Santa Clara University
School of Engineering

Introduction CUDA Programming Model

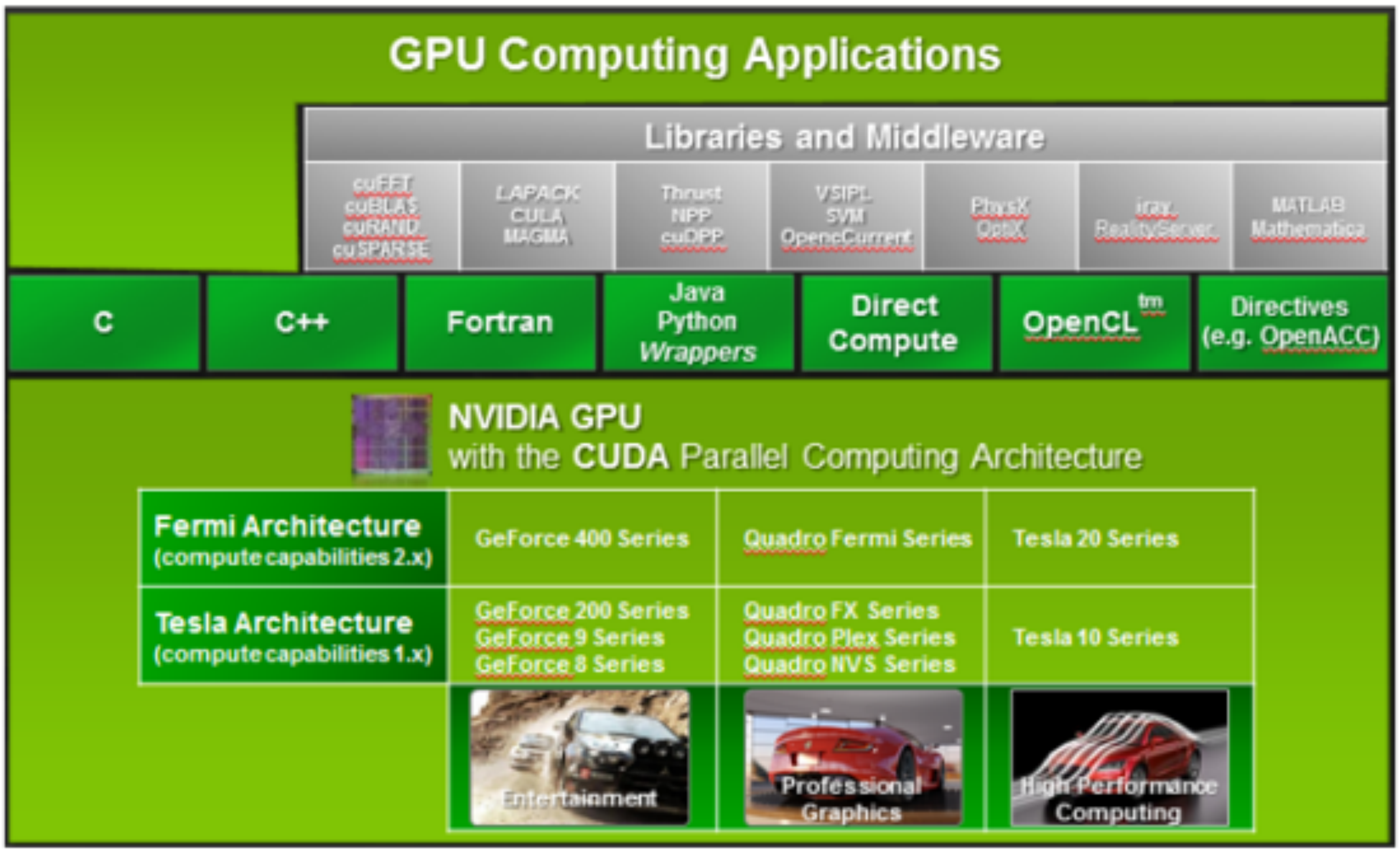
COEN 145/296 2012
Dr. Maria Pantoja

CUDA Programming Model

- To a cuda programmer the computing system consists of a host (CPU) and one or more devices (GPUs)
- Modern software applications show rich amount of data parallelism (perform arithmetic operations in a simultaneous manner)
  - CUDA devices accelerate the execution of this applications by harvesting a large amount of data parallelism

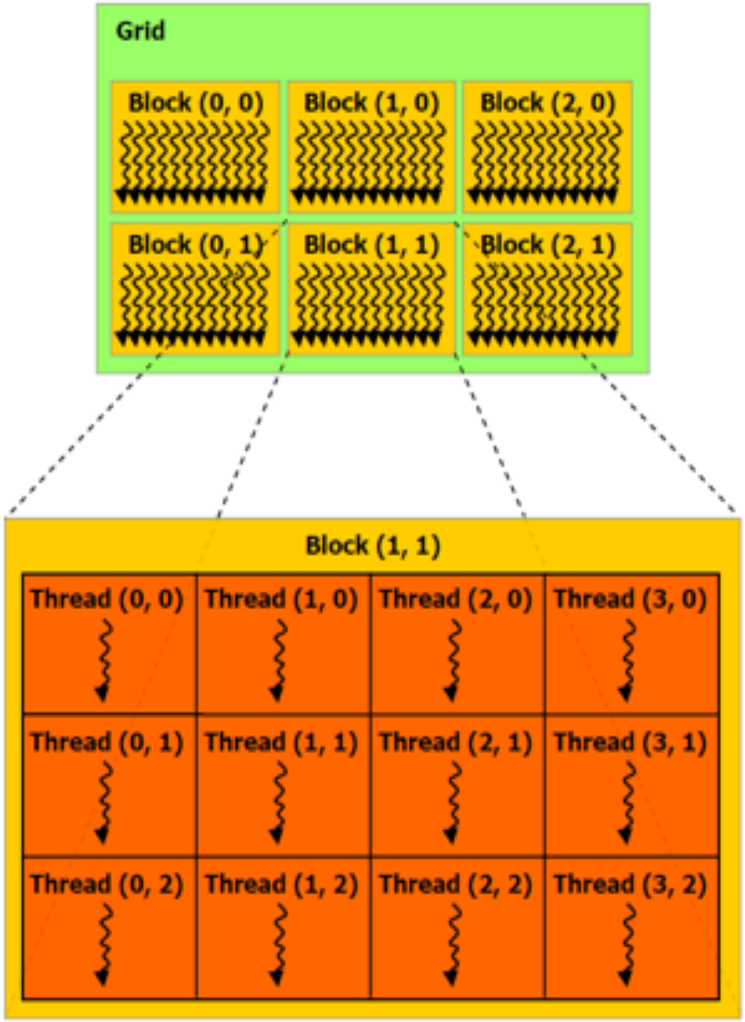
CUDA Programming Model

- CUDA comes with a software environment that allows user to use C as a high level programming language



CUDA Programming Model

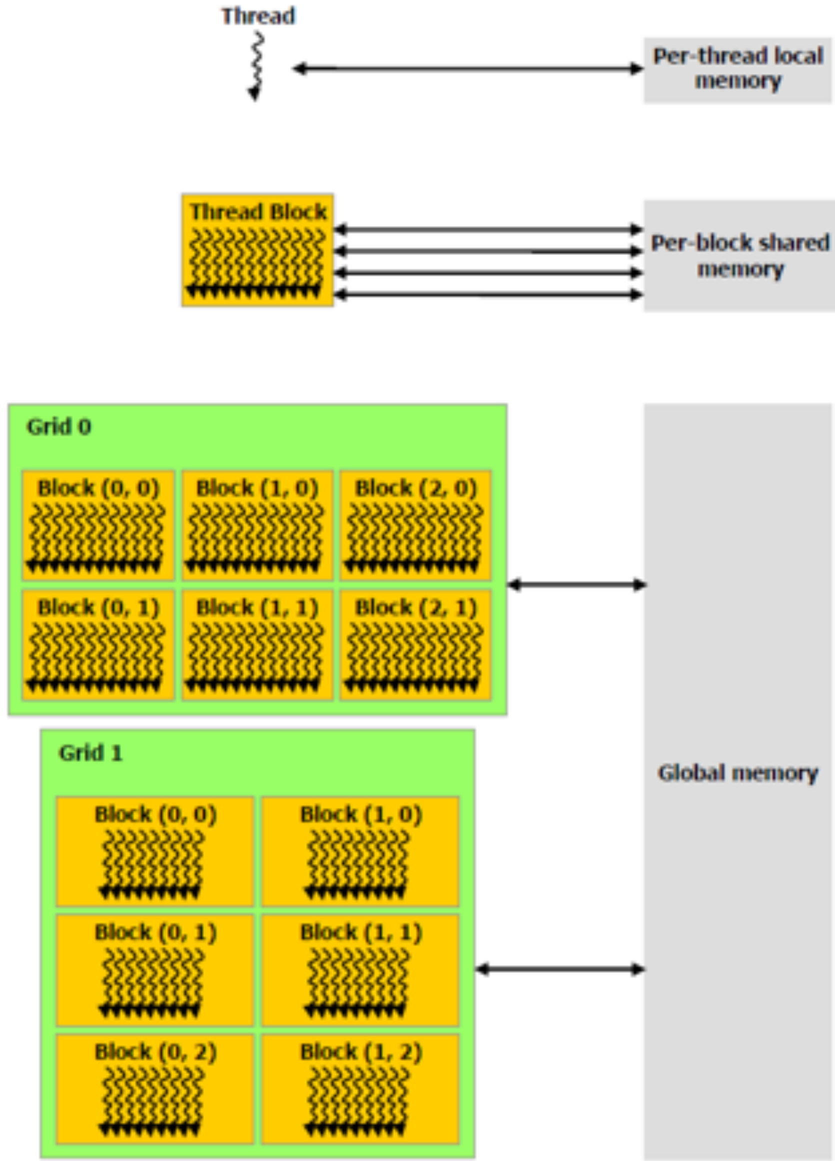
- Parallel code (kernel) is launched and executed on a device by many threads
  - A kernel function specifies the code to be executed by all threads during a parallel phase
- Launches are hierarchical
  - Threads are grouped into blocks
  - Blocks are grouped into grids



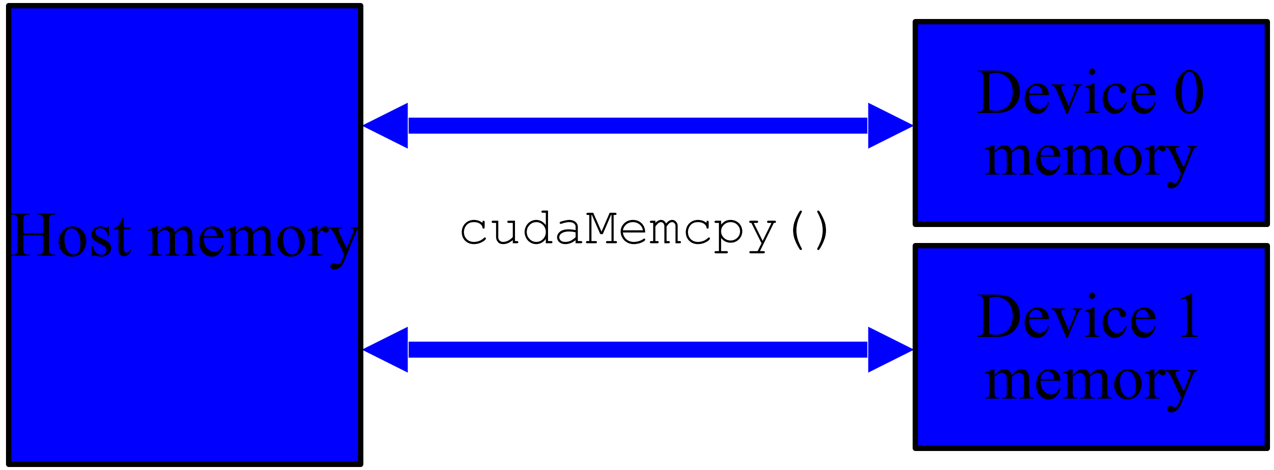
Thread Hierarchy

- Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory

Memory Model



Memory Model



Hello World

We begin by writing “hello World”

At its most basic there is no difference between CUDA C and standard C

Since “Hello World” runs entirely on the host CPU the code is the same as what you have seen before in C

Hello World

```
/*Hello World using CUDA */
#include <cuda.h>
#include <stdio.h>

// Host function
int main(void)
{
  // everyone's favorite part
  printf("Hello world\n");
  return 0;
}
```

Hello World

Lets build a more interesting Hello World program that actually used the device (GPU)
to execute code.

A funtion that executes on the device is called a kernel.

```
#include <cuda.h>
#include <stdio.h>
__global__ void kernel( void){

}

int main (void) {

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

    printf("Hello World");

    return 0;

}
```

Hello World

GPU code differences:

- ❑ An empty function named Kernel() qualified by ___global___
  - ❑ ___global___ alerts the compiler that a function should be compiled to run in the device not in the host
  - ❑ nvcc gives the function kernel() to the compiler that handles device code, and feeds main() to the host compiler
- ❑ A call to the empty function kernel()
  - ❑To invoke a device function from the host
  - ❑<<<x,y>>> are NOT parameters to the device function, they are parameters that will influence how the runtime will launch our device code. We will learn about this parameters later

Scalar addition

```
#include <cuda.h>
#include <stdio.h>

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

int main (void) {
    int c;
    int *dev_c;
    cudaMalloc((void **)&dev_c, sizeof(int));
    add<<<1,1>>>(2,7,dev_c);
    cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
    printf("2+7=%d\n",c);
    cudaFree(dev_c);
    return 0;
}
```

Scalar addition

- We can pass parameters to a Kernel function
- We need to allocate memory to do anything useful on a device, such as return value to the host
- cudaMalloc(), behaves very similar to the malloc() function in C but allocates memory on the device

```
cudaMalloc((void **)&dev_c, sizeof(int));
```

First Parameter is a pointer to the pointer you want to hold the address of the newly allocated memory

Second parameter is the size of the allocation

Scalar addition

The pointer allocated by malloc:

- ❑ can be passed to functions that execute on device
- ❑ can be used to read or write memory from code that execute on device
- ❑ can pass pointers to functions that execute on the host
- ❑ CANNOT be used to read or write memory from code that executes on the host

Scalar addition

- To free memory
  - cudaFree(dev_c);
- We can allocate and free memory on the device from the host, but we CANNOT modify this memory from the host
  - To access device memory:
    - cudaMemcpy(void * destination, const void * source, size_t num , xxx)
      - Copy num bytes from source to destination pointer
      - cudaMemcpyDeviceToHost =>source is a device pointer, destination is a cpu pointer
      - cudaMemcpyHostToDevice
    - Pointers allocated by cudaMalloc()

Example:Querying Devices

- Is nice to know:
  - How much memory the device has
  - What type of capabilities the device had
    - cudaGetDeviceCount();
    - struct cudaDeviceProp{

      …. Example

      }

      .Do something

Querying Devices

```
int main( void ) {
   cudaDeviceProp  prop;

   int count;
   cudaGetDeviceCount( &count ) ;
   for (int i=0; i< count; i++) {
      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:  " );
```

Querying Devices

```
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 );
```

Querying Devices

```
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" );
    }
}
```

CPU Code : SUM two Vectors

```
#include <stdio.h>
#define N   10

//we only have one core on the CPU
void add( int *a, int *b, int *c ) {
   int i;
   for(i=0;i<N;i++){
          c[i]=a[i]+b[i];
    }
}
```

CPU Code : SUM two Vectors

```
int main( void ) {   //on a CPU
    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;
}
```

CPU with 2 cores Code : SUM two Vectors

```
//we have two cores on the CPU
void add( int *a, int *b, int *c ) { //CPU core 1
  int tid=0;
  while(tid<N){
        c[tid]=a[tid]+b[tid];
        tid +=2;
  }
}
void add( int *a, int *b, int *c ) { //CPU core 2
  int tid=1;
  while(tid<N){
        c[tid]=a[tid]+b[tid];
        tid +=2;
  }
}
```

GPU Code : SUM two Vectors

Summing Vectors on GPU:

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

GPU Code : SUM two Vectors

```
#include <cuda.h>
#include <stdio.h>

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

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

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

GPU Code : SUM two Vectors

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

add<<<N,1>>>( dev_a, dev_b, dev_c ); //No dimension of your launch
                                     //of blocks may exceed 65535

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

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

Parallel Programming In CUDA

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

    return 0;
}
```

Parallel Programming In CUDA

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

We are launching N parrallel blocks

The collection of parallel blocks is called a grid

So:

Block1:

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

Block2:

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

Block3:

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

Block4:

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

But the number of blocks we can run can not exceed 65,535

Thread Cooperation

Launch N blocks 1 thread each:

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

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

Launch 1 block N threads per block

```
 add<<<1,N>>>( dev_a, dev_b, dev_c );
__global__ void add( int *a, int *b, int *c ) {
 int tid=threadId.x;
 if (tid<N)
        c[tid]=a[tid]+b[tid];
}
```

Thread Cooperation

The Hardware limits the dim of a grid of blocks in a single launch to 65535

The hardware limits the number of threads per block by maxThreadsPerBlock (most of the GPU currently available have 512 or 1024 as the values for maxThreadsPerBlock)

Example 8800mgtx

Warp size: 32

Maximum number of threads per block: 512

Maximum sizes of each dimension of a block: 512 x 512 x 64

Maximum sizes of each dimension of a grid: 65535 x 65535 x 1

Each block is submitted to a multiprocessor, but only 32 threads (warp size) are executed at a time. Each block can have maximum 512 threads which can be arranged in a 3D grid, while the blocks can be only in a 2D grid. So, in total executed 32 threads x number_of_multiprocessor at a time, but you can submit kernel with a total of 512x65535x65535 number of threads.

Thread Cooperation

Device "GeForce GTX 560 Ti"
Cuda Capability: 2.1
Total amount of global memory: 2014MB
(8) Multiprocessors * (48) Cuda Corse/MP: 384 CUDA cores

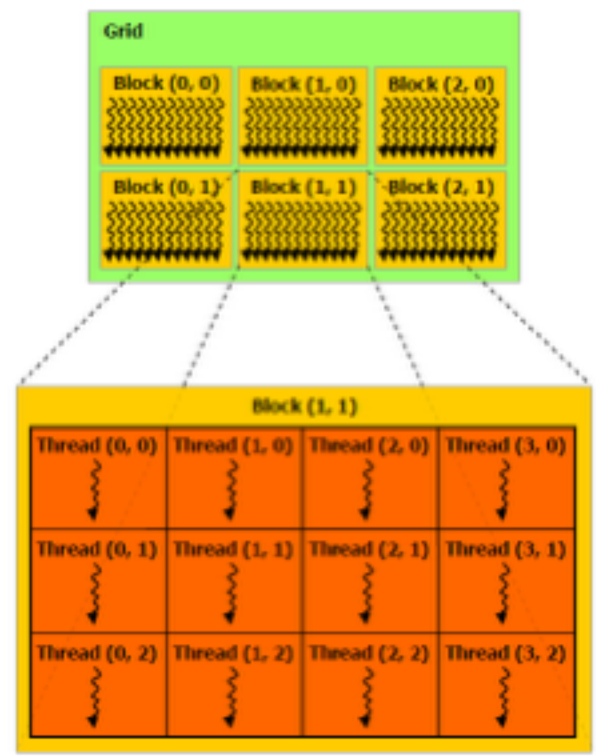
Wrap Size: 32
Max threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535

ThreadIdx, BlockIdx

Block0	Thread 0	Thread 1	Thread 2	Thread 3
Block1	Thread 0	Thread 1	Thread 2	Thread 3
Block2	Thread 0	Thread 1	Thread 2	Thread 3
Block3	Thread 0	Thread 1	Thread 2	Thread 3

Id=x+y*DIM

BlockIndex=2
ThreadIndex=2
Idx= ThreadIndex+BlockIndex*BlockDim
=2+2*4=10



Example: Vector Addition Kernel. Splitting Parallel Blocks

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}

//what if N <256?
```

Host Code

Example: Host code for `vecAdd`

```
// allocate and initialize host (CPU) memory
float *h_A = …,    *h_B = …; *h_C = …(empty)
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );
// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

Example: Host code for `vecAdd` (2)

```
// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);

// copy result back to host memory
cudaMemcpy( h_C, d_C, N * sizeof(float),
  cudaMemcpyDeviceToHost) );

// do something with the result…

// free device (GPU) memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Kernel Variations and Output

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

Thread Cooperation

GPU Sum of a longer vector:

- Hardware limits the number of blocks in a single launch to 65535 or 2^{31}
- Hardware also limits the number of threads we can launch per block. This value is especified in variable *maxThreadsPerBlock.* For many GPU this number is 512 or 1024
- So, how will we use a combination of threads/ blocks approach if we need to add two vectors of more than 512 elements.

Thread Cooperation

Index calculation : #blocks, # threads

similar to standard method of converting from a two dimmensional index space to a linear space

```
int tid=threadIdx.x+blockIdx.x*blockDim.x
```

blockDim is the #threads along each dimmension of the block. Is a three dim variable.Therefore each block is a three dimensional array of threads

Thread Cooperation

Arbitrarily set the number of threads per block to 128

Therefore need to launch N/128 Blocks

Add<<<(N+127)/128, 128>>>(dev_a, dev_b,dev_c);

Note (N+127)/128. To get the ceiling of N/128

Thread Cooperation

Limitation:

Blocks max = 65538

launch=128

means 65535*128 = 8388480 elements in array.

But today data is usually larger than that

Solution:

Thread Cooperation

```
__global__ void add(int *a, int*b, int *c) {
  int tid=threadId.x+blockId.x*blockDim.x;
  while(tid<N){
      c[tid]=a[tid]+b[tid];
      tid+=gridDim.x*blockDim.x;
  }
//very similar to the code for adding on a
//multicore cpu. In GPU implementation, we consider the
//numbers of threads launches as the number of cores
```

Thread Cooperation

Now:

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

Will fail if the number of blocks is above 65535.

To ensure we will not launch too many blocks we do fix them to a reasonably small value, for example:

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

We will see later on the consequences of this choices

Example: Vector Addition Kernel for Arbitrarily long vectors

If our current vector exceeds 65535*128 elements we will hit launch failures

Solution:

```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    while(i<N){
      C[i] = A[i] + B[i];
      i+= blockDim.x*gridDIm.x;
    }
}
```

Example: Vector Addition Kernel

```
int main()
{
      …
    // Run grid of 128 blocks of 128 threads each
    vecAdd<<< 128, 128 >>>(d_A, d_B, d_C);
      …
}
```

Code executed on GPU

- C/C++ with some restrictions:
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
  - No recursion
  - No dynamic polymorphism

- Must be declared with a qualifier:

  __global__ : launched by CPU,
  cannot be called from GPU must return void

  __device__ : called from other GPU functions,
  cannot be called by the CPU

  __host__ : can be called by CPU

  __host__ and __device__ qualifiers can be combined

Memory Spaces

- CPU and GPU have separate memory spaces
  - Data is moved across PCIe bus
  - Use functions to allocate/set/copy memory on GPU
    - Very similar to corresponding C functions
- Pointers are just addresses
  - Can't tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa

GPU Memory Allocation / Release

- Host (CPU) manages device (GPU) memory:
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - doesn’t start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

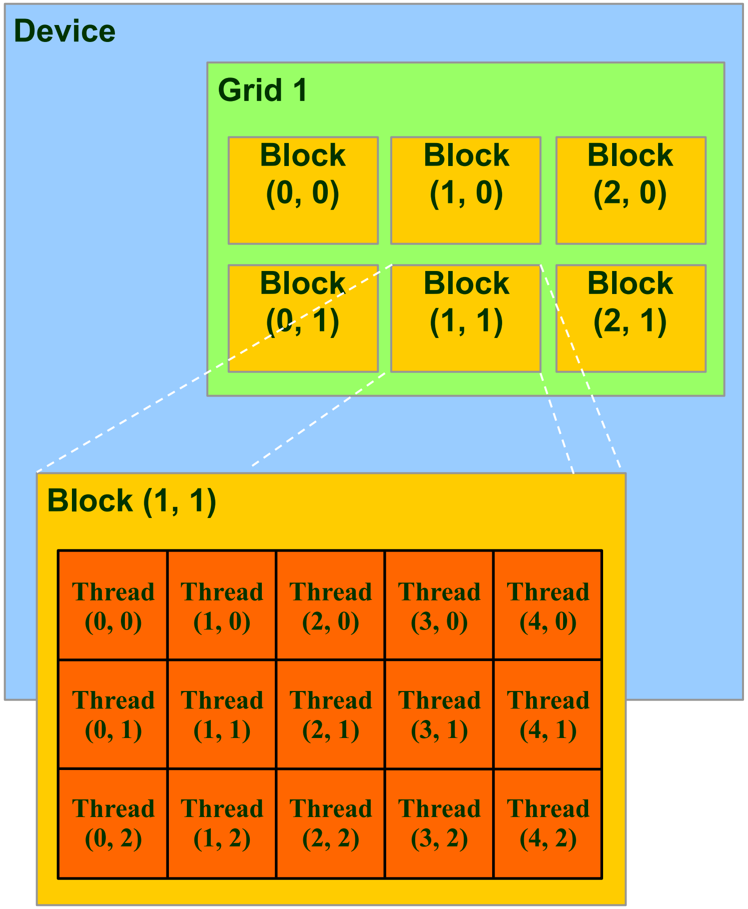
Example: Shuffling Data

```
// Reorder values based on keys

// Each thread moves one element

__global__ void shuffle(int* prev_array, int* new_array, int*
   indices)

{

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

    new_array[i] = prev_array[indices[i]];

}

int main(){

    // Run grid of N/256 blocks of 256 threads each

    shuffle<<< N/256, 256>>>(d_old, d_new, d_ind);

}
```

Host Code

IDs and Dimensions

- Threads:
  - 3D IDs, unique within a block
- Blocks:
  - 2D IDs, unique within a grid
- Dimensions set at launch
  - Can be unique for each grid
- Built-in variables:
  - threadIdx, blockIdx
  - blockDim, gridDim



Kernel with 2D Indexing

```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;


    a[idx]  = a[idx]+1;
}
```

```
__global__ void kernel( int *a, int dimx, int dimy )
{
   int ix   = blockIdx.x*blockDim.x + threadIdx.x;
   int iy   = blockIdx.y*blockDim.y + threadIdx.y;
   int idx = iy*dimx + ix;

   a[idx]  = a[idx]+1;
}
```

```
int main(){
   int dimx = 16;    int dimy = 16;
   int num_bytes = dimx*dimy*sizeof(int);

   int *d_a=0, *h_a=0; // device and host pointers

   h_a = (int*)malloc(num_bytes);
   cudaMalloc( (void**)&d_a, num_bytes );

   if( 0==h_a || 0==d_a )    {
      printf("couldn't allocate memory\n");
      return 1;
   }
   cudaMemset( d_a, 0, num_bytes );
   dim3 grid, block;
   block.x = 4;     block.y = 4;
   grid.x  = dimx / block.x;     grid.y  = dimy / block.y;

   kernel<<<grid, block>>>( d_a, dimx, dimy );

   cudaMemcpy( h_a, d_a, num_bytes,cudaMemcpyDeviceToHost);

   for(int row=0; row<dimy; row++){
      for(int col=0; col<dimx; col++)
         printf("%d ", h_a[row*dimx+col] );
      printf("\n");
   }
   free( h_a );
   cudaFree( d_a );
   return 0;
}
```

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
- Independence requirement gives scalability

Matrix Addition in GPU

- threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional *thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.*

- The index of a thread and its thread ID relate to each other in a straightforward way:

  - For a one-dimensional block, they are the same;
  - for a two-dimensional block of size *(Dx, Dy), the thread ID of a thread of index (x, y) is (x + y Dx);*
  - *for a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is (x + y Dx + z Dx Dy).*

- As an example, the following code adds two matrices *A and B of size NxN and stores the result into matrix C:*

Matrix Addition in GPU

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
   int i = threadIdx.x;
   int j = threadIdx.y;
   C[i][j] = A[i][j] + B[i][j];
}
int main()
{
   ...
   // Kernel invocation with one block of N * N * 1 threads
   int numBlocks = 1;
   dim3 threadsPerBlock(N, N);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
   ...
}
```

Matrix Addition in GPU

Extending the previous **MatAdd() example to handle multiple blocks, the code becomes as follows.**

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i < N && j < N)
         C[i][j] = A[i][j] + B[i][j];
}
int main()
{
   ...
   // Kernel invocation
   dim3 threadsPerBlock(16, 16);
   dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
   ...
}
```

Matrix Addition in GPU

- A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before.
- Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. Enabling programmers to write code that scales with the number of cores.
- Threads within a block can cooperate by sharing data through some *shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the* ***__syncthreads(). Next chapter will show how to use shared memory.***
- For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and **__syncthreads() is expected to be lightweight.**

Julia Set

How to draw slices of the Julia Set.

The Julia Set is the boundary of a certain class of functions over complex numbers. For almost all values of the function's parameters, this boundary forms a fractal.

At its heart, the Julia Set evaluates a simple iterative equation for points in the complex plane. A point is not in the set if the process of iterating the equation diverges for that point. That is, if the sequence of values produced by iterating the equation grows toward infinity, a point is considered outside the set. Conversely, if the values taken by the equation remain bounded, the point is in the set

The iterative equation is:

$$Z_{n+1}=Z^2_n+C$$

Computing an iteration involves squaring the current value and adding a constant to get the next value of the equation

Julia Set on CPU

```
#include <bitmap.h>
//for info on bitmap library
//http://bitmap.codeplex.com

int main (void){
   CPUBitmap bitmap( DIM, DIM); //creates a bitmap using library
   unsigned char *ptr=bitmap.get_ptr();

   kernel(ptr); //passes a pointer to the bitmap to a kernel function
   bitmap.display_and_exit();
}
```

Julia Set on CPU

```
#define DIM 300
typedef struct {
    float   r;
    float   i;
} cuComplex;
float magnitude2(cuComplex x) {
   return x.r*x.r +x.i*x.i;
}
cuComplex add(cuComplex x, cuComplex y) {
   cuComplex z;
   z.r=x.r+y.r;
   z.i=x.r+y.r;
   return z;
}
cuComplex mult(cuComplex x, cuComplex y) {
   cuComplex z;
   z.r=x.r*y.r-x.i*a.i;
   z.i=x.i*y.r+x.r*y.i;
   return z;
}
```

Julia Set on CPU

```
//iterates through all points we care to render, calling julia() on each point
    to determine membership
void kernel (unsigned char *ptr){
   int y,x,offset;
   for(y=0;y<DIM;y++){
          for(x=0;x<DIM;x++){
                 offset=x+y*DIM;   //Linear offset into output buffer
                 juliaValue=julia(x,y);
                 ptr[offset*4+0]=255*juliaValue; //set color to red if point is in set
                 ptr[offset*4+1]=0;     //green
                 ptr[offset*4+2]=0;     //blue
          }
   }
}
```

Julia Set on CPU

```
int julia(int x, int y){
   const float scale=1.5;
    //translate pixel coordinate into complex space and scale to -1, 1
   float jx=scale*(float)(DIM/2-x)/(DIM/2);
   float jy=scale*(float)(DIM/2-y)/(DIM/2);

   cuComplex c; c.r=-0.8; c.i=0.156;
   cuComplex a; a.r=jx; a.i=jy;
   int i=0;
   for(i=0; i<200;i++){
           a=add(mult(a,a),c);
           if(magnitude2(a) >1000)
                 return 0;
   }
   return 1;
}
```

Julia Set on GPU

```
#define DIM 300
struct cuComplex {
    float   r;
    float   i;
    __device__ cuComplex( float a, float b ) : r(a), i(b)  {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

Julia Set on GPU

```
__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

Julia Set on GPU

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

    // now calculate the value at that position
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

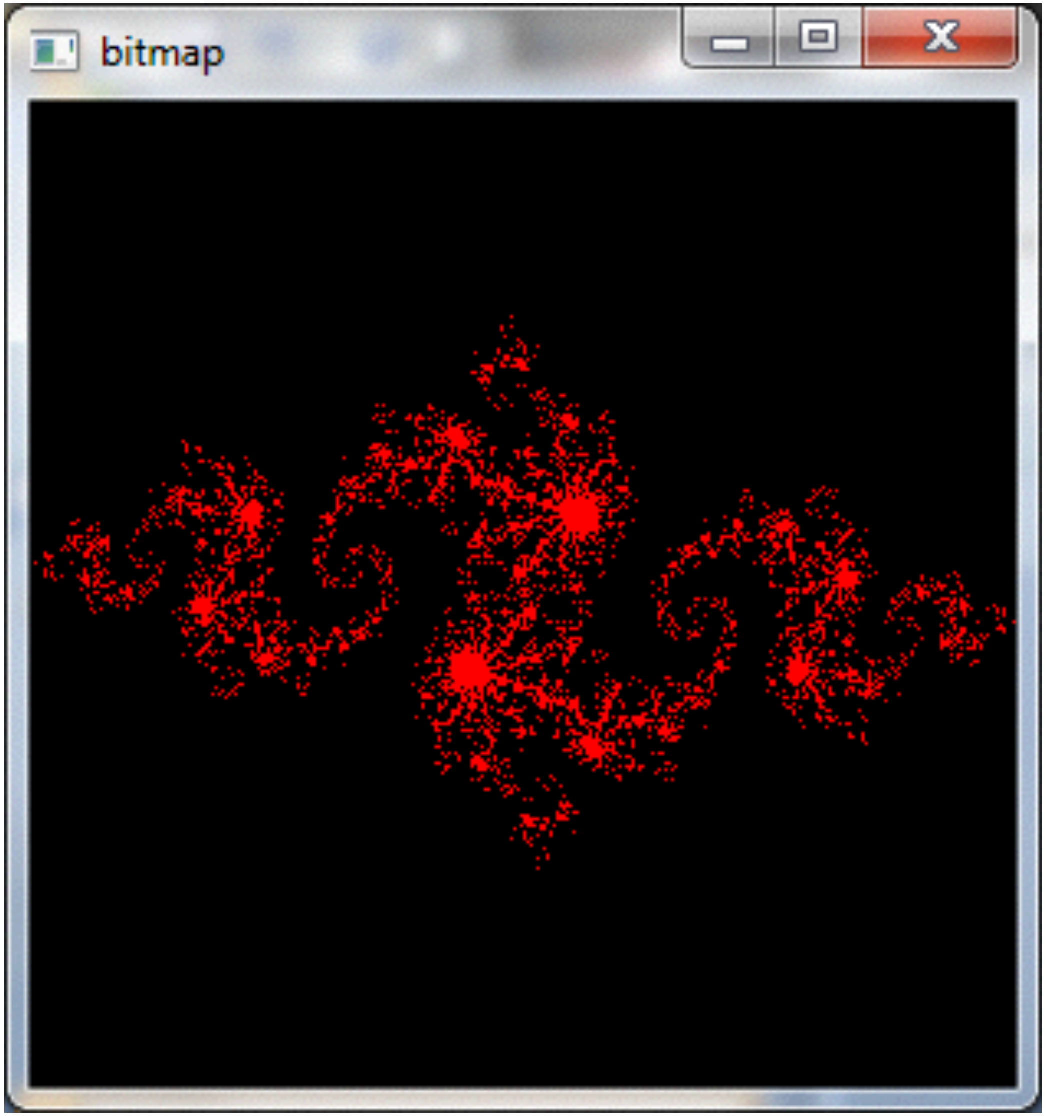
Julia Set on GPU

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

    // now calculate the value at that position
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

Julia Set on GPU

```
// globals needed by the update routine
struct DataBlock {
    unsigned char   *dev_bitmap;
};
int main( void ) {
    DataBlock   data;
    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char    *dev_bitmap;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap, bitmap.image_size() ) );
    data.dev_bitmap = dev_bitmap;
    dim3    grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );

    HANDLE_ERROR( cudaFree( dev_bitmap ) );

    bitmap.display_and_exit();
}
```

Julia Set on GPU



Hello World

```
/*** Hello World using CUDA The string "Hello World!" is mangled then restored using a common    CUDA idiom
Byron Galbraith */
#include <cuda.h>
#include <stdio.h>

// Prototypes
__global__ void helloWorld(char*);

// Host function
int main(int argc, char** argv){
  int i;
  // desired output
  char str[] = "Hello World!";

  // mangle contents of output
  // the null character is left intact for simplicity
  for(i = 0; i < 12; i++)
    str[i] -= i;

  // allocate memory on the device
  char *d_str;
  size_t size = sizeof(str);
  cudaMalloc((void**)&d_str, size);
```

Hello World

```
// copy the string to the device
  cudaMemcpy(d_str, str, size, cudaMemcpyHostToDevice);

  // set the grid and block sizes
  dim3 dimGrid(2);   // one block per word
  dim3 dimBlock(6); // one thread per character

  // invoke the kernel
  helloWorld<<< dimGrid, dimBlock >>>(d_str);

  // retrieve the results from the device
  cudaMemcpy(str, d_str, size, cudaMemcpyDeviceToHost);

  // free up the allocated memory on the device
  cudaFree(d_str);

  // everyone's favorite part
  printf("%s\n", str);

  return 0;
}
```

Hello World

```
// Device kernel
__global__ void
helloWorld(char* str)
{
  // determine where in the thread grid we are
  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  // unmangle output
  str[idx] += idx;
}
```

GPU Ripple Using GPU

```
__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;

    // now calculate the value at that position
    float fx = x - DIM/2;
    float fy = y - DIM/2;
    float d = sqrtf( fx * fx + fy * fy );
    unsigned char grey = (unsigned char)(128.0f + 127.0f *
                                         cos(d/10.0f - ticks/7.0f) /
                                         (d/10.0f + 1.0f));
    ptr[offset*4 + 0] = grey;
    ptr[offset*4 + 1] = grey;
    ptr[offset*4 + 2] = grey;
}
```

GPU Ripple Using GPU

```
struct DataBlock {
    unsigned char   *dev_bitmap;
    CPUAnimBitmap  *bitmap;
};

void generate_frame( DataBlock *d, int ticks ) {
    dim3    blocks(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<blocks,threads>>>( d->dev_bitmap, ticks );

    HANDLE_ERROR( cudaMemcpy( d->bitmap->get_ptr(),
                              d->dev_bitmap,
                              d->bitmap->image_size(),
                              cudaMemcpyDeviceToHost ) );
}
```

GPU Ripple Using GPU

```
// clean up memory allocated on the GPU
void cleanup( DataBlock *d ) {
   HANDLE_ERROR( cudaFree( d->dev_bitmap ) );
}
int main( void ) {
   DataBlock   data;
   CPUAnimBitmap  bitmap( DIM, DIM, &data );
   data.bitmap = &bitmap;

   HANDLE_ERROR( cudaMalloc( (void**)&data.dev_bitmap,
                       bitmap.image_size() ) );

   bitmap.anim_and_exit( (void (*)(void*,int))generate_frame,
                       (void (*)(void*))cleanup );
}
```

GPU Ripple Using GPU

Debugging

Nvidia's cuda-gdb debugger:

compile code with –g flag

nvcc –g –G example.cu –o example

Commands:

breakpoint(b)

run (r)

next(n)

backtrace(bt)

thread:list current CPU thread

cuda thread: current GPU thread

cuda kernel: list currently active GPU

Debugging in Windows

Parallel Nsight:

Integrated with Visual Studio

Tutorial: http://http.developer.nvidia.com/ParallelNsight/2.1/Documentation/UserGuide/HTML/Parallel_Nsight_User_Guide.htm