Introduction to CUDA C

A Tutorial

Jayant Apte, ASPITRG
Introduction to CUDA C
A Tutorial

Jayant Apte, ASPITRG
Parallel Computing

*Why do we care?*

*Why build parallel systems?*

*How about we automatically convert serial programs to parallel programs?*
Parallel Computing

Why do we care?

Why build parallel systems?

How about we automatically convert serial programs to parallel programs?
• Very limited success with converting C/C++ code to parallel code.

• Step back. Rework the algorithm.
Parallel Computing

Why do we care?

Why build parallel systems?

How about we automatically convert serial programs to parallel programs?
<table>
<thead>
<tr>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Data</td>
</tr>
<tr>
<td>Single</td>
<td><strong>SISD</strong></td>
</tr>
<tr>
<td>Typical thread</td>
<td><strong>SIMD</strong></td>
</tr>
<tr>
<td>Single</td>
<td>Vector processors</td>
</tr>
<tr>
<td>Multiple</td>
<td>GPUs</td>
</tr>
<tr>
<td>Instruction</td>
<td>SSE instructions</td>
</tr>
<tr>
<td>Multiple</td>
<td><strong>MISD</strong></td>
</tr>
<tr>
<td>Rare</td>
<td><strong>MIMD</strong></td>
</tr>
<tr>
<td>Possibly set of</td>
<td>Cluster of computers</td>
</tr>
<tr>
<td>Filters</td>
<td></td>
</tr>
</tbody>
</table>
Comparison with CPU

- Cache sizes
- Floating point capability
Comparison with CPU

- Cache sizes
- Floating point capability
Comparison with CPU

- Cache sizes
- Floating point capability
CUDA

Compute Unified Device Architecture
CUDA

Compute Unified Device Architecture

<table>
<thead>
<tr>
<th>GPU Computing Applications</th>
<th>Libraries and Middleware</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C++</td>
</tr>
<tr>
<td>Fortran</td>
<td>Direct Compute</td>
</tr>
<tr>
<td>Lower Python versions</td>
<td>OpenMP</td>
</tr>
<tr>
<td>Direct Math</td>
<td>OpenCL</td>
</tr>
<tr>
<td>Direct3D</td>
<td>CUDA</td>
</tr>
<tr>
<td>DirectCompute</td>
<td>Bathymetry</td>
</tr>
<tr>
<td>DirectCompute</td>
<td>PETSc</td>
</tr>
<tr>
<td>DirectCompute</td>
<td>FLAME</td>
</tr>
</tbody>
</table>

NVIDIA GPU with the CUDA Parallel Computing Architecture

- Fermi Architecture (compute.capabilities.2.0)
  - GeForce 460 Series
  - Quadro Fermi Series
  - Tesla 20 Series

- Tesla Architecture (compute.capabilities.1.3)
  - GeForce 280 Series
  - Quadro FX Series
  - QuadroFX Series
  - Tesla 10 Series

Professional
- Workstation Computing
CUDA

Compute Unified Device Architecture
CUDA C : Execution model

CUDA C = C + Some More keywords
C Program
Sequential Execution

Serial code

Parallel kernel
Kernel0<<<>>>(())

Serial code

Parallel kernel
Kernel1<<<>>>(())

Host

Device

Grid 0

Block (0, 0)

Block (1, 0)

Block (2, 0)

Block (0, 1)

Block (1, 1)

Block (2, 1)

Host

Device

Grid 1

Block (0, 0)

Block (1, 0)

Block (0, 1)

Block (1, 1)

Block (0, 2)

Block (1, 2)

Serial code executes on the host while parallel code executes on the device.
CUDA C : Execution model

CUDA C = C + Some More keywords
What Do I need to get started?**

- A CUDA-enabled graphics processor
- An NVIDIA device driver
- A CUDA development toolkit
- A standard C compiler

**Or you can ask Dr. Walsh to give you an account on aspitrg2**
Let's look at some code.
Hello World?

A Kernel Call

#include "../common/book.h"
__global__ void kernel( void ) {

}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
#include "book.h"
__device__ int addem( int a, int b ) {
   return a + b;
}

__global__ void add( int a, int b, int *c ) {
   *c = addem( 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 );
   HANDLE_ERROR( cudaFree( dev_c ) );
   return 0;
}
#include "book.h"

tint 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" );
        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: %ld\n", prop.multiProcessorCount );
        printf( "Shared mem per mp: %ld\n", prop.sharedMemPerBlock );
        printf( "Registers per mp: %ld\n", prop.regPerBlock );
        printf( "Threads in warp: %d\n", prop.warpSize );
        printf( "Max threads per block: %ld\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" );
    }
}
Let's look at some code

Vector Addition

```c
#include "block.h"

void vector_addition() {
    // Code for vector addition
}
```

Vector Additions Parallel...

```c
// Parallel code for vector addition
```
Vector Addition

#include "book.h"
#define N 10
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;
}