An Introduction to GPU Computing and CUDA Architecture

Sarah Tariq, NVIDIA Corporation
M02: High Performance Computing with CUDA

Parallel Programming with CUDA

Ian Buck
GPU Computing

- GPU: Graphics Processing Unit
- Traditionally used for real-time rendering
- High computational density (100s of ALUs) and memory bandwidth (100+ GB/s)
- Throughput processor: 1000s of concurrent threads to hide latency (vs. large fast caches)
What is CUDA?

- CUDA Architecture
  - Expose GPU computing for general purpose
  - Retain performance

- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.

- This session introduces CUDA C/C++
CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
HELLO WORLD!

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
Heterogeneous Computing

- Terminology:
  - Host: The CPU and its memory (host memory)
  - Device: The GPU and its memory (device memory)
Heterogeneous Computing

```c
#include <iostream>
#include <algorithm>

using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__
void stencil_1d(int* in, int* out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int* x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int* in, *out;
    // host copies of a, b, c
    int* d_in, *d_out;
    // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int*)malloc(size);
    fill_ints(in, N + 2*RADIUS);
    out = (int*)malloc(size);
    fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void**)&d_in, size);
    cudaMalloc((void**)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE, BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in);
    cudaFree(d_out);
    return 0;
}
```
1. Copy input data from CPU memory to GPU memory
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
Hello World!

```c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no `device` code

Output:

```
$ nvcc hello_world.cu
$ a.out
Hello World!
$ 
```
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

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

- Two new syntactic elements…
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc, cl.exe`
Hello World! with Device Code

mykernel<<<1,1>>>();

- Triple angle brackets mark a call from host code to device code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment

- That’s all that is required to execute a function on the GPU!
Parallel Programming in CUDA C/C++

- But wait… GPU computing is about massive parallelism!
- We need a more interesting example…
- We’ll start by adding two integers and build up to vector addition
Addition on the Device

- A simple kernel to add two integers

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

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host
Addition on the Device

- Note that we use pointers for the variables

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

- `add()` runs on the device, so `a`, `b`, and `c` must point to device memory

- We need to allocate memory on the GPU
Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code

- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`
int main(void) {
    int a, b, c;               // host copies of a, b, c
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Setup input values
    a = 2;
    b = 7;
Addition on the Device: `main()`

```c
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
```
CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

RUNNING IN PARALLEL
Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

  \[
  \text{add}<<<1, 1>>>();
  \]

  \[
  \downarrow
  \]

  \[
  \text{add}<<<N, 1>>>();
  \]

- Instead of executing `add()` once, execute N times in parallel
Vector Addition on the Device

- With `add()` running in parallel we can do vector addition.

- Terminology: each parallel invocation of `add()` is referred to as a block.
  - The set of blocks is referred to as a grid.
  - Each invocation can refer to its block index using `blockIdx.x`.

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

- By using `blockIdx.x` to index into the array, each block handles a different index.
Vector Addition on the Device

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

- On the device, each block can execute in parallel:

<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>c[0] = a[0] + b[0];</td>
<td>c[1] = a[1] + b[1];</td>
</tr>
<tr>
<td>Block 2</td>
<td>Block 3</td>
</tr>
</tbody>
</table>
#define N 512
int main(void) {

    int *a, *b, *c;     // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void**)&d_a, size);
    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
Vector Addition on the Device: main()

// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
INTRODUCING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
CUDA Threads

- Terminology: a block can be split into parallel threads

- Let’s change `add()` to use parallel threads instead of parallel blocks

Using blocks:

```c
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
    add<<<N,1>>>(d_a, d_b, d_c);
}
```

Using threads:

```c
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
    add<<<1,N>>>(d_a, d_b, d_c);
}
```
COMBINING THREADS AND BLOCKS
Combining Blocks and Threads

- We’ve seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads

- Let’s adapt vector addition to use both blocks and threads
Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

<table>
<thead>
<tr>
<th>threadIdx.x</th>
<th>threadIdx.x</th>
<th>threadIdx.x</th>
<th>threadIdx.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

- With M threads/block a unique index for each thread is given by:

  ```
  int index = threadIdx.x + blockIdx.x * M;
  ```
Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block
  
  ```c
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  ```

- Combined version of `add()` to use parallel threads and parallel blocks
  
  ```c
  __global__ void add(int *a, int *b, int *c) {
      int index = threadIdx.x + blockIdx.x * blockDim.x;
      c[index] = a[index] + b[index];
  }
  ```

- What changes need to be made in `main()`?
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
Addition with Blocks and Threads: `main()`

```c
// Copy inputs to device
cudamemcpy(d_a, a, size, cudamemcpyHostToDevice);
cudamemcpy(d_b, b, size, cudamemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N_THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudamemcpy(c, d_c, size, cudamemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudafree(d_a); cudafree(d_b); cudafree(d_c);
return 0;
```
Launching kernels on GPU

Launch parameters:

- grid dimensions (up to 2D)
- thread-block dimensions (up to 3D)
- shared memory: number of bytes per block
  - for extern smem variables declared without size
  - Optional, 0 by default
- stream ID
  - Optional, 0 by default

```c
dim3 grid(16, 16);
dim3 block(16, 16);
kernel<<grid, block, 0, 0>>>(...);
kernell<<32, 512>>>(...);
```
IDs and Dimensions

**Threads:**
- 3D IDs, unique within a block

**Blocks:**
- 2D IDs, unique within a grid

**Dimensions set at launch time**
- Can be unique for each section

**Built-in variables:**
- threadIdx, blockIdx
- blockDim, gridDim
Minimal Kernel for 2D data

`__global__ void assign2D(int* d_a, int w, int h, int value)`

```c
{
    int iy = blockDim.y * blockIdx.y + threadIdx.y;
    int ix = blockDim.x * blockIdx.x + threadIdx.x;
    int idx = iy * w + ix;

    d_a[idx] = value;
}
```
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);
```
Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize

- To look closer, we need a new example…
Warps and SIMT

- A warp is a group of threads within a block that are launched together and (usually) execute together

Conceptual Programming Model

Control  Control  Control  Control  Control  Control

ALU      ALU      ALU      ALU      ALU      ALU

Conceptual SIMT Execution Model

Control

ALU      ALU      ALU      ALU      ALU      ALU
Thread Blocks are Executed as Warps

- Each thread block is mapped to one or more warps
  - When the thread block size is not a multiple of the warp size, unused threads within the last warp are disabled automatically

- The hardware schedules each warp independently
  - Warps within a thread block can execute independently
Thread and Warp Scheduling

- The processors (streaming multiprocessors) can switch between warps with no apparent overhead.
- Warps with instruction whose inputs are ready are eligible to execute, and will be considered when scheduling.
- When a warp is selected for execution, all (active) threads execute the same instruction.
Filling Warps

- Prefer thread block sizes that result in mostly full warps
  - **Bad:** kernel<<<N, 1>>> ( ... )
  - **Okay:** kernel<<<N / 32, 32>>>( ... )
  - **Better:** kernel<<<N / 128, 128>>>( ... )

- Prefer to have enough threads per block to provide hardware with many warps to switch between
  - This is how the GPU hides memory access latency

- Resource like `__shared__` may constrain threads per block
  - Algorithm and decomposition will establish some preferred amount of shared data and `__shared__` allocation
Control Flow Divergence

- The system automatically handles control flow divergence, conditions in which threads within a warp execute different paths through a kernel.

- Often, this requires that the hardware execute multiple paths through a kernel for a warp
  - For example, both the if clause and the corresponding else clause
COOPERATING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __sycnthreads()
- Asynchronous operation
- Handling errors
- Managing devices
1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

- If radius is 3, then each output element is the sum of 7 input elements:
Implementing Within a Block

- Each thread processes one output element
  - `blockDim.x` elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times
Sharing Data Between Threads

- Terminology: within a block, threads share data via **shared memory**

- Extremely fast on-chip memory, user-managed

- Declare using `__shared__`, allocated per block

- Data is not visible to threads in other blocks
Implementing With Shared Memory

- Cache data in shared memory
  - Read \((\text{blockDim}.x + 2 \times \text{radius})\) input elements from global memory to shared memory
  - Compute \(\text{blockDim}.x\) output elements
  - Write \(\text{blockDim}.x\) output elements to global memory

- Each block needs a halo of radius elements at each boundary
__global__ void stencil_ld(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }
}

Stencil Kernel
```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS; offset <= RADIUS; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
```
Data Race!

- The stencil example will not work...

- Suppose thread 15 reads the halo before thread 0 has fetched it...

```c
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]  
Skiped, threadIdx > RADIUS

Load from temp[19]
**__syncthreads()**

- `void __syncthreads();`

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block
Stencil Kernel

```c
__global__ void stencil_ld(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + radius;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }

  // Synchronize (ensure all the data is available)
  __syncthreads();
}"
```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
MANAGING THE DEVICE

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
Coordinating Host & Device

- Kernel launches are **asynchronous**
  - Control returns to the CPU immediately

- CPU needs to synchronize before consuming the results

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaMemcpy()</td>
<td>Blocks the CPU until the copy is complete</td>
</tr>
<tr>
<td></td>
<td>Copy begins when all preceding CUDA calls have completed</td>
</tr>
<tr>
<td>cudaMemcpyAsync()</td>
<td>Asynchronous, does not block the CPU</td>
</tr>
<tr>
<td>cudaDeviceSynchronize()</td>
<td>Blocks the CPU until all preceding CUDA calls have completed</td>
</tr>
</tbody>
</table>
Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself
  - OR
  - Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:
  ```
  cudaError_t cudaGetLastError(void)
  ```

- Get a string to describe the error:
  ```
  char *cudaGetErrorString(cudaError_t)
  ```

  ```
  printf("%s\n", cudaGetErrorString(cudaGetLastError()));
  ```
Device Management

- Application can query and select GPUs
  
  ```c
  cudaGetDeviceCount(int *count)
  cudaSetDevice(int device)
  cudaGetDevice(int *device)
  cudaGetDeviceProperties(cuDeviceProp *prop, int device)
  ```

- Multiple CPU threads can share a device

- A single CPU thread can manage multiple devices
  
  ```c
  cudaSetDevice(i) to select current device
  cudaMemcpy(...) for peer-to-peer copies
  ```

  † requires OS and device support
Resources

- We skipped some details, you can learn more:
  - CUDA Programming Guide
  - CUDA Zone – tools, training, webinars and more
    http://developer.nvidia.com/cuda
```c
__global__ void copy(float *odata, float *idata, int width, int height)
{
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

    int index = xIndex + width*yIndex;

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
    {
        odata[index+i*width] = idata[index+i*width];
    }
}
```
```c
__global__ void copySharedMem(float *odata, float *idata, int width, int height)
{
    __shared__ float tile[TILE_DIM][TILE_DIM];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

    int index = xIndex + width*yIndex;

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
    {
        if (xIndex < width && yIndex < height)
        {
            tile[threadIdx.y][threadIdx.x] = idata[index];
        }
    }
}
```
__syncthreads();

for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
{
    if (xIndex < height && yIndex < width)
    {
        odata[index] = tile[threadIdx.y][threadIdx.x];
    }
}
void computeTransposeGold(float *gold,
    float *idata, const int size_x, const int size_y)
{
    for (int y = 0; y < size_y; ++y)
    {
        for (int x = 0; x < size_x; ++x)
        {
            gold[(x*size_y)+y] = idata[(y*size_x)+x];
        }
    }
}
Naive GPU Transpose

```c
__global__ void transposeNaive(float *odata,
    float *idata, int width, int height)
{
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

    int index_in  = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
    {
        odata[index_out+i] = idata[index_in+i*width];
    }
}
```
Driver Code for GPU Transpose

dim3 grid(size_x/TILE_DIM, size_y/TILE_DIM);
dim3 threads(TILE_DIM, BLOCK_ROWS);
kernel<<<grid, threads>>>(d_odata, d_idata, size_x, size_y);
Coalesced GPU Transpose with Shared Memory

```c
__global__ void transposeCoalesced(float *odata, 
  float *idata, int width, int height)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int i=0; i<TILE_DIM; i+= BLOCK_ROWS)
  {
    tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
  }
}
```

Nvidia Cuda Programming
Coalesced GPU Transpose with Shared Memory

```c
__syncthreads();

for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
{
    odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
}
```
Coalesced GPUTranspose without Bank Conflicts

```c
__global__ void transposeNoBankConflicts(float *odata, float *idata, int width, int height)
{
    __shared__ float tile[TILE_DIM][TILE_DIM+1];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*width;

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*height;

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
    {
        tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
    }
}
```

Nvidia Cuda Programming
Coalesced GPU Transpose without Bank Conflicts

```c
__syncthreads();

for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
{
    odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
}
```
# GPU Transpose Performance

<table>
<thead>
<tr>
<th></th>
<th>Effective Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2048x2048, GTX 280</td>
</tr>
<tr>
<td>Loop over kernel</td>
<td>Loop in kernel</td>
</tr>
<tr>
<td>Simple Copy</td>
<td>96.9</td>
</tr>
<tr>
<td>Shared Memory Copy</td>
<td>80.9</td>
</tr>
<tr>
<td>Naïve Transpose</td>
<td>2.2</td>
</tr>
<tr>
<td>Coalesced Transpose</td>
<td>16.5</td>
</tr>
<tr>
<td>Bank Conflict Free Transpose</td>
<td>16.6</td>
</tr>
</tbody>
</table>
Other Thoughts

- **UIUC Classes on GPUs:**
  - ECE 408, CS 483 - Applied Parallel Programming - Course director is Wen-Mei Hwu
  - Occasionally 500 level classes are offered on advanced gpu programming

- **Other many core platforms/frameworks:**
  - OpenCL - Framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors
  - Intel MIC (Many Integrated Core) - A coprocessor computer architecture for many core computing

- **Libraries:**
  - Thrust - Parallel algorithms and data structures such as sort, scan, transform, etc
  - cuBLAS - GPU version of BLAS (Basic Linear Algebra Subprograms) library
  - MAGMA - Collection of next gen linear algebra routines
  - Many others can be found at developer.nvidia.com/gpu-accelerated-libraries