programming Massively Parallel Processors*:
  - text available online for download.
- Chapters covered:
  - Intro to Data Parallelism & CUDA, Ch3
  - Data Parallel Execution Model, Ch 4
  - CUDA Memories, Ch 5
  - Performance considerations, Ch6 (if time permits)
- Download Cuda Tutorial: Volume 1 Exercises & Instructions:
  `https://developer.nvidia.com/cuda-training/`

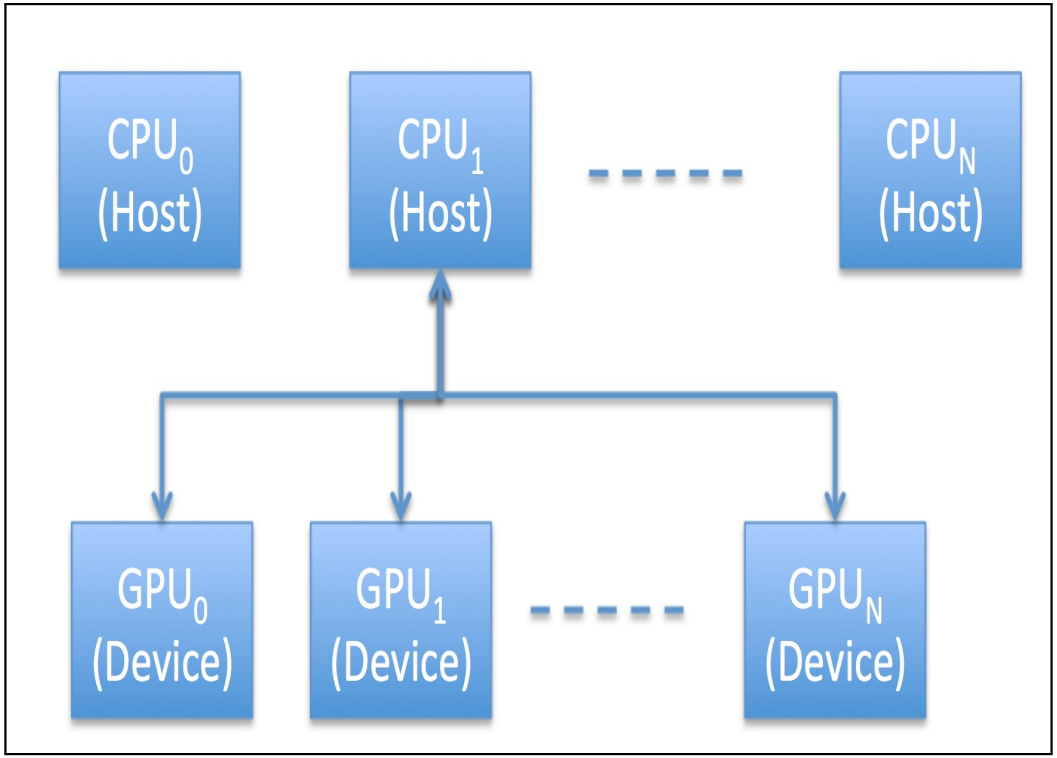
NVIDIA GPU/CUDA Refs

- Textbook: CUDA API, by Sanders & Kandrot (Chs 3, 4, 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`

Additional GPU/CUDA Reading

- 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`
- Matrix Multiplication with CUDA — A basic introduction to the CUDA programming model. Robert Hochberg, August 11, 2012

GPU Computing: Simplified Hardware View

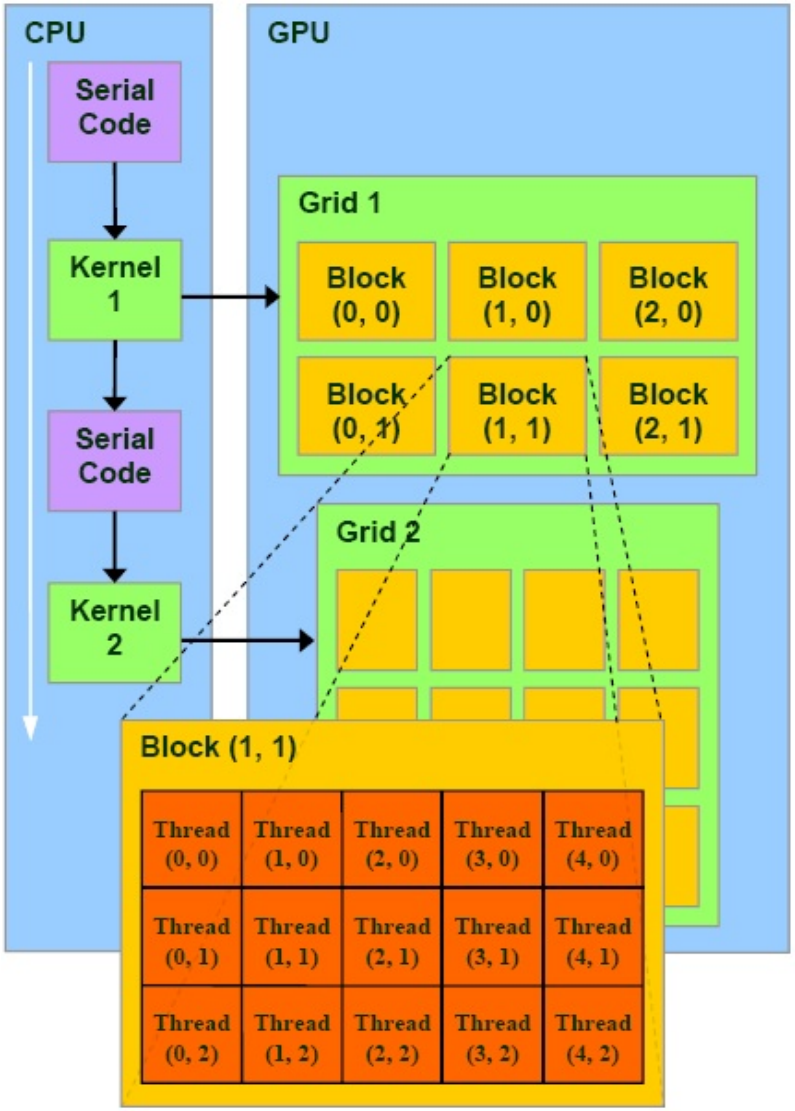


- CPU = Central Processing Units
  - single or miltiple processing units (cores)
  - standalone or integrated into clusters
  - designed to run processes, supports threads
- GPU = Graphical Processing Units
  - Usually attached to a host CPU
  - Developed for games (think Sony, PS3), and visualization (OpenGl, think Pixar)
  - Designed to run lightweight threads, may have multiple PE's
  - Accessible via specialized libraries, compiler directives (OpenACC), and extensions to languages (C, C++ and Fortran).
- CUDA (Compute Unified Device Architecture)
  - a parallel computing platform and programming model created by NVIDIA.
  - extension of C programming language

GPU Architecture

GPU Architecture

- GPU is a highly threaded coprocessor to the host CPU and associated memory
- Kernels are sections of the application that are run on the GPU by a thread.
- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
  - Each block is organized as 3D array of threads: (*blockDim.x*, *blockDim.y*, and *blockDim.z*)
- Threads within a thread block must:
  - execute the same kernel
  - share data, so they must be issued to the same processor
- A grid is a collection of blocks:
  - A Grid is organized as a 2D array of blocks: (*gridDim.x* and *gridDim.y*)



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

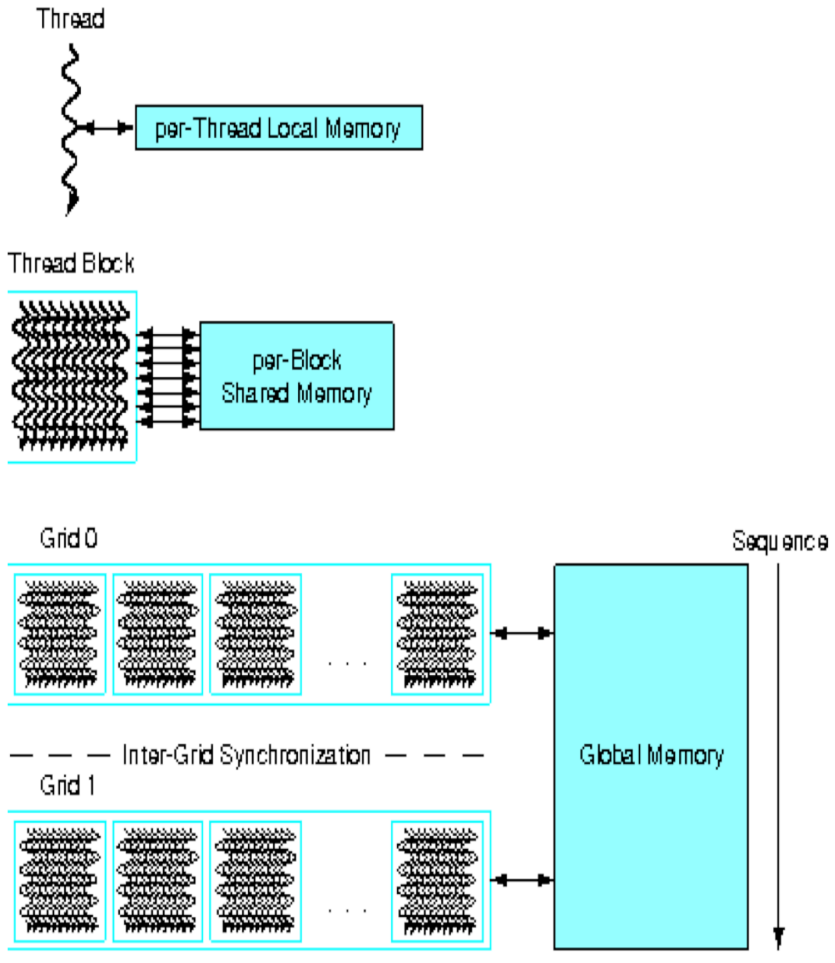
GPU: Grid

- A collection of blocks that can (but are not required to) execute in parallel.
- Theres no synchronization at all between the blocks.
- Number of [concurrent] grids on a GPU:

  1 for Cuda Cores < 2.0
  16 for $2.0 <= CC <= 3.0$
  32 for $CC = 3.5$
  . . .
  Need to use right API in order to avoid serialization.
- There is an API for querying the GPU system.

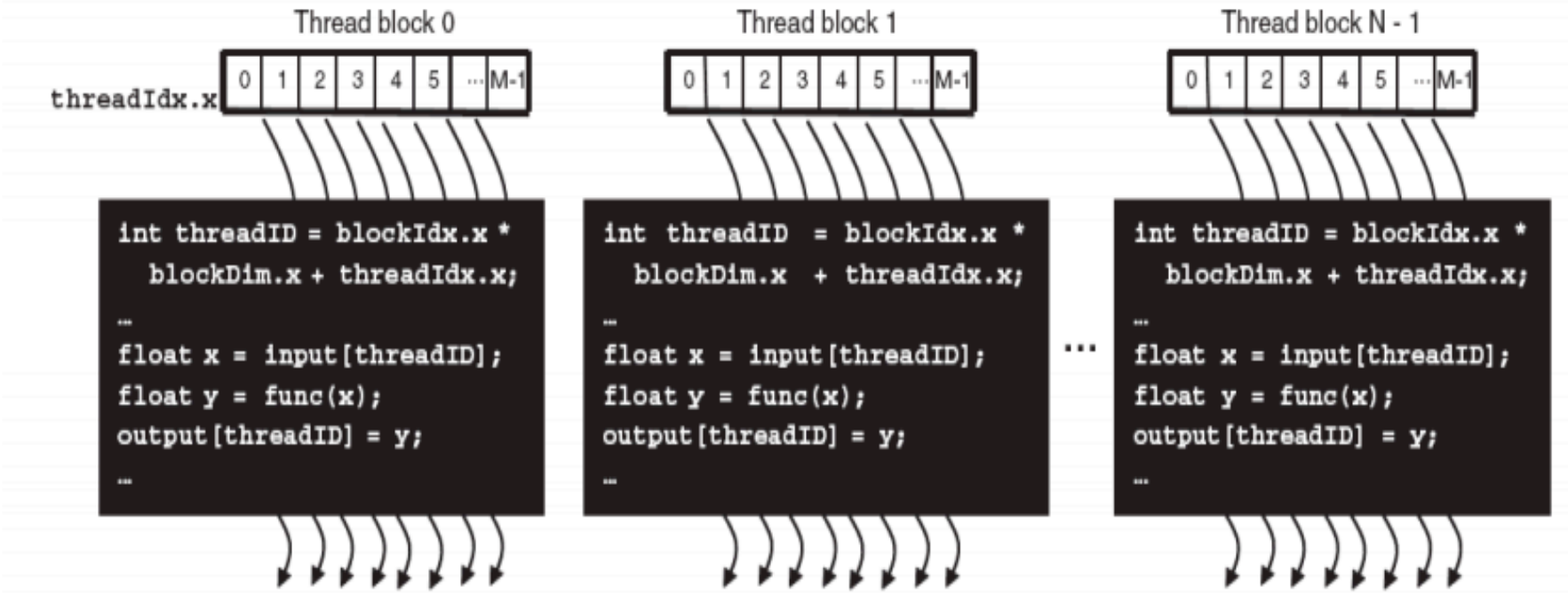
GPU: Threads

- A thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler.
- A thread is a light-weight process.
- In most cases, a thread is contained inside a process.
- Multithreading generally occurs by time-division multiplexing (as in multitasking)
- Multiprocessor (including multi-core system): threads or tasks run at the same time - each processor or core runs a particular thread or task.



Source: http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci360/lecture_notes/gpus.pdf

NVIDIA Thread Calculations



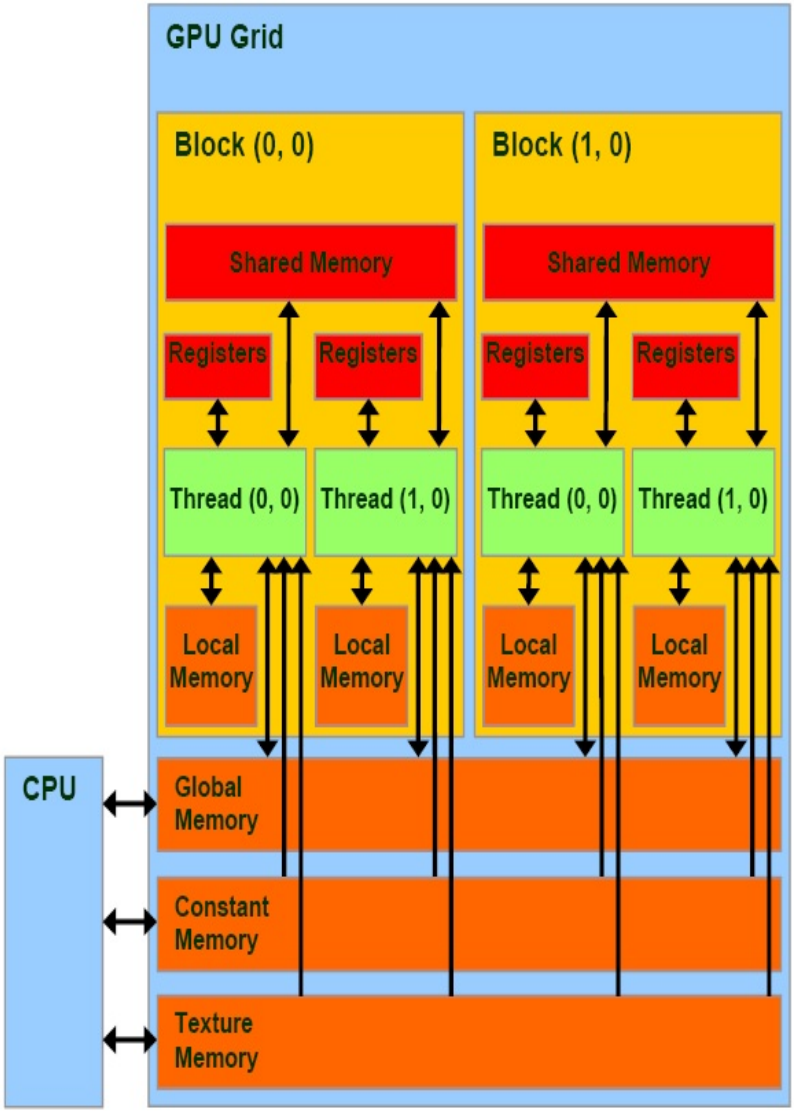
- Thread ID is unique within a block
- Bock ID is unique
- Can make unique ID for each thread per kernel using Thread and Block IDs.

GPU: Kernel

- Kernels are not full applications:
  - they are the parallel sections or critcal blocks
- They are executed by a grid of unordered thread blocks:
  - up to 512 threads per block.
  - Thread blocks start at the same instruction address, execute in parallel
  - Blocks can have different endpoints (divergence) but these are limited
  - Communicate through shared memory and synchronization barriers.
  - Must be assigned to the same processor

The CUDA Memory Model

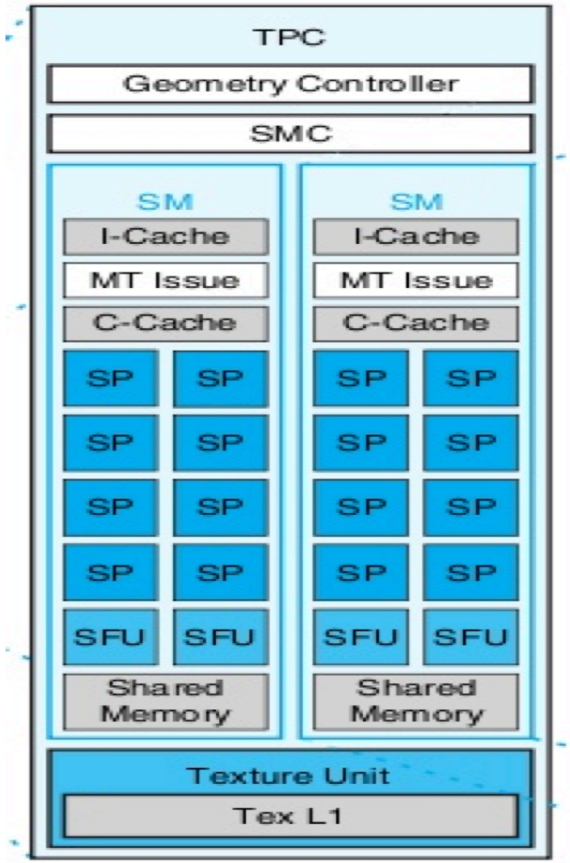
- Red is fast on-chip, orange is DRAM
- Register file & local memory are private for each thread
- Shared memory is used for communication between threads (appx same latency as regs)
- DRAM, Readonly:
  - Constant memory (64KB) used for random accesses (such as instructions)
  - Texture memory (large) and has two dimensional locality
- Global Memory: visible to an entire grid, can be arbitrarily written to and read from by the GPU or the CPU.



Source: NVIDIA

NVIDIA Hardware: GEForce 8800

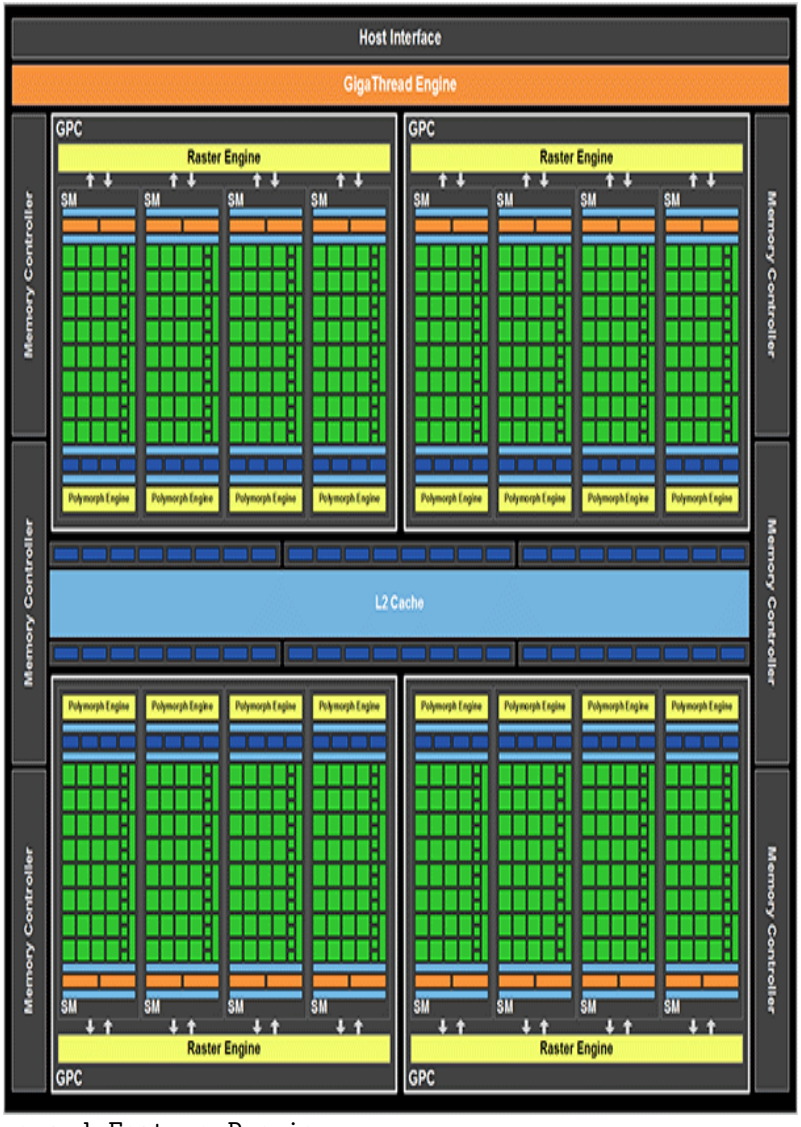
- Streaming Multiprocessors (SMs, also called nodes)
- 8 Stream Processors (SPs) (or cores): primary thread processor
- has 1000's of registers that can be partitioned among threads of execution
- Multiple caches:
  - shared memory for fast data interchange between threads,
  - constant cache for fast broadcast of reads from constant memory,
  - texture cache to aggregate bandwidth from texture memory,
  - L1 cache: reduce latency to memory
- warp schedulers: switch contexts between threads and instructions to warps;
- Execution cores:
  - Integer and floating point ops
  - Special Function Units (SFUs)



Source: http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci360/lecture_notes/gpus.pdf

NVIDIA GPU GF100 High-Level Block Diagram (2010)

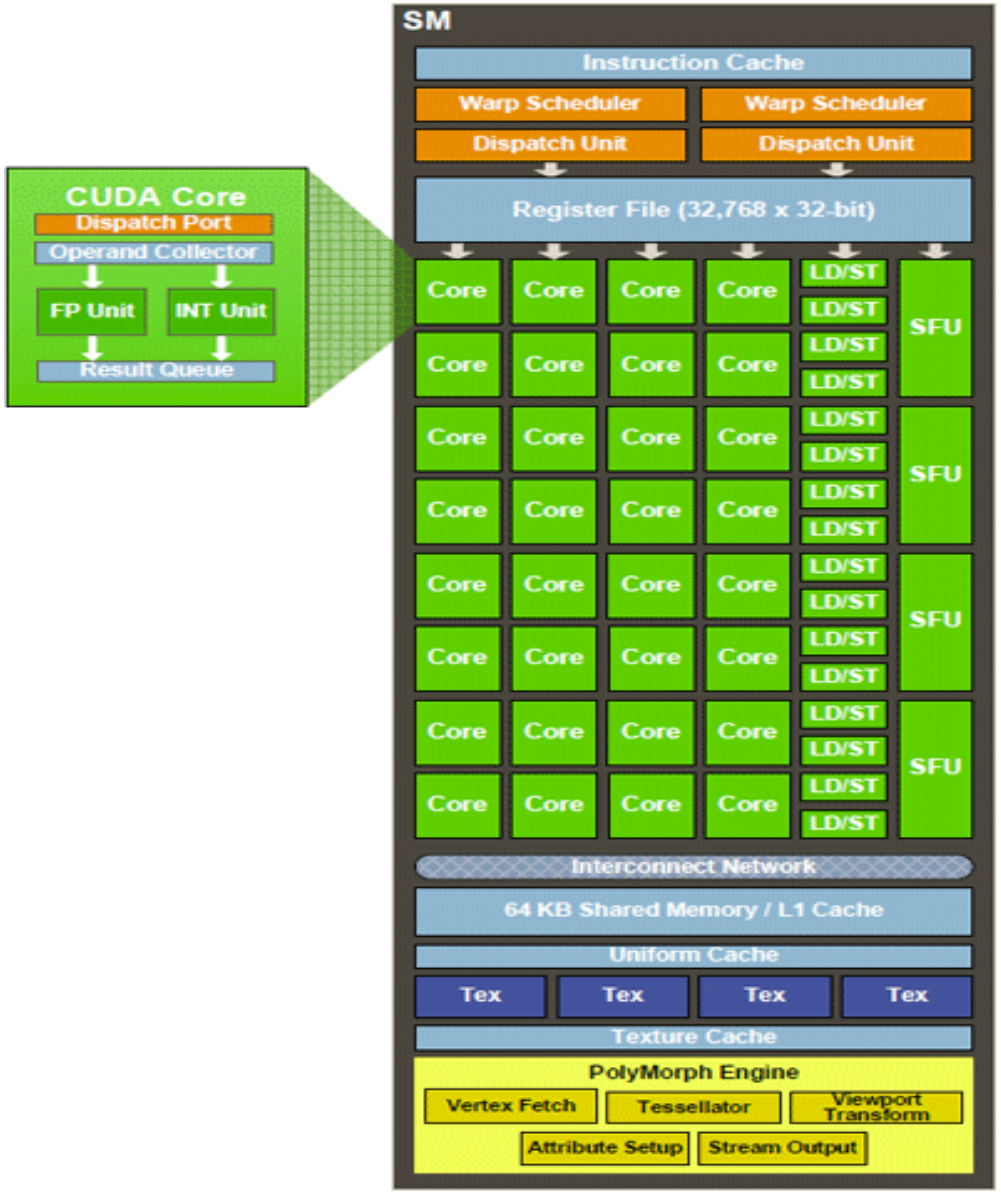
- CPU is the host
- GPU is the device
- The GF100 has 4 Graphics Processing Clusters (GPCs): laid out in (2x2) arrangement (also called "Raster Engine").
- Each GPC has 4 Streaming Multiprocessors, (SMs): NVIDIAs' term for multiprocessor (also called "Polymorph Engines").
  Arranged in 1x4 layout.
  Total number of SMs $= 4 * 4 = 16$
- Each SM has a block of Stream Processors (SPs) or Cores– also called execution units.
  Arranged in 8x4 layout.
  Total number of SPs on each SM $= 8 * 4 = 32$
- Total number of cores on the GPU
  $\#Cores = \#SPs/SM \times \#SMs/GPC \times \#GPCs$
  $= 32 \times 4 \times 4 = 512$



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

NVIDIA GF100 SM Block Diagram

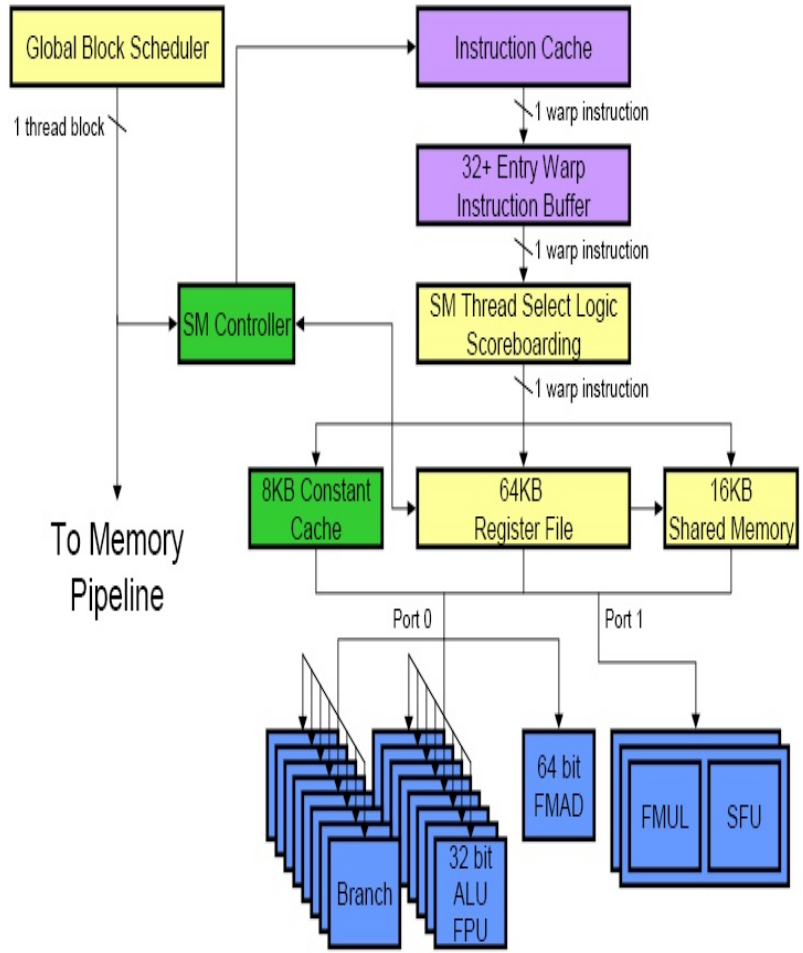
- each SM block in each GPC is comprised of 32 cores
- 48/16KB of shared memory (3 x that of GT200),
- 16/48KB of L1 (there is no L1 cache on GT200),



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

NVIDIA GT200 SM Arch (2008)

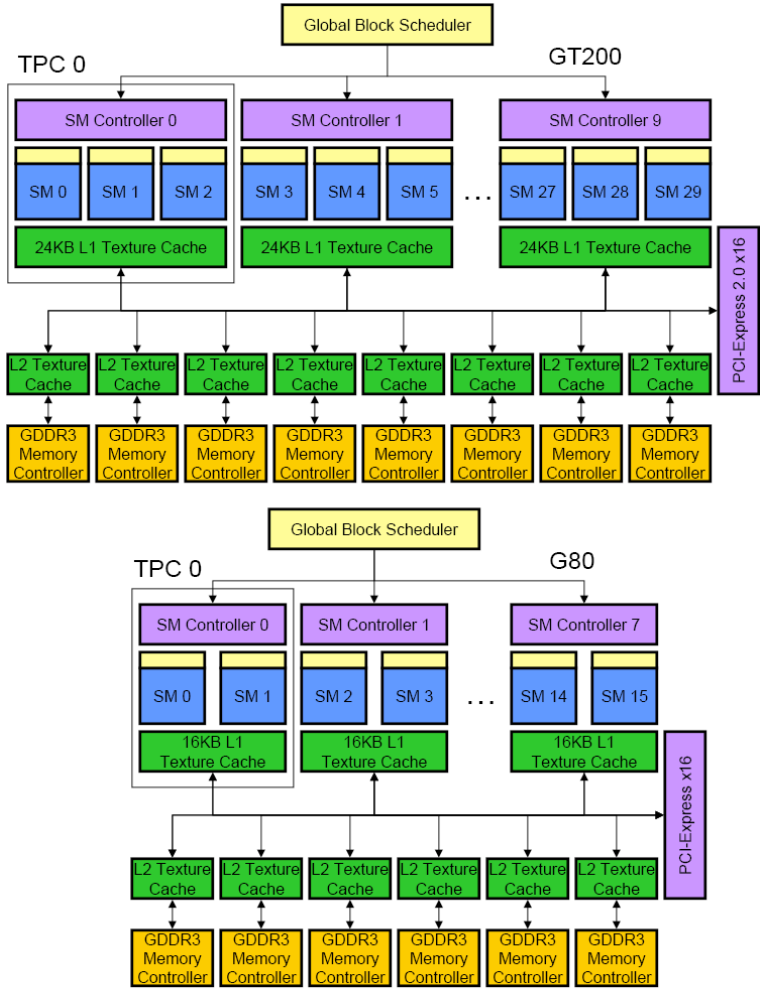
- highly threaded single-issue processor with SIMD/SIMT (single instruction multiple thread)
- 8 functional units
- Each SM can execute up to 8 thread blocks concurrently and a total of 1024 threads concurrently
- warp: a group of threads managed by SM thread scheduler
- Single Instruction, Multiple Thread (SIMT) programming model



Source: `http://www.realworldtech.com/gt200`

GPU Global Scheduler (work distribution unit)

- Manages coarse grained parallelism at thread block level
- At kernel startup, information for grid sent from CPU (host) to GPU (device)
- Scheduler reads information and issues thread blocks to streaming multiprocessors (SM)
- Issues thread blocks in a round-robin fashion to SMs
- Uniformly distribute threads to SMs
- Key distribution factors:
  - kernel demand for threads per block
  - shared memory per block
  - registers per thread
  - thread and block state requirements
  - current availability resources in SM

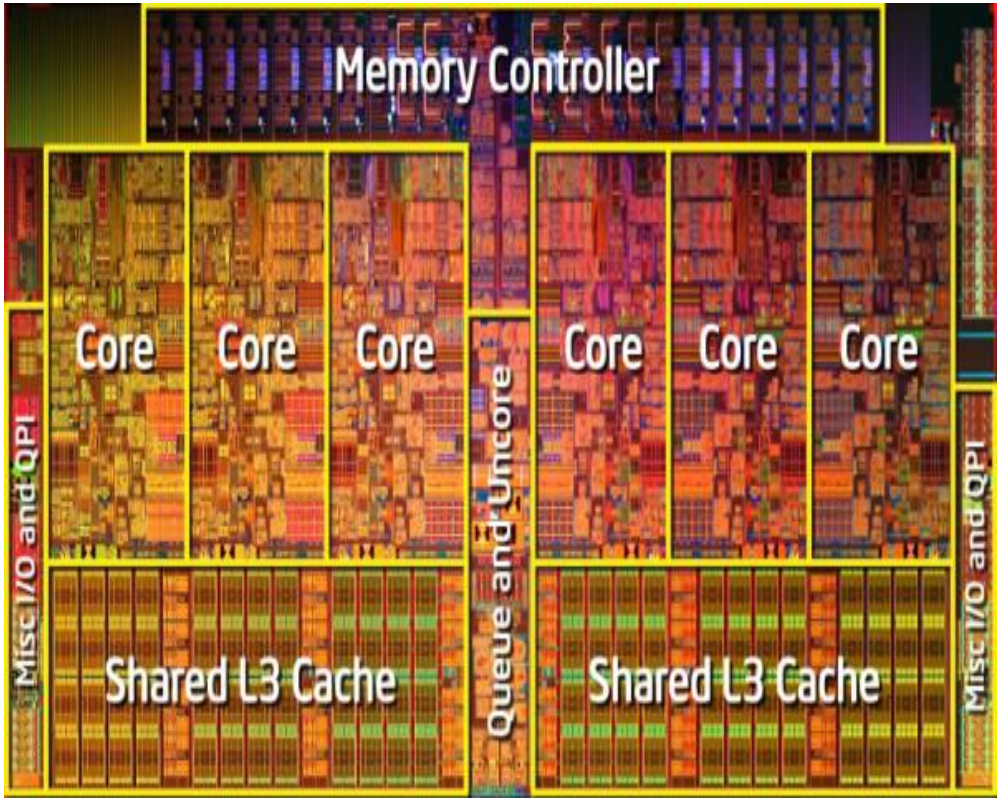


http://www.realworldtech.com/gt200/6/

A CPU/GPU Cluster: tuckoo.sdsu.edu

```
-------------------------------------------
 Welcome to baby (tuckoo) -- a student cluster
 -------------------------------------------
 the cluster system has 11 nodes with various CPUs:

     node1 thru node5 --  8 cores each -- node property= core8
     node6 thru node9 --  4 cores each -- node property= core4
     csrc-gpu (node10)-- 12 cores     -- node properties= core16 and gpu
     csrc-gpu2(node11)--  8 cores     -- node properties= gpu2

 CPUs & RAM
 -------------------------------------------
 node1 thru node4, Xeon E5405 @ 2.00GHz, node1=14GB, node2-node4=12GB
 node5             Xeon E5420 @ 2.50GHz, 20GB
 node6 thru node9, Xeon X3360 @ 2.83GHz, 8GB
 csrc-gpu          Xeon X5650 @ 2.67GHz, 48GB
 csrc-gpu2         Xeon E5620 @ 2.40GHz, 48GB

 csrc-gpu  has 2 Tesla C1060 gpu cards
 csrc-gpu2 has 2 Tesla C2075 gpu cards
```

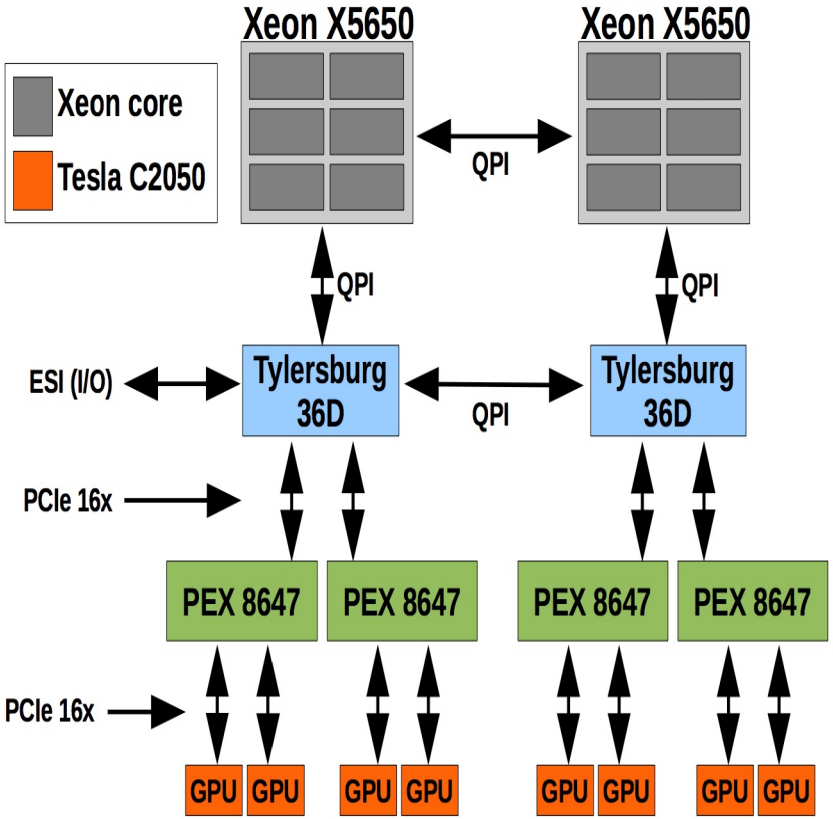
- Intel Xeon X5650 system contains six CPUs (Xeon 5650)
- QPI-PCIe bridge;
- PCI-e switch for GPUs.



Source: www

System with 2 Intel Xeon X5650 and 8 Nvidia GPU Teslas

- Intel Xeon X5650 system
- Two six-core CPUs (Xeon 5650)
- eight GPUs
- Tylesburg-36D, QPI-PCIe bridge
- PXE8647 PCI-e switch for GPU pairs.



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

Table: Table of tuckoo CPU/GPU configurations (2014)

Property	**csrc-gpu**	**csrc-gpu2**	**csrc-gpu3**
node ID	10	11	12
CPU Type	2 Xeon X5650	2 Xeon X5650	2 Xeon E5620
#CPU cores	6*2=12	4*2=8	3*2=6
GPU Type	**2 Tesla C1060**	**2 Tesla C2075**	**2 Tesla C2075**
Multiprocessor (MP)	30	14	14
#SP/Cores	240	448	448
#GigaFLOPS		515	
Max Thd/Block	512	1024	1024
Max Grd Dim	$64k \times 64k \times 1$	$64k \times 64k \times 64$	$64k \times 64k \times 64$

GPU Performance Example: GEForce 8800 GTX

- Single-precision multiply-add ops
- Ops take place in a single instruction cycle.
- Peak Perf (all PEs busy all the time):

$$\begin{aligned} Perf_{peak} &= 16 SMs \times (8 cores/SM) \\ &\times (2 FLOPS/op/core) \\ &\times (1 \quad instruction/\text{clock cycle}) \\ &\times (1.35x10^9 \text{ clock cycles}/sec) \\ &= 345.6 \text{ GFLOPs / second} \end{aligned}$$

- With same clock speed, for the Kepler GK110, $Perf_{peak}$ 8 *TeraFlops*

Source: http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci360/lecture_notes/gpus.pdf

GPU Bandwidth Example: GEForce 8800 GTX

- Each GEForce SM has a local store of 16KB, & 8192 32-bit registers.
- Shared memory partitions:
  - 6 DDR3 DRAM (900 MHz)
  - 8-byte wide data path
  - 128 MB per DDRAM partition.
  - $Mem_{tot} = 768$ MB
- Peak bandwidth (double-data rate):

$$\begin{aligned} BW_{peak} &= 6x8\, bytes/transfer \times 2\, transfers/clockcycle \\ &\quad \times 0.9x10^9\, clockcycles/second \\ &= 6 \times 8 \times 2 \times 0.9 GB/second \\ &= 86.4 GB/second \end{aligned}$$

Source: http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci360/lecture_notes/gpus.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
```

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

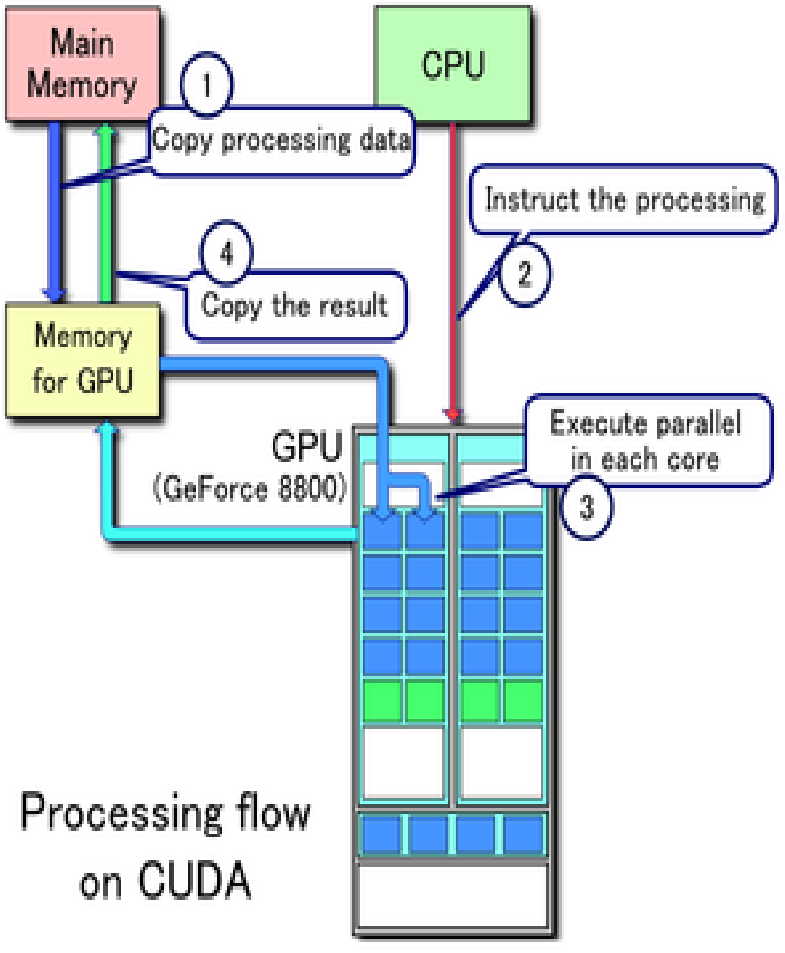
Outline:

- 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



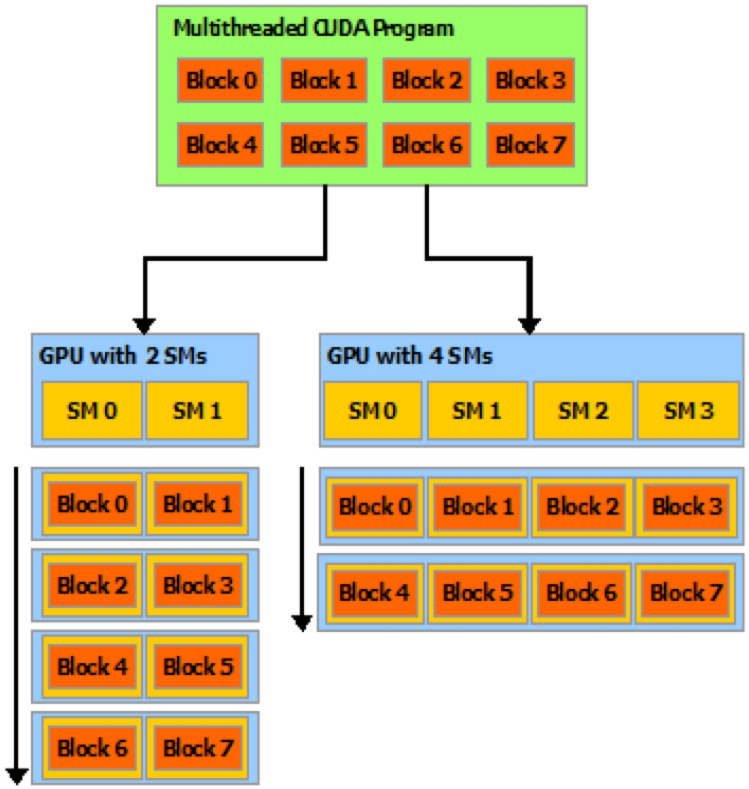
Source: http://en.wikipedia.org/wiki/CUDA

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

CUDA code highlights:

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

```
[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 API: Kernel

In its simplest form it looks like:

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

Kernel runs on SMs. 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

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*

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:

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

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 hose
  - Callable from host only

Source:

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

Memory Allocation

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

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;
   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 );

   cudaFree( dev_c );

   return 0;
}
```

```
[mthomas@tuckoo chapter03]$ cat simple_device_call.o69555
-----------------------------------------------------
Job is running on node node7
-----------------------------------------------------
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_b
PBS: execution mode is PBS_BATCH
PBS: job identifier is 69555.tuckoo.sdsu.edu
PBS: job name is simple_device_call
PBS: node file is /var/spool/torque/aux//69555.tuckoo.sdsu
PBS: current home directory is /home/mthomas
-----------------------------------------------------
simple_device_call using 1 cores...
2 + 7 = 9
```

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

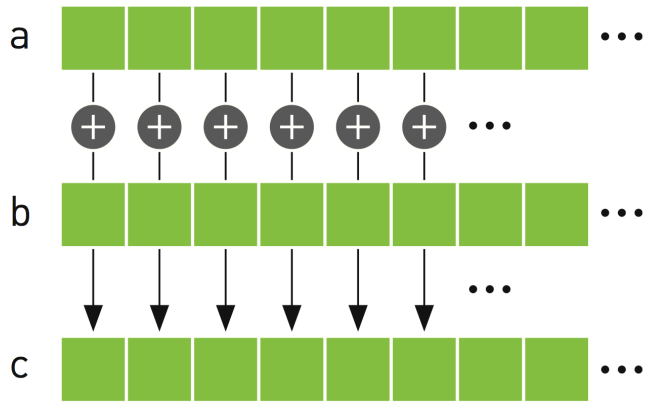
CPU

```
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



GPU block assignment for add kernel for $N = 4$ blocks.

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];
}
```

Source: *Cuda By Example*

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*

Source code for: *add_loop_gpu.cu*

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

#define N   10

__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    //  <--- cuda variable
        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 N blocks, 1 thread per block

add <<< N, 1 >>> (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;
}
```

Batch Script for Running CUDA code.

```
#!/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:node7
#PBS -N add_loop_gpu
#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 ------------------------------------------------------

./add_loop_gpu  $NTHDS
```

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

```
[mthomas@tuckoo chapter04]$qsub -v NTHDS=10 bat.addloop
20199.tuckoo.sdsu.edu
[mthomas@tuckoo chapter04]$
[mthomas@tuckoo chapter04]$ cat addloop.o20199[tuckoo]$ ./add_loop_gpu
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
GPU Nthreads=10, Telap(msec)=         0.5953919887542725
```

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

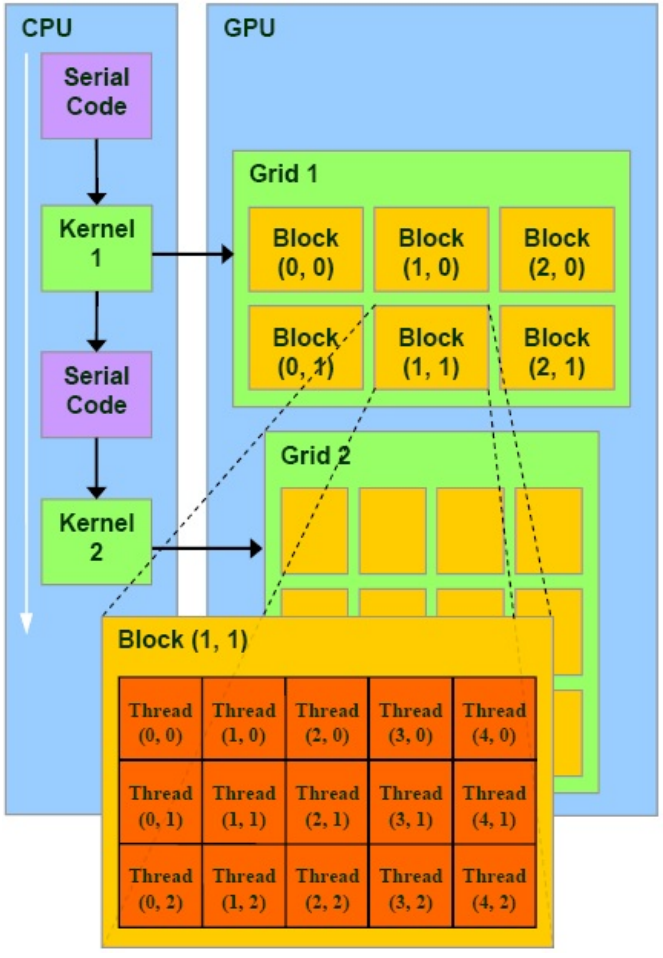
```
#include <stdio.h>
__global__ void hello_kernel(float * x) {
// By Ingemar Ragnemalm 2010
#include <stdio.h>

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;
```

CUDA Thread Parallelism (K&H Ch4, S&K, Ch5)

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_blocks.cu

```
#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 N blocks, 1 thread 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: setNthdsFromCmdArg.cu

```
/*
 * CODE: setNthdsFromCmdArg.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.
  */

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

Thread Parallelism: setNthdsFromCmdArg.cu

```
int main( int argc, char** argv ) {
    //get number of 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;
    float time;
    cudaEvent_t start, stop;

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

     // set the number of threads to shared 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();

    // 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);
    }
```

Thread Parallelism: setNthdsFromCmdArg.cu

```
// 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 ) ;

// display the results
if(h_N < 10 ) {
   for (i=0; i<h_N; i++) {
       printf( "%d + %d = %d\n", a[i], b[i], c[i] );
   }
}

// 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("Nthreads=%ld, Telapsed(msec)= %26.16f\n",h_N,time);

return 0;
```

Thread Parallelism: add_loop_blocks.cu (output)

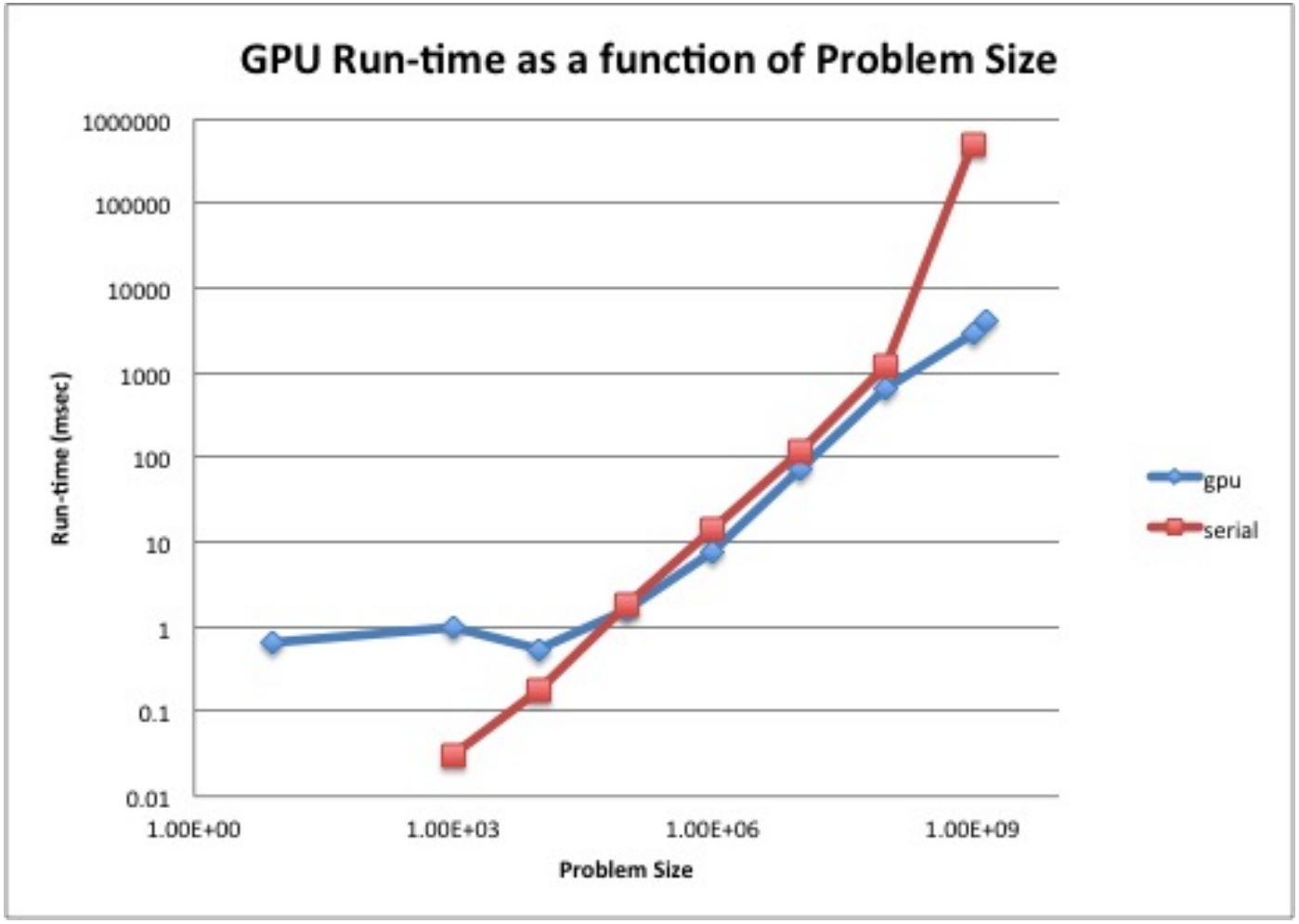
```
mthomas@tuckoo chapter04]$ cat addloop.o72627
running add_loop_gpu using 8 threads
There are 2 CUDA devices.
CUDA Device #0
Device Name: GeForce GTX 480
Maximum threads per block:     1024
Maximum dimensions of block: blockDim[0,1,2]=[ 1024  1024  64 ]
CUDA Device #1
Device Name: GeForce GTX 480
Maximum threads per block:     1024
Maximum dimensions of block: blockDim[0,1,2]=[ 1024  1024  64 ]
h_N = 8, h_Ndevice=8
1 + 1 = 2
2 + 4 = 6
3 + 9 = 12
4 + 16 = 20
5 + 25 = 30
6 + 36 = 42
7 + 49 = 56
8 + 64 = 72
Nthreads=8, Telapsed(msec)=         0.6642879843711853
```

Thread Parallelism: add_loop_blocks.cu (output)

```
[mthomas@tuckoo chapter04]$ 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

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
ls *.8
```

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

Performance comparison of serial vs GPU runtimes

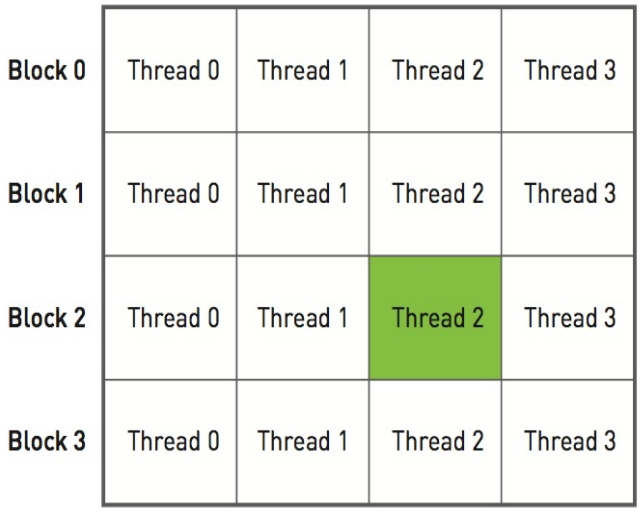


Note, for small N, the GPU performance degrades after 10^3 but then improves for very large N.

Thread Parallelism – longer vectors

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

- You will loose parallel efficiency
- Use device query to find out the max (1024 on tuckoo).
- tuckoo.sdsu.edu (Spring 2014):
  - Max threads per block: 512/1024
  - Max thread dimensions: (512, 512, 64)/(1024x1024/64)
  - Max grid dimensions: (65535, 65535, 1)
  - What is Max Array dimension?
- For large N, need combination of threads and blocks
- Convert from a 2D [*block*, *thread*] space to a 1D indexing scheme
  $$tid = threadIdx.x + blockIdx.x * blockDim.x;$$
- *blockDim* is constant, stores number of threads along each dimension
- similar to *gridDim* which stores number of blocks along each dimension



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

```
Vector  V=  [0]    [1]    [2]    [3]    [4]    [5]    [6]    [7]    [8]    [9]    [10]   [11]   [12]   [13]   [14]   [1
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]
```

```
__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;
    }
}
```

This looks remarkably like our *original* version of vector addition! In fact, compare it to the following CPU implementation from the previous chapter:

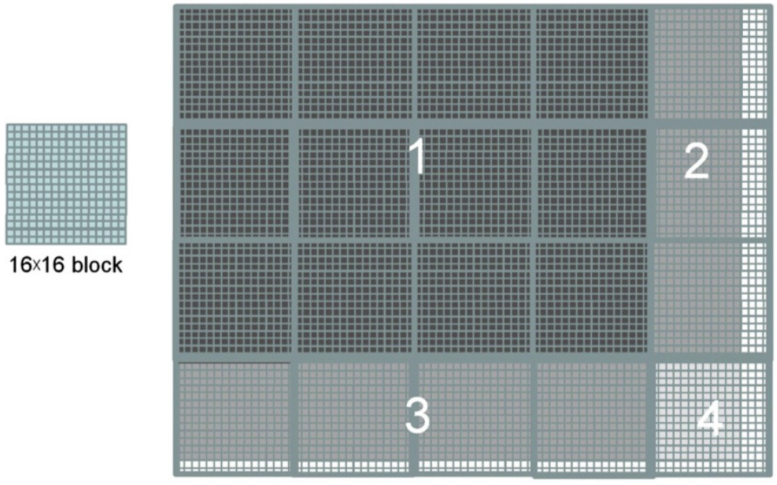
```
void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}
```

Distribute threads using grid dimension as a stride, until all are gone (while loop)

add_loop_long_blocks.cu (K&S, Ch5)

Note that N > 128*128, but code will run

```
#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;
}
```



Example: Covering a 76x62 picture with 16x16 blocks

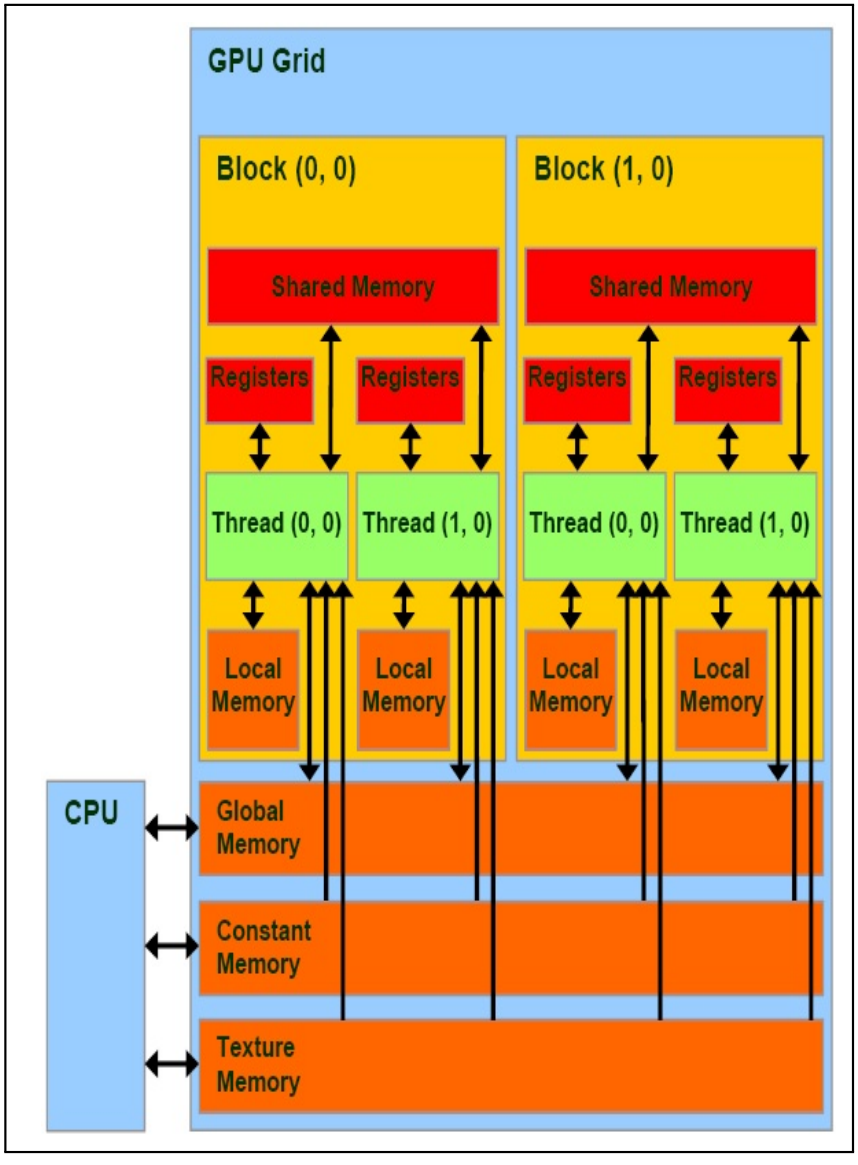
76 horiz (x) x 62 verti (y) pixels

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

The CUDA Memory Model

- The *kernel* is executed by a batch of threads
- Threads are organized into a *grid* of thread *blocks*.
- Each thread has its own registers, no other thread can access it
- the *kernel* uses registers to store private thread data
- *Shared memory*: allocated to thread blocks - promotes *thread cooperation*
- *global memory*: host/threads can read/write
- *constan and texture memory*: host/threads read only

- threads in same block can share memory
- requires synchronization – essentially communication
- Example: Dot product:
  $(x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) =$
  $x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$



Source: NVIDIA

dot.c

```
#include "../common/book.h"
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
            imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
int main( void ) {
    float   *a, *b, c, *partial_c;
    float   *dev_a, *dev_b, *dev_partial_c;
    // allocate memory on the cpu side
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                              N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                              N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                              blocksPerGrid*sizeof(float) ) );
    // fill in the host memory with data
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;     }

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

dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
                                        dev_partial_c );

// copy the array 'c' back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                          blocksPerGrid*sizeof(float),
                          cudaMemcpyDeviceToHost ) );

// finish up on the CPU side
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}

#define sum_squares(x)  (x*(x+1)*(2*x+1)/6)
printf( "Does GPU value %.6g = %.6g?\n", c,
         2 * sum_squares( (float)(N - 1) ) );

// free memory on the gpu side
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_partial_c ) );

// free memory on the cpu side
free( a );
free( b );
free( partial_c );
```

Shared Memory Model: dot.cu (S&K Ch5)

```
__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];  // buffer of shared memory - store sum
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float   temp = 0;   // each thread computes running sum of product
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
       cache[cacheIndex] = temp;    // set the cache values in the shared buffer


      __syncthreads(); // synchronize threads in this BLOCK


    // for reductions, threadsPerBlock must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }
    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```

Reduction Operation

```
// for reductions, threadsPerBlock must be a power of 2
  //
  int i = blockDim.x/2;
  while (i != 0) {
    if (cacheIndex < i)
       cache[cacheIndex]+=cache[cacheIndex + i];
     __syncthreads();
     i /= 2;
  }
  if (cacheIndex == 0)  // only need one thread to write t
     c[blockIdx.x] = cache[0];
}
```

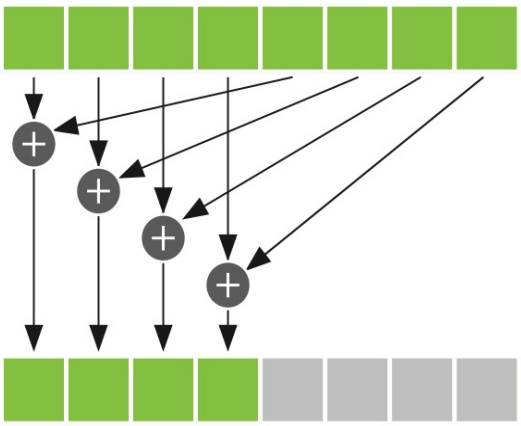
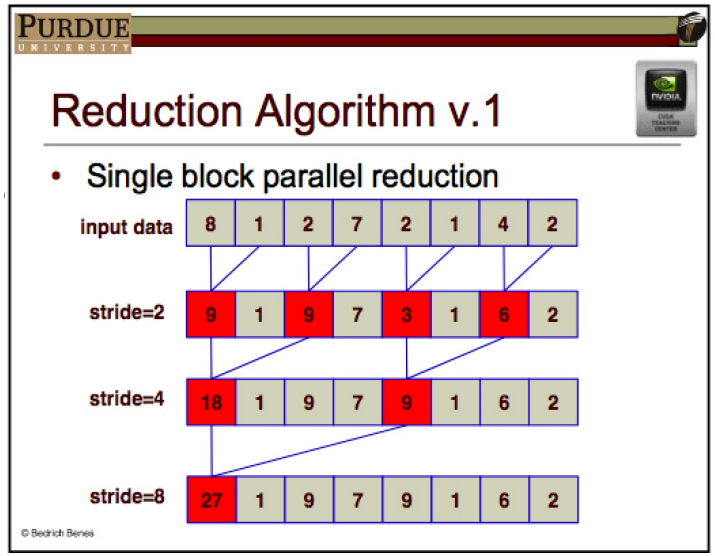


Figure 5.4 One step of a summation reduction



Source: NVIDIA

Calculating smallest multiple of threads per block greater than N

```
const int blocksPerGrid =  imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
```

- more efficient (reduce wasted threads)
- should be either 32 or the number calc above

dot.cu (part 1)

```
#include "../common/book.h"
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =  imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );

__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
    float   temp = 0;
    while (tid < N)    {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;            }
    cache[cacheIndex] = temp;       // set the cache values
    __syncthreads();       // synchronize threads in this block

    // for reductions, threadsPerBlock must be a power of 2 because of the following code
    int i = blockDim.x/2;
    while (i != 0)     {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;      }
    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```

dot.cu (part 2)

```
int main( void ) {
    float   *a, *b, c, *partial_c;
    float   *dev_a, *dev_b, *dev_partial_c;

    // allocate memory on the cpu side
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,blocksPerGrid*sizeof(float) ) );

    // fill in the host memory with data
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;             }

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

    dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,  dev_partial_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR(cudaMemcpy(partial_c, dev_partial_c, blocksPerGrid*sizeof(float), cudaMemcpyDeviceToHost));
```

dot.cu (part 3)

```
    // finish up on the CPU side
    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i];
    }

    #define sum_squares(x)  (x*(x+1)*(2*x+1)/6)
    printf( "Does GPU value %.6g = %.6g?\n", c,
             2 * sum_squares( (float)(N - 1) ) );

    // free memory on the gpu side
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_partial_c ) );

    // free memory on the cpu side
    free( a );
    free( b );
}
```

```
/* Function:    OMP_CalcPI  */
double OMP_CalcPI(double a, double b, int n) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();
   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;

   my_result = CalcPI(local_a, local_b, local_n, h);

   return my_result;
}  /* OMP_CalcPI */

/*--------------------------------------------------------------------
double CalcPI( double left_endpt  /* in */,
          double right_endpt /* in */,
          int    trap_count  /* in */,
          double base_len    /* in */) {
     double estimate, x;
     int i;
    estimate = (f_pi(left_endpt) + f_pi(right_endpt))/2.0;
   for (i = 1; i <= trap_count-1; i++) {
        x = left_endpt + i*base_len;
       estimate += f_pi(x);
   }
   estimate = estimate*base_len;
   return estimate;
} /*  CalcPI  */

/*--------------------------------------------------------------------
 * Function:    f_pi   */
double f_pi(double x /* in */) {
   return 4.0/(1+x*x);
} /* f_pi */
```