



Technology for a better society

Johan Seland

CUDA Programming

16th November - USIT Course Week



Overview

- Getting Started
- Minimal example
 - Power of two
- CUDA Memories
- Intermediate example
 - Gaussian Blur
- Debugging

What is needed?

- CUDA toolkit
 - Includes nvcc – nvidias CUDA compiler
 - Headers and libraries
 - Download and install
- Regular C/C++ compiler
 - gcc
 - Visual Studio 2005 – present
 - Xcode

Quick overview

- 2 APIs
 - CUDA Runtime API
 - CUDA Driver API
- CUDA utility library
 - Wraps CUDA calls
 - But must be compiled manually
- Domain Specific Libraries

Minimal CUDA - example

- The program can be in a single file, with .cu extension
 - .cu files are compiled by nvcc and then your regular compiler
- Only need to include <cuda_runtime.h>
 - Cuda.h for driver API

```
#include <cuda_runtime.h>
```

Basic CUDA - example

- Context created by first CUDA call
 - Here allocation on GPU
 - Remember this when profiling!
- C-like memory management

```
cudaMalloc((void**)&d_res, total_threads * sizeof(int));
```

```
cudaFree(d_res);
```

Basic CUDA - example

- Kernel code
 - `__global__` main entry point from host side
 - Storing results in prev. allocated memory buffer
 - `threadIdx` CUDA variable

```
__global__ void pow2Kernel( unsigned int* results)
{
    results[threadIdx.x] = 1 << threadIdx.x;
}
```

Basic CUDA - example

- Launch kernel by calling a `__global__` function as this

```
//call the kernel to do our calculation  
pow2Kernel<<<num_blocks, num_threads>>>(d_res);
```

- `<<< >>>` nvcc keyword
- `#blocks` and `# threads` specified at launch

Basic CUDA - example

- Transfer data between CPU and GPU
 - Specify in bytes how much to transfer

```
cudaMemcpy(h_res, d_res, total_threads * sizeof(int),  
           cudaMemcpyDeviceToHost);
```

- And in what direction to copy the data

```
cudaMemcpyHostToDevice  
cudaMemcpyDeviceToDevice  
cudaMemcpyDeviceToHost  
cudaMemcpyHostToHost
```

Basic CUDA - example

```
#include <cuda_runtime.h>

__global__ void pow2Kernel( unsigned int* results)
{
    results[threadIdx.x] = 1 << threadIdx.x;
}

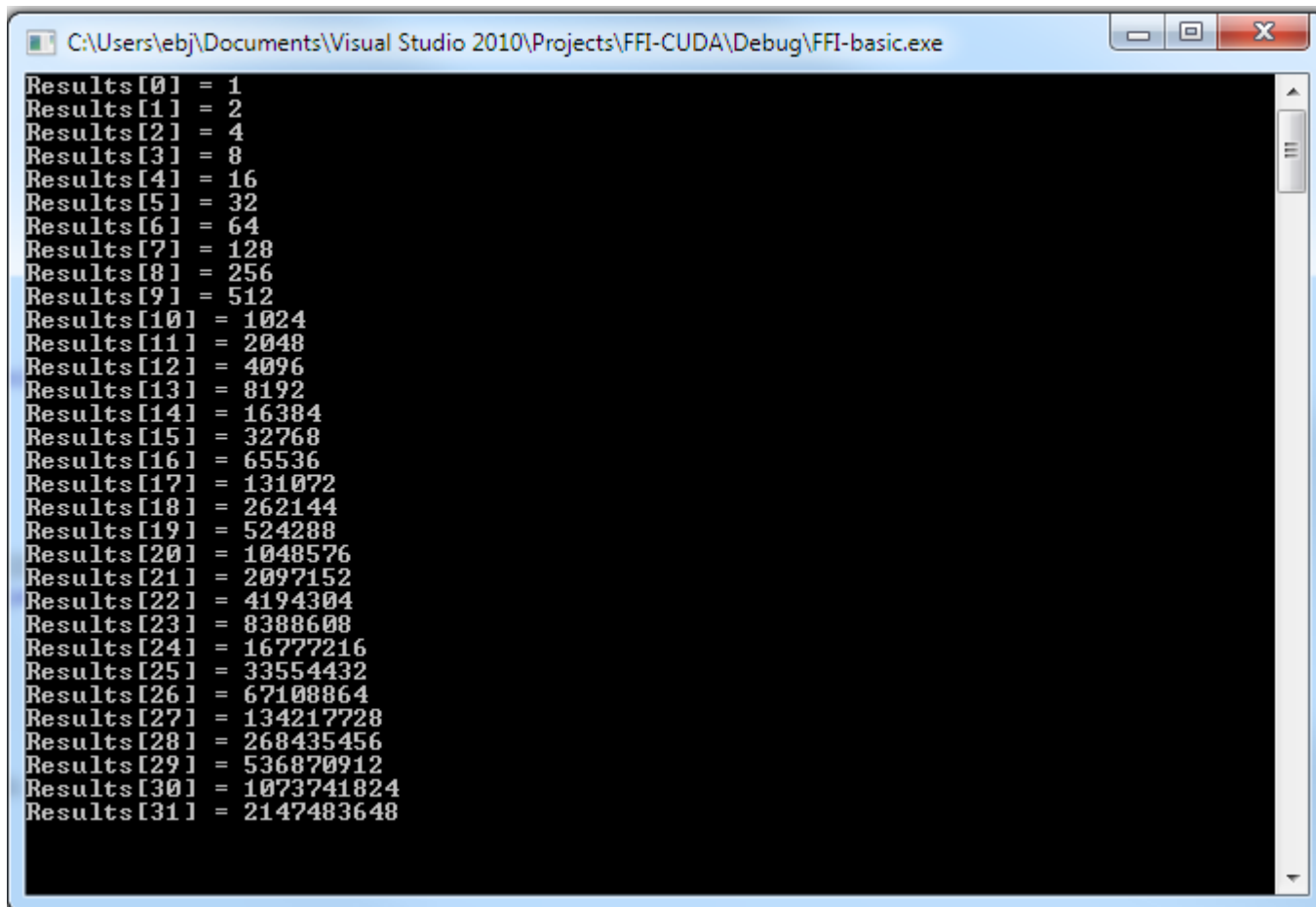
int main(int argc, char *argv[]){
    int num_blocks = 1, num_threads = 32, total_threads = 32;
    //allocate buffer to hold results on device
    unsigned int* d_res;
    cudaMalloc((void**)&d_res, total_threads * sizeof(int));

    //call the kernel to do our calculation
    pow2Kernel<<<num_blocks, num_threads>>>(d_res);

    //get the results back to the host
    unsigned int *h_res = new unsigned int[total_threads]();
    cudaMemcpy(h_res, d_res, total_threads * sizeof(int),
               cudaMemcpyDeviceToHost);

    //clean up memory
    cudaFree(d_results);
    cudaDeviceReset(); //tear down CUDA context
}
```

Basic CUDA - results



```
C:\Users\ebj\Documents\Visual Studio 2010\Projects\FFI-CUDA\Debug\FFI-basic.exe
Results[0] = 1
Results[1] = 2
Results[2] = 4
Results[3] = 8
Results[4] = 16
Results[5] = 32
Results[6] = 64
Results[7] = 128
Results[8] = 256
Results[9] = 512
Results[10] = 1024
Results[11] = 2048
Results[12] = 4096
Results[13] = 8192
Results[14] = 16384
Results[15] = 32768
Results[16] = 65536
Results[17] = 131072
Results[18] = 262144
Results[19] = 524288
Results[20] = 1048576
Results[21] = 2097152
Results[22] = 4194304
Results[23] = 8388608
Results[24] = 16777216
Results[25] = 33554432
Results[26] = 67108864
Results[27] = 134217728
Results[28] = 268435456
Results[29] = 536870912
Results[30] = 1073741824
Results[31] = 2147483648
```

CUDA Memory Types

- GPUs have a bewildering array of memory types
- All are exposed
 - Unlike CPU memories
- Limited caching
- Huge performance gains are possible



Register Memory

- Local to each stream processor
- Super fast
- Limited
- Limits #blocks per SM

Global Memory

- Global GPU Memory
- Same for all threads/blocks
- Reads one cache-line at the time
- Slowest memory available
 - Aside from system
- Coalesced reads
 - All threads in warp reading consecutive addresses
- Automatic cache on newer devices



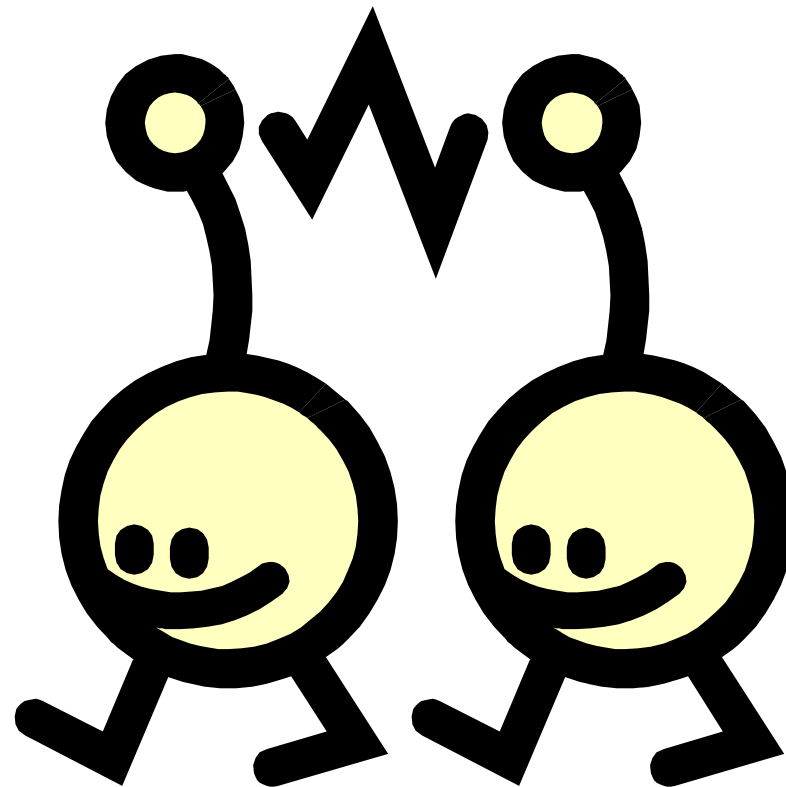
Local Memory

- AKA - Private Global Memory
- Only accessible from specific thread
- Commonly used for register spilling
- Automatically allocated



Shared memory

- Shared between threads in one block
- Size specified at run-time before kernel call
- Dedicated synchronization primitive
- Limited size available
- Limits #blocks per SM
- Can give HUGE performance benefits



Constant Memory

- Same for all threads/blocks
- Only updated from host
- Limited size
- Special cache for global memory
- Nice for:
 - Precomputed tables
 - Parameter objects



Texture Memory

- Specialised cache for global memory
- Reads are 32 bytes
- Optimised for 2D locality
- Addressing done by fixed function units outside kernel
- Can convert 8-/16-bit integer to 32-bit floats in range $[0.0, 1.0]$ / $[-1.0, 1.0]$
- Can give interpolation for free



Page Locked Host Memory

- CPU RAM allocated by `cudaHostAlloc()`
- Higher bandwidth
- Copy to GPU RAM concurrently with kernel execution
- Can be mapped into address space of device
- Very useful for "streaming" applications

CUDA Memory Types - Recap

Memory Type	Visibility	Writeability	Speed	Note
Register	Thread	Read-Write	Fast	
Global Memory	Grid	Read-Write	Slow	
Local Memory	Thread	Read-Write	Slow	
Shared Memory	Block	Read-Write	Medium - Fast	Blockwise sync-primitive
Constant	Grid	Read	Fast	Cached, very limited
Texture Memory	Grid	Read	Fast	Spatially cached, expensive to build
Page-Locked	CPU	Read-Write	Varies	

Don't Panic!

- You don't NEED to use all the memories
 - They are easy optimizations
- Syntax is simple
- Shared memory is crucial!



Intermediate CUDA – Gaussian Blur

- Mask applied to each pixel
- Example demonstrates:
 - Shared memory
 - Global memory
 - Constant memory
- Overlap between kernel areas



Intermediate CUDA – example

- Gaussian blur
 - 2D filter per pixel
 - 5x5 kernel
 - Bandwidth heavy
 - 25 memory fetches
 - Alu heavy
 - 25 div
 - 25 mul
 - 25 add

1	4	6	4	1
4	16	24	16	4
6	24	36	24	6
4	16	24	16	4
1	4	6	4	1

Parallelization Strategy

- Split image into tiles
 - One multiproc per tile
 - One thread per pixel
- Read tile into shared mem
- Read border into shared mem



Intermediate CUDA – example

- Optimise
 - Shared memory
 - 1 memory fetch per pixel + fetches for border
- Initially
 - $16 * 16 * 25 = 6400$ global memory fetches
- Optimised
 - $16 * 16 + 8 * 18 = 400$ global memory fetches
- Don't be afraid to split things up!

Global Addressing

- Pixels stored in cont. array
- Index of pixel (x, y)

```
int getGlobalPixelPos( int x, int y ) {  
    return y * width + x;  
}
```

- Global index of thread

```
Int2 globalIdx;  
  
globalIdx.x = blockIdx.x * blockDim.x + threadIdx.x;  
globalIdx.y = blockIdx.y * blockDim.y + threadIdx.y;
```



Local Addressing

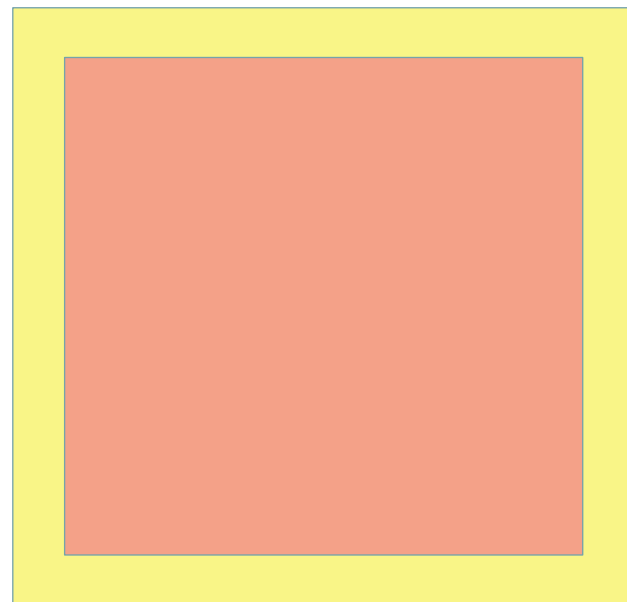
- Blocksize: 16 x 16
- Tilesize: 20 x 20
- Allow negative indices

```
getLocalPixelPos( int x, int y ) {  
    int edge = radius/2;  
    return (y + edge) * tileWidth + x + edge;  
}
```

- Local index of thread

```
int2 localIndex;  
  
localIndex.x = threadIdx.x;  
localIndex.y = threadIdx.y;
```

(-2,-2)



(17,17)

Using Shared Memory

- In the kernel

```
extern __shared__ float3 sdata[];
```

- When calling kernel

```
dim3 block(16, 16, 1); int edge = 2;
dim3 grid(width/block.x, height/block.y, 1);
int tileWidth = block.x + edge*2

int shared_size = tileWidth * tileWidth * sizeof(float3);

ourKernel<<<grid, block, shared_size>>>(...);
```

Populating Shared Memory

```
// Load main block of memory
int globalPixelPos = imageParams.getGlobalPixelPos( globalIndex.x, globalIndex.y );
int localPixelPos = imageParams.getLocalPixelPos( threadIdx.x, threadIdx.y );

shared_mem[localPixelPos] = intToRgb( input_image[globalPixelPos] );

syncthreads()
```

Populating Shared Memory

- The tile borders must be handled separately

```
// Load upper and lower
if ( threadIdx.y < 2*edge ) {
    int offset = threadIdx.y < edge ? -edge : blockDim.y - edge;

    int globalPixelPos = getGlobalPixelPos( globalIndex.x, globalIndex.y + offset );
    int localPixelPos = getLocalPixelPos( threadIdx.x, threadIdx.y + offset );

    shared_mem[localPixelPos] = intToRgb( input_image[globalPixelPos] );
}
```

- Remember to synchronize at the end

```
syncthreads()
```

Intermediate CUDA – example

- Optimise
 - Pre-divide
 - 25 memory fetches
 - 25 MAD per pixel
 - Split into two passes
 - Horizontal
 - Vertical
 - 10 memory fetches
 - 10 mad

1/256	4/256	6/256	4/256	1/256
4/256	16/256	24/256	16/256	4/256
6/256	24/256	36/256	24/256	6/256
4/256	16/256	24/256	16/256	4/256
1/256	4/256	6/256	4/256	1/256

1/16	4/16	6/16	4/16	1/16
------	------	------	------	------

Using constant memory

- Hardcode

```
__device__ __constant__ float our_constant = 1.f;
```

- Dynamic

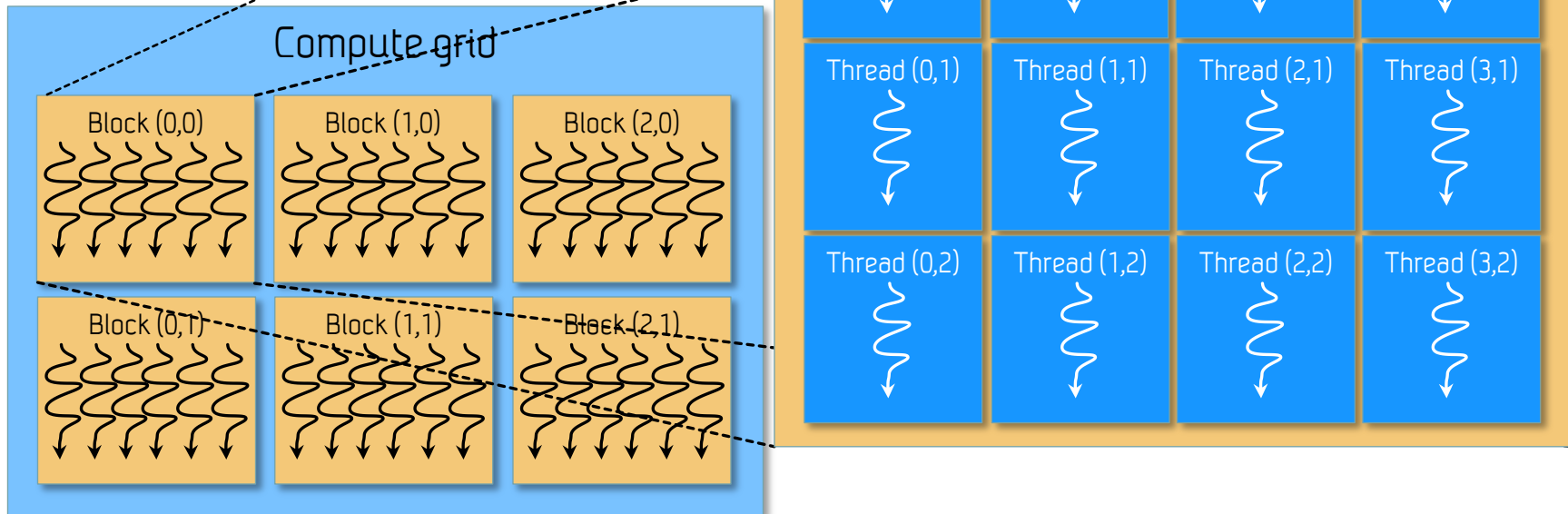
```
__device__ __constant__ float our_constants[3];
```

```
cudaMemcpyToSymbol(our_constants,  
    host_constants, 3 * sizeof(float));
```

Intermediate CUDA – example

- Synchronisation
 - Blocks execute in random order
 - Switching in and out depending if ready/waiting for memory fetch
 - Warps execute in random order
 - No guarantee all memory fetches completed for all threads after a fetch
 - Need to manually synchronise in kernel
 - Will only synchronise threads in block, not in grid!

GPU compute grids



- Execution invoked by CPU over a **compute grid**
 - Compute grid subdivided into a set of **blocks**
 - Blocks contains a set of **threads**, which can access block-level **shared memory**
- All threads in the compute grid run **the same program**
 - But with **individual data** and **individual code flow**

Vertical Pass

```
float3 pixel = make_float3( 0.0f );

for(int i = -edge; i <= edge; ++i){
    int pixelPos = getLocalPixelPos( threadIdx.x,
                                     threadIdx.y + i );
    pixel += gauss_const[i + edge] *
            shared_mem[pixelPos];
}

syncthreads();
shared_mem[localPixelPos] = pixel;
syncthreads();
```

Vertical Pass – Left/Right Edge

```
float3 pixel = make_float3(0.0f, 0.0f, 0.0f);
if(threadIdx.x < 2 * edge) {

    int delta = threadIdx.x < edge ? -edge :
                                   blockDim.x - edge;
    for(int i = -edge; i <= edge; ++i) {
        int pixelPos = getLocalPixelPos( threadIdx.x + delta,
                                           threadIdx.y + i );

        pixel += gauss_const[i+edge] *
                shared_mem[smIdx.y + smIdx.x + i];
    }
    shared_mem[smIdx.y + smIdx.x] = pixel;
}
syncthreads();
```

Horizontal Pass

```
for ( int i = -edge; i <= edge; ++i ) {  
    int pixelPos = getLocalPixelPos( threadIdx.x + i ,  
                                     threadIdx.y );  
  
    pixel += gauss_const[ i + edge ] * shared_mem[ pixelPos ];  
}  
  
global_mem[ globalPixelPos ] = pixel;
```

Putting it all together

```
__global__ void cudaTwoPass( unsigned int const * input_image,
                             ImageFilterParams imageParams,
                             unsigned int* output_image ) {

    loadInteriorBlock( input_image, imageParams );
    syncthreads();

    verticalPass( imageParams );
    syncthreads();

    float3 pixel = horizontalPass( imageParams );

    unsigned int output_value = rgbToInt( pixel );

    output_image[globalPixelPos] = output_value;
}
```

Intermediate CUDA – example

- DEMO

Debugging

- Windows
 - Parallel Nsight 2.0
- Linux / MAC OS X
 - CUDA-gdb
- All platforms
 - Visual Profiler
 - [cuPrintf](#)
 - Registered developers only
- [Intermediate](#)

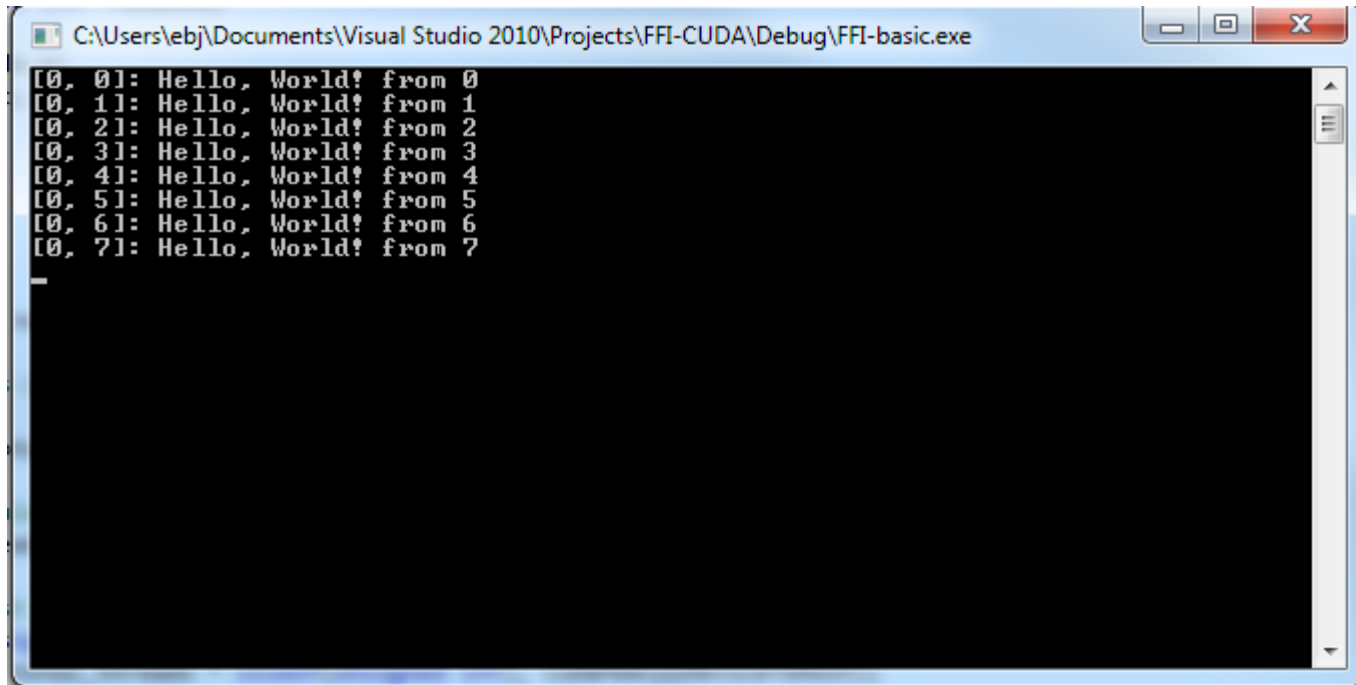
cuPrintf

- Adds printf-like support to CUDA
 - Only prints after kernel invocation
- Place cuPrintf.cu in source folder, and #include in kernel.cu
- Add the following to source
- In kernel, call

```
cudaPrintfInit();  
//launch kernel  
cudaPrintfDisplay(stdout/err, true);  
cudaPrintfEnd();
```

```
cuPrintf("Hello, World! from %d\n", threadIdx.x);
```

cuPrintf



```
C:\Users\ebj\Documents\Visual Studio 2010\Projects\FFI-CUDA\Debug\FFI-basic.exe
[0, 0]: Hello, World! from 0
[0, 1]: Hello, World! from 1
[0, 2]: Hello, World! from 2
[0, 3]: Hello, World! from 3
[0, 4]: Hello, World! from 4
[0, 5]: Hello, World! from 5
[0, 6]: Hello, World! from 6
[0, 7]: Hello, World! from 7
```

- [Intermediate](#)