LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Introduction CUDA

Instructor: Haidar M. Harmanani

Spring 2017

Memory Management

- `host` and `device` memory are distinct entities in CUDA:
  - Device pointers point to GPU memory
  - May be passed to and from host code
  - May not be dereferenced from host code
- Host pointers point to CPU memory
  - May be passed to and from device code
  - May not be dereferenced from device code
- Basic CUDA API for dealing with device memory
  - `cudaMalloc(), cudaFree(), cudaMemcpy()`
  - Similar to their C equivalents, `malloc(), free(), memcpy()`

Adding two numbers on the GPU

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

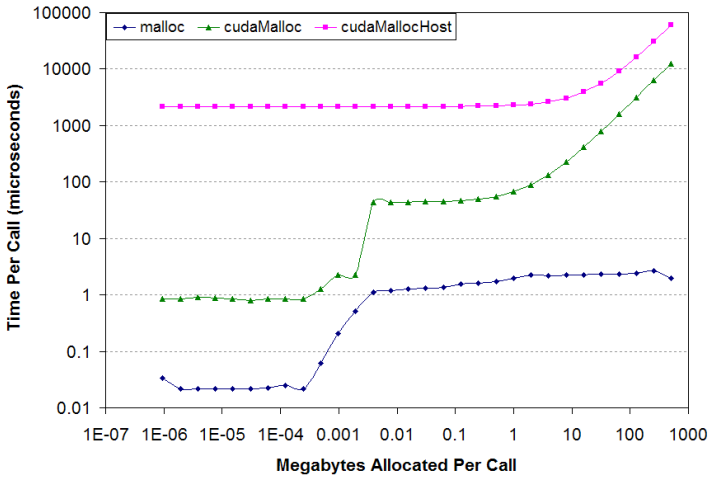
- add()
  - __global__ indicates that the function runs on the device and called from host code
    - …so a, b, and c must point to device memory

Adding two numbers on the GPU

```
int main( void )
{
      int a, b, c;                              // host copies of a, b, c
      int *dev_a, *dev_b, *dev_c;               // device copies of a, b, c
      int size = sizeof( int);                  // we need space for an integer

      // allocate device copies of a, b, c
      cudaMalloc( (void**)&dev_a, size );
      cudaMalloc( (void**)&dev_b, size );
      cudaMalloc( (void**)&dev_c, size );
      a = 2;lock
      b = 7;
      // copy inputs to device
      cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice);
      cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice);

    // launch add() kernel on GPU, passing parameters
      add<<< 1, 1 >>>( dev_a, dev_b, dev_c);

    // copy device result back to host copy of c
    cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost);
    cudaFree( dev_a);
    cudaFree( dev_b);
    cudaFree( dev_c);
    return 0;
}
```

Kernel Launch for one block with one thread

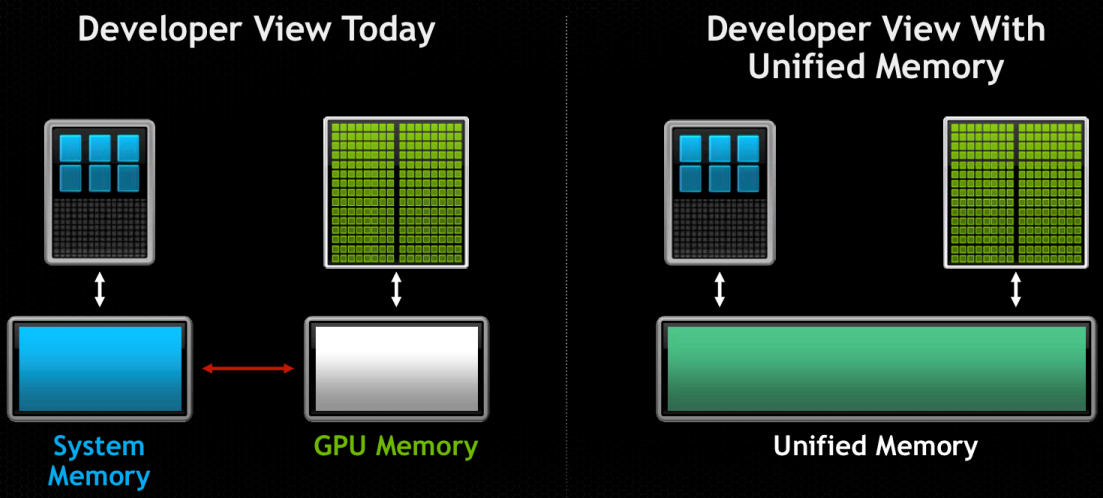
Time per allocation as a function of the number of bytes allocated per call



Unified Memory in CUDA 6

- As of CUDA 6, NVIDIA introduced `Unified Memory`
  - In a typical PC or cluster node today, the memories of the CPU and GPU are physically distinct and separated by the PCI-Express bus.
  - Data that is shared between the CPU and GPU must be allocated in both memories, and explicitly copied between them by the program.
- Bridging the CPU-GPU divide
  - Unified Memory creates a pool of managed memory that is shared between the CPU and GPU
  - Managed memory is accessible to both the CPU and GPU using a single pointer.
  - The system automatically migrates data allocated in Unified Memory between host and device so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU.

Unified Memory
Dramatically Lower Developer Effort



Super Simplified Memory Management Code

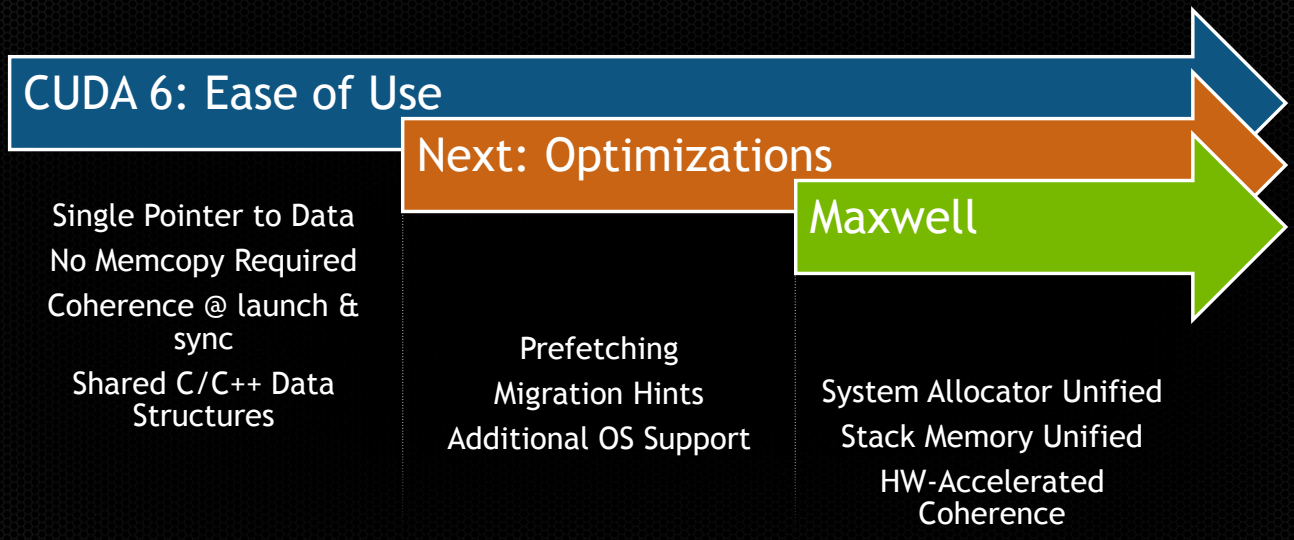
CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

Unified Memory Roadmap



"cudaMallocManaged" vs. "cudaMalloc"

- Some have benchmarked both and noted that is slightly "cudaMallocManaged" slower than "cudaMalloc"
- The following can be noted:
  - The main bottleneck is on the PCIE bus
  - cudaMallocManaged is of interest to a non-proficient CUDA programmer
    - Simpler learning curve
  - The Maxwell and Pascal architectures provide HW support for unified memory so cudaMallocManaged() provides better performance on those architectures

Back to Our Addition Example: Unified Memory (CUDA 6 and beyond)

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

int main( void )
{
      int *a, *b, *c;              // host copies of a, b, c
      int size = sizeof (int);     // The total number of bytes per vector

   // Allocate memory
   cudaMallocManaged(&a, size);
   cudaMallocManaged(&b, size);
   cudaMallocManaged(&c, size);

  // launch add() kernel on GPU, passing parameters
   add<<< 1, 1 >>> ( a, b, c);

   cudaDeviceSynchronize(); // Wait for the GPU to finish

    // Free all our allocated memory
    cudaFree( a ); cudaFree( b ); cudaFree( c );
}
```

Run one block with one thread

Blocks until the device has completed all preceding requested tasks.

Vector Addition on the GPU

- Well, since GPUs are about massive parallelism we will rewrite the add() method such that
  - Each invocation of add() is referred to as a *block*
  - Kernel can refer to its block's index with the variable blockIdx.x
  - Each block adds a value from a[] and b[], storing the result in c[]
- By using blockIdx.x to index arrays, each block handles different indices

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

Vector Addition on the GPU

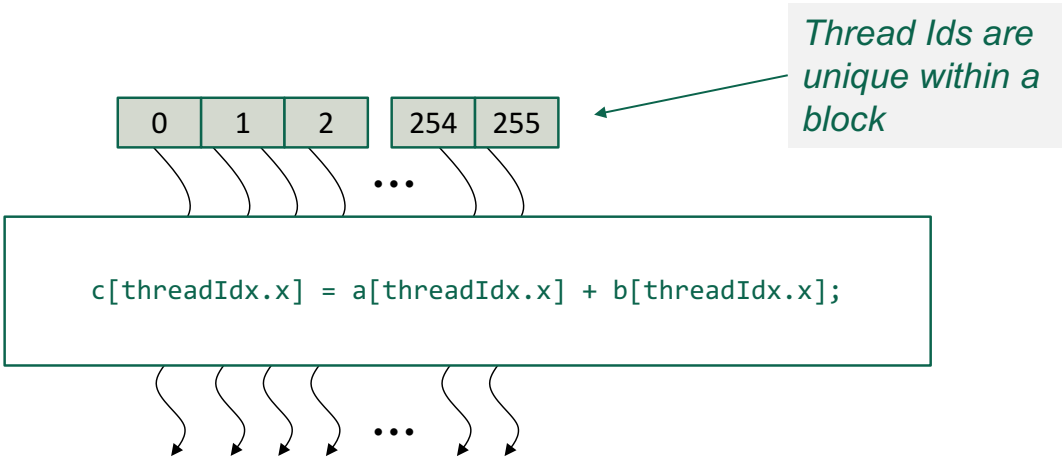

```
__global__ void add( int *a, int *b, int *c) {
      c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```


Vector Addition on the GPU: N Blocks

```
#define N 512
int main( void )
{
      int *a, *b, *c;                    // a, b, c
      int size = N * sizeof( int);       // The total number of bytes per vector

      // Allocate memory
      cudaMallocManaged(&a, size);
      cudaMallocManaged(&b, size);
      cudaMallocManaged(&c, size);

      random_ints( a, N );
      random_ints( b, N );

      // launch add() kernel on GPU using N Parallel Blocks
      add<<< N, 1 >>>( a, b, d);

      cudaDeviceSynchronize(); // Wait for the GPU to finish

      // Free all our allocated memory
      cudaFree( a ); cudaFree( b ); cudaFree( c );
}
```

Launch N blocks with 1 thread each

Arrays of Parallel Threads



Arrays of Parallel Threads

- To execute kernels in parallel with CUDA
  - Launch a grid of blocks of threads, specifying the number of blocks per grid (bpg) and threads per block (tpb).
  - Total number of threads launched will be the product of bpg × tpb
  - This can be in the millions!

Caveat: Threads in CUDA

- Hardware limits
  - The number of blocks in a single launch to 65,535
  - The number of threads per block with which we can launch a kernel to a maximum of `maxThreadsPerBlock`
  - Number is hardware dependent
    - 1024 threads per block on a Kepler, 512 on most older GPUs
- How do we use a thread-based approach to add two vectors of size greater than 512 or 1024?
- Each thread has a gloabl ID
  - This is basically just finding an offset given a 2D grid of 3D blocks of 3D threads, but can get very confusing!

GPU *n* Numbers Thread Addition

- Divide thread array into multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks cannot cooperate

GPU *n* Numbers Thread Addition

- A block can be split into parallel threads
- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (SPMD)
  - Each thread has an index that it uses to compute memory addresses and make control decisions

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

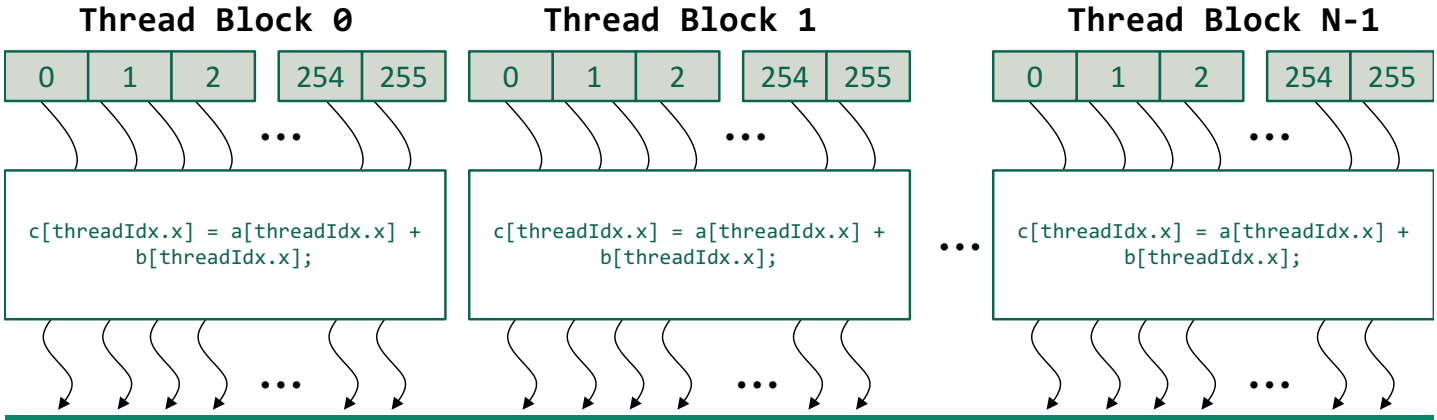

Vector Addition on the GPU: 1 Block with N Threads

```
#define N 512
int main( void )
{
      int *a, *b, *c;                    // a, b, c
      int size = N * sizeof( int);       // The total number of bytes per vector

      // Allocate memory
      cudaMallocManaged(&a, size);
      cudaMallocManaged(&b, size);
      cudaMallocManaged(&c, size);

      random_ints( a, N );
      random_ints( b, N );

      // launch add() kernel on GPU using N Parallel Blocks
      add<<< 1, N >>>( a, b, d);

      cudaDeviceSynchronize(); // Wait for the GPU to finish


      // Free all our allocated memory
      cudaFree( a ); cudaFree( b ); cudaFree( c );
}
```

Launch 1 block with N threads each

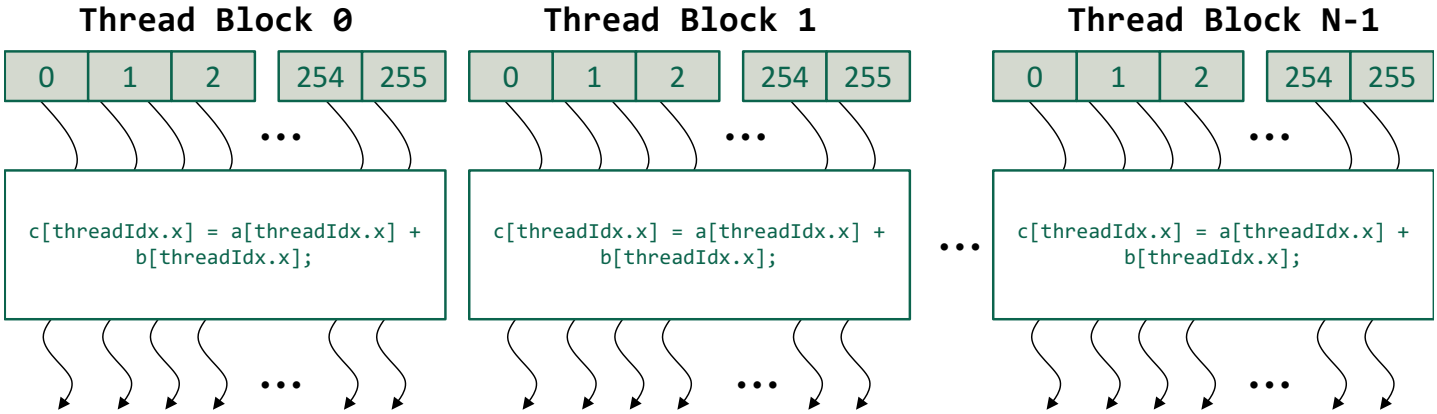
Combining Threads and Blocks

Thread Ids are not unique across all blocks???



Combining Threads and Blocks

- Indexing arrays with threads and blocks is now problematic!
- Solution?



Blocks, Grids, and Threads

- Kernels are launched as *grids* of *blocks* of threads.
  - Threads can further be divided into 32-thread *warps*, and each thread in a warp is called a *lane*
- Thread blocks are separately scheduled onto SMs, and threads within a given block are executed by the same SM

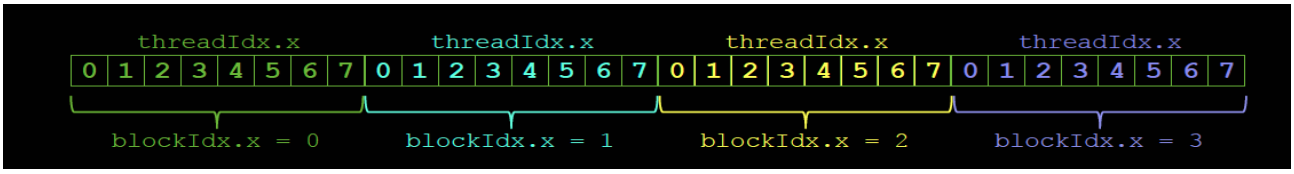
Combining Threads and Blocks

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

Figure 5.1 A two-dimensional arrangement of a collection of blocks and threads

Combining Threads and Blocks

- No longer as simple as just using `threadIdx.x` or `blockIdx.x` as indices
- To index array with 1 thread per entry (using 8 threads/block)



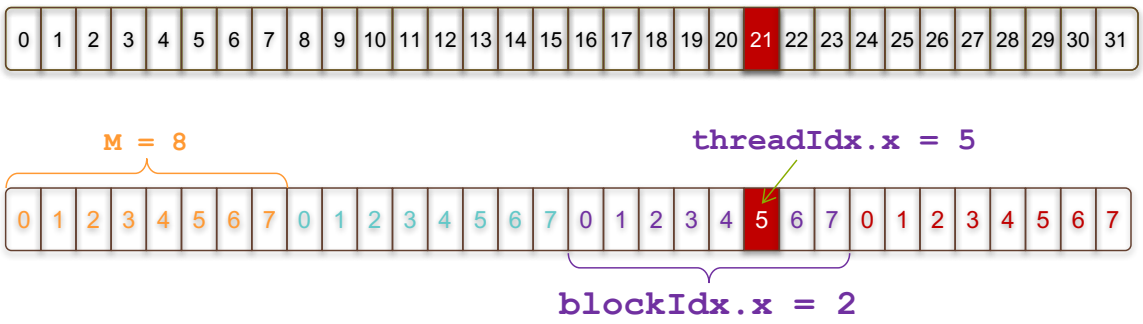
- If we have M threads/block, a unique array index for each entry given by

  int index = threadIdx.x + blockIdx.x * M;

  int index = x + y * Width

Indexing Arrays: Example

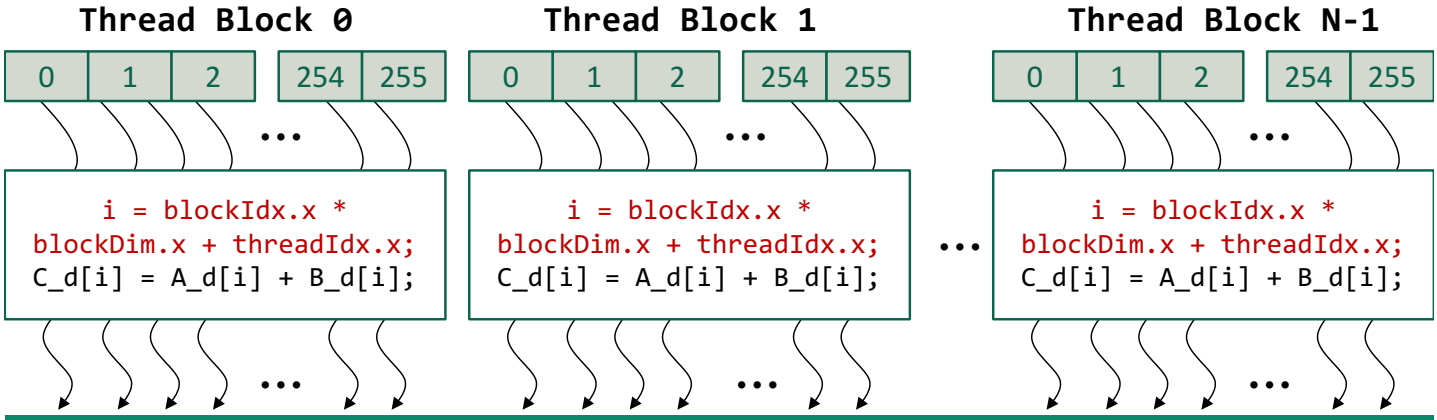
- Which thread will operate on the red element?




```
int index = threadIdx.x + blockIdx.x * M;
          =       5     +      2       * 8;
          = 21;
```

Combining Threads and Blocks

- Indexing arrays with with threads and blocks is a problem
- Solution?



Combining Threads and Blocks

- The blockDim.x is a built-in variable for threads per block:

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

- A combined version of the vector addition kernel to use blocks *and* threads:

```
__global__ void add( int *a, int *b, int *c ) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

```
int main( void )
{
       int *a, *b, *c;              // host copies of a, b, c
       int*dev_a, *dev_b, *dev_c;   // device copies of a, b, c

       int size = N * sizeof( int);             // we need space for an integer

       // allocate device copies of a, b, c
       cudaMalloc( (void**)&dev_a, size );
       cudaMalloc( (void**)&dev_b, size );
       cudaMalloc( (void**)&dev_c, size );

       a = (int*) malloc( size );
       b = (int*) malloc( size );
       c = (int*) malloc( size );
       random_ints( a, N );
       random_ints( b, N );

       // copy inputs to device
       cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice);
       cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice);

     // launch add() kernel on GPU using N threads
       add <<< N, N >>>( dev_a, dev_b, dev_c);

     // copy device result back to host copy of c
     cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost);

     Free(a); free(b); free(c);
     cudaFree( dev_a); cudaFree( dev_b); cudaFree( dev_c);
     return 0;
}
```

Launch N blocks with N threads each

Adding n-numbers on the GPU

Recap

- Launching parallel kernels
  - Launch **N** copies of **add()** with **add<<<N/M,M>>>(…);**
  - Use **blockIdx.x** to access block index
  - Use **threadIdx.x** to access thread index within block

- Allocate elements to threads:

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

Adding n-numbers on the GPU

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512

int main( void )
{
    int *a, *b, *c;             // host copies of a, b, c
    int*dev_a, *dev_b, *dev_c; // device copies of a, b, c
    intsize = N * sizeof( int);       // we need space for an integer

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*) malloc( size );
    b = (int*) malloc( size );
    c = (int*) malloc( size );
    random_ints( a, N );
    random_ints( b, N );
```


Adding n-numbers on the GPU (Cont.)

```
    // copy inputs to device
    cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice);

    // launch add() kernel on GPU, passing parameters
    add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(dev_a, dev_b, dev_c);

    // copy device result back to host copy of c
    cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost);

    Free(a); free(b); free(c);
    cudaFree(dev_a); cudaFree( dev_b); cudaFree( dev_c);
    return 0;
}
```

Handling Arbitrary Vector Sizes

Typical problems are not friendly multiples of `blockDim.x`

Avoid accessing beyond the end of the arrays:

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

Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```