# Table of Contents

1. **GPU Architecture**
2. **Introduction to CUDA (S&K, Ch3)**
   - The Kernel
   - Passing Parameters
   - Getting Device Information
3. **CUDA Parallelism (S&K, Ch4)**
4. **Block Parallelism**
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:
  - CUDA SDK:
  - CUDA example files on tuckoo in /COMP605/cuda

- GPU Architectures:
  - References: NVIDIA online documents
    and lecture notes by S.Weiss

- More relevant reading
  - Matrix Multiplication with CUDA — A basic introduction to the CUDA programming model. Robert Hochberg, August 11, 2012
GPU Architecture
GPU Architecture

- GPU is a highly threaded coprocessor to the host CPU and associated memory.
- GPU useful for extremely data parallel workloads, where similar calculations are run on vast quantities of data arrayed in a regular grid-like fashion.
- **Kernels** are sections of the application that run on the GPU.
- A **grid** is a collection of **thread blocks**.
  - A grid is organized as a 2D array of blocks: \((\text{gridDim}.x \text{ and gridDim}.y)\)
  - Each block is organized as a 3D array of threads \((\text{blockDim}.x, \text{blockDim}.y, \text{and blockDim}.z)\)
- Threads composing a thread block must:
  - execute the same kernel
  - share data, so they must be issued to the same processor.

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

NVIDIA Thread Calculations

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

- A collection of thread blocks that can (but are not required to) execute in parallel.

- Number of [concurrent] grids on a GPU:
  1. 1 for Cuda Cores $< 2.0$
  2. 16 for $2.0 \leq CC \leq 3.0$
  3. 32 for $CC = 3.5$

but use right API in order to avoid serialization.
GPU: Kernel

- Kernels are not full applications, they are the parallel sections or critical blocks.
- They are executed by a grid of unordered thread blocks (up to 512 threads).
- 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.
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)

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
GPU Architecture

NVIDIA GPU GF100 High-Level Block Diagram (2010)

- CPU is the host; GPU is the device
- 4 graphics processing clusters (GPC)
- GPC has 4 SM (Streaming Multiprocessors, NVIDIA term for multiprocessor)
- Each SM has 512 stream processors (SPs) or CORES—also called execution units.

Source: http://hothardware.com/Articles/NVIDIA-GF100-Architecture-and-Feature-Preview
each SM core 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 block managed by SM thread scheduler
- Single Instruction, Multiple Thread (SIMT) programming model

Source: http://www.realworldtech.com/gt200
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):
  \[ \text{Perf}_{\text{peak}} = 16 \text{ SMs} \times (8\text{cores}/\text{SM}) \times (2\text{FLOPS/op/core}) \times (1\text{ instruction/clock cycle}) \times (1.35 \times 10^9 \text{ clock cycles/sec}) \]
  \[ = 345.6 \text{ GFLOPs/second} \]
- With same clock speed, for the Kepler GK110,
  \[ \text{Perf}_{\text{peak}} = 8 \text{TeraFlops} \]
- 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.
  - \( \text{Mem}_{\text{tot}} = 768 \text{ MB} \)
- Peak bandwidth (double-data rate):
  \[ \text{BW}_{\text{peak}} = 6 \times 8 \text{ bytes/transfer} \times 2 \text{ transfers/clock cycle} \times 0.9 \times 10^9 \text{ clock cycles/second} \]
  \[ = 6 \times 8 \times 2 \times 0.9 \text{ GB/second} \]
  \[ = 86.4 \text{ GB/second} \]

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-PCle 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

<table>
<thead>
<tr>
<th>Property</th>
<th>csrc-gpu</th>
<th>csrc-gpu2</th>
<th>csrc-gpu3</th>
</tr>
</thead>
<tbody>
<tr>
<td>node ID</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>CPU Type</td>
<td>2 Xeon X5650</td>
<td>2 Xeon X5650</td>
<td>2 Xeon E5620</td>
</tr>
<tr>
<td>#CPU cores</td>
<td>6*2=12</td>
<td>4*2=8</td>
<td>3*2=6</td>
</tr>
<tr>
<td>GPU Type</td>
<td>2 Tesla C1060</td>
<td>2 Tesla C2075</td>
<td>2 Tesla C2075</td>
</tr>
<tr>
<td>Multiprocessor (MP)</td>
<td>30</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>#SP/Cores</td>
<td>240</td>
<td>448</td>
<td>448</td>
</tr>
<tr>
<td>#GigaFLOPS</td>
<td>512</td>
<td>515</td>
<td>515</td>
</tr>
<tr>
<td>Max Thd/Block</td>
<td>64k x 64k x 1</td>
<td>64k x 64k x 64</td>
<td>64k x 64k x 64</td>
</tr>
<tr>
<td>Max Grd Dim</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Introduction to Compute Unified Device Architecture (CUDA, 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 memories
  - 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 Multipros (SMs)
- Multithreaded program partitioned into blocks of threads that execute independently from each other

Source: NVIDIA cuda-c-programming-guide
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

```c
#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 <<< gridSize, 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 (rows):
  - \( \text{gridDim} \) == number of blocks along grid dims: \([\text{gridDim}.x, \text{gridDim}.y]\)
  - \( \text{blockIdx}.x \) is the index of the block along \( x \) axis of the grid

- A **Block** is a collection of threads (columns):
  - \( \text{blockDim} \) == size or number of threads along dims of the block
  - \( \text{threadIdx}.x \) is the index of the thread along the \( x \) axis of the block

- **Threads** execute the *kernel* code on *device*:
Two types of parallelism:

**Block Parallelism:**
Launch N blocks with 1 thread each:

```
add <<< N, 1 >>> (dev_a, dev_b, dev_c) >>>
```

**Thread Parallelism**
Launch 1 block with N threads:

```
add <<< 1, N >>> (dev_a, dev_b, dev_c) >>>
```
Function type qualifiers

- **__device__**
  - Executed on GPU
  - Launched on GPU

- **__host__** (optional)
  - Executed on CPU
  - Launched on CPU

- **__global__**
  - Executed on GPU
  - Launched on CPU
Memory Allocation

- CPU: malloc, calloc, free, cudaMallocHost, cudaFreeHost
- GPU: cudaMalloc, cudaMallocPitch, cudaFree, cudaMallocArray, cudaFreeArray
simple_kernel_params.cu (part 1)

```c
#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

2 + 7 = 9
obtaining device information: enum_gpu.cu (1) from K&S, Ch 3

#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)

```c
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" );
}
```
Introduction to CUDA (S&K, Ch3)

Getting Device Information

--- 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 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)
CUDA Parallelism (S&K, Ch4)
Block Parallelism: S&K, Ch 4

- 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
Summing two vectors

**CPU**

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

**GPU**

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

```c
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 );
}
```
Block parallelism: add_loop_gpu.cu (S&K, Ch 04, pg 41)

```c
#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;
}
```
Block parallelism: add_loop_gpu.cu (batch script)

```bash
#!/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 ------------------------------------------------------

mpirun -np 1 -hostfile $PBS_NODEFILE ./add_loop_gpu
```
Block parallelism: add_loop_gpu.cu (output)

/var/spool/torque/mom_priv/jobs/36452.tuckoo.sdsu.edu.SC: line 10: -r: command not found
------------------------------------------------------
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/cuda_by_example/chapter04
PBS: execution mode is PBS_BATCH
PBS: job identifier is 36452.tuckoo.sdsu.edu
PBS: job name is add_loop_gpu
PBS: node file is /var/spool/torque/aux//36452.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:/usr/bin:
------------------------------------------------------
0 + 0 = 0
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
What happens to global variables?

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

```c
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] );
```
CPU Code for add function, for Np=4:

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

Example above is for cpu code on a 2 core node.

Source: Cuda By Example
GPU Code for add kernel

```c
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x; // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Example above is for GPU code with N threads.

Source: Cuda By Example
<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 2</th>
</tr>
</thead>
</table>
| ```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
``` | ```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
``` |

<table>
<thead>
<tr>
<th>Block 3</th>
<th>Block 4</th>
</tr>
</thead>
</table>
| ```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
``` | ```
__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
Setting kernel dimensions

Launch $N$ blocks with 1 thread each:

$$add \lll N, 1 \rrr (dev_a, \; \; dev_b, \; \; dev_c)$$

Launch 1 block with $N$ threads:

$$add \lll 1, N \rrr (dev_a, \; \; dev_b, \; \; dev_c)$$
code for new add kernel

```c
#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];
}

Source: Cuda By Example
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;
}
add_loop_blocks.cu (output)

[gidget:cuda/cuda_by_example/chapter05] mthomas% ./add_loop_blocks

0 + 0 = 0
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
What happens when the number of threads is larger than the number of blocks??

- cannot exceed `maxThreadsPerBlock`, typically 512
- need to distribute the threads
- need a combination of threads and blocks
- alg to convert from 2d space to 1D:
  
  ```
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  ```
- `blockDim` is constant
```c
#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;
}
```

kernel assignment for new add kernel

<table>
<thead>
<tr>
<th>Block 0</th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1</td>
<td>Thread 0</td>
<td>Thread 1</td>
<td>Thread 2</td>
<td>Thread 3</td>
</tr>
<tr>
<td>Block 2</td>
<td>Thread 0</td>
<td>Thread 1</td>
<td>Thread 2</td>
<td>Thread 3</td>
</tr>
<tr>
<td>Block 3</td>
<td>Thread 0</td>
<td>Thread 1</td>
<td>Thread 2</td>
<td>Thread 3</td>
</tr>
</tbody>
</table>

Source: Cuda By Example
Block Parallelism

add_loop_gpu.cu (S&K, Ch 04, pg 41)

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

#define N 10

__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x; // handles the data at its thread id
    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 ) );

    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;
}
```

add_loop_gpu.cu (output)

[gidget:cuda/cuda_by_example/chapter04] mthomas% ./add_loop_gpu
0 + 0 = 0
-1 + 1 = 0
-2 + 4 = 2
-3 + 9 = 6
-4 + 16 = 12
-5 + 25 = 20
-6 + 36 = 30
-7 + 49 = 42
-8 + 64 = 56
-9 + 81 = 72