First Program

Learning CUDA to Solve Scientific Problems.

Miguel Cárdenas Montes

Centro de Investigaciones Energéticas Medioambientales y Tecnológicas,
Madrid, Spain
miguel.cardenas@ciemat.es

2010

Table of Contents

Objectives

- To create a first and basic program.
- To know and to apply the more common compilation options.

Technical Issues

- To be able to create a basic program.
- Compilation options.

First and Basic Program

First Program I

First Program

```
#include <stdio.h>
#include <cuda.h>
// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{

}
// main routine that executes on the host
int main(void)
{
  // Pointer to host & device arrays
  // Number of elements in arrays
  // Allocate array on host
  // Allocate array on device
  // Initialize host array and copy it to CUDA device
  // Do calculation on device:
  // Retrieve result from device and store it in host array
  // Print results
  // Cleanup
}
```


First Program II

Kernel definition

```
// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

- Kernel definition
- Multiplication operation (square)
- Two variables as argument

First Program III

Variable definition

```
// main routine that executes on the host
int main(void)
{
  float *a_h, *a_d;  // Pointer to host & device arrays
  const int N = 10;  // Number of elements in arrays
.
.
.
}
```

- main definition.
- Declaration of the array *a* both: for host part code

  `a_h`

  and for device part code

  `a_d`

First Program IV

Memory allocation

```
size_t size = N * sizeof(float);
a_h = (float *)malloc(size);        // Allocate array on host
cudaMalloc((void **) &a_d, size);   // Allocate array on device
```

- Allocation of the variables at the memories, both: on host and on device.

First Program V

Data movement

```
 // Initialize host array and copy it to CUDA device
for (int i=0; i<N; i++) a_h[i] = (float)i;
cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
```

- The array on the host is initialize.
- Later, the array is transferred from the memory host to memory device.

First Program VI

Kernel invocation

```
// Do calculation on device:
int block_size = 4;
int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
square_array <<< n_blocks, block_size >>> (a_d, N);
```

- Calculation of grid block size.
- Kernel invocation.

First Program VII

Data movement

```
// Retrieve result from device and store it in host array
cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
```

- And finally the results are retrieved from the memory device to the host.

First Program VIII

Memory release

```
// Print results
for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
// Cleanup
free(a_h); cudaFree(a_d);
```

- Print the result and cleanup of the memory (on host and on device).

Compilation Options

Compilation Options I

- `nvcc <filename>.cu -o <executable>`
  - Builds release mode.
  - Most usual compilation.
- `nvcc g <filename>.cu`
  - Builds debug (device) mode.
  - Can debug host code but not device code (runs on GPU).

Compilation Options II

- `nvcc deviceemu <filename>.cu`
  - Builds device emulation mode.
  - All code runs on CPU, but no debug symbols.
- `nvcc deviceemu g <filename>.cu`
  - Builds debug device emulation mode.
  - All code runs on CPU, with debug symbols.
  - Debug using gdb or other linux debugger.

Persistence of global memory

Persistence on global memory.

Source code. Body

```
int main(void)
{
  float *a_h, *a_d;  // Pointer to host & device arrays, b & c omited

  const int N = 1000;  // Number of elements in arrays
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);      // Allocate array on host, b & c omited
  cudaMalloc((void **) &a_d, size); // Allocate array on device, b & c omited

  // Initialize host array and copy it to CUDA device
  for (int i=0; i<N; i++) a_h[i] = (float)i;

  cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);

  // Do calculation on device:
  int block_size = 256;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  kernel1 <<< n_blocks, block_size >>> (a_d, b_d, N);

  cudaThreadSynchronize( );

  kernel2 <<< n_blocks, block_size >>> (b_d, c_d, N);

  // Retrieve result from device and store it in host array
  // Print results, array c
  // Cleanup
}
```


Persistence on global memory II

Source code. Kernels

```
__global__ void kernel1(float *a, float *b, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) b[idx] = a[idx] + 1.0 ;
}

__global__ void kernel2( float *b, float *c, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) c[idx] = b[idx] + 1.0 ;
}
```


Hardware Specifications

Knowing the Hardware Specifications I

Source code

```
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:  " );
        if (prop.deviceOverlap)
            printf( "Enabled\n" );
        else
            printf( "Disabled\n");
        printf( "Kernel execution timeout :  " );
        if (prop.kernelExecTimeoutEnabled)
            printf( "Enabled\n" );
        else
            printf( "Disabled\n" );
.
.
.
    }
}
```

Knowing the Hardware Specifications II

Source code

```
int main(void)
{
.
.
.
        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 );

        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" );
    }
}
```


Thanks

Thanks

Questions?

More questions?

EUFORIA