CUDA – First Programs

“Hello, world” is traditionally the first program we write. We can do the same for CUDA. Here it is:

In file `hello.cu`:

```
#include "stdio.h"

int main()
{
        printf("Hello, world\n");
        return 0;
}
```

On our Tesla machine, you can compile and this with:

```
$ nvcc hello.cu
$ ./a.out
```

You can change the output file name with the –o flag: `nvcc –o hello hello.cu`

If you edit your .bashrc file you can also add your current directory to your path if you don’t want to have to type the preceding . all of the time, which refers to the current working directory. Add

```
export PATH=$PATH:.
```

To the .bashrc file. Some would recommend not doing this for security purposes.

You might consider this program to be cheating, since it doesn’t really use any CUDA functionality. Everything runs on the host. However, the point is that CUDA C programs can do everything a regular C program can do.

Here is a slightly more interesting (but inefficient and only useful as an example) program that adds two numbers together using a kernel function:

```
#include "stdio.h"

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

int main()
{
 int a,b,c;
 int *dev_c;

 a=3;
 b=4;
 cudaMalloc((void**)&dev_c, sizeof(int));
 add<<<1,1>>>(a,b,dev_c);
 cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
 printf("%d + %d is %d\n", a, b, c);
 cudaFree(dev_c);
 return 0;
}
```

To do in class: walk through program; show similar program in straight C and it runs much faster! Why?

cudaMalloc returns cudaSuccess if it was successful; could check to ensure that the program will run correctly.

Example: Summing Vectors

This is a simple problem. Given two vectors (i.e. arrays), we would like to add them together in a third array. For example:

A = {0, 2, 4, 6, 8}

B = {1, 1, 2, 2, 1}

Then A + B =

C = {1, 3, 6, 8, 9}

In this example the array is 5 elements long, so our approach will be to create 5 different threads. The first thread is responsible for computing C[0] = A[0] + B[0]. The second thread is responsible for computing C[1] = A[1] + B[1], and so forth.

Here is how we can do this with traditional C code:

```
#include "stdio.h"

#define N 10

void add(int *a, int *b, int *c)
{
        int tID = 0;
        while (tID < N)
        {
                c[tID] = a[tID] + b[tID];
                tID += 1;
        }
}

int main()
{
        int a[N], b[N], c[N];
        // Fill Arrays
        for (int i = 0; i < N; i++)
        {
                a[i] = i,
                b[i] = 1;
        }
        add (a, b, c);
        for (int i = 0; i < N; i++)
        {
                printf("%d + %d = %d\n", a[i], b[i], c[i]);
        }
        return 0;
}
```

This is a rather roundabout way to add two arrays – our reason is because this will translate a little nicer to the CUDA version. To compile and run it, we have to use g++ (since it uses some C++ style notations that don't work in C). Here is the CUDA version:

```
#include "stdio.h"
#define N 10

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

int main()
{
        int a[N], b[N], c[N];
        int *dev_a, *dev_b, *dev_c;

        cudaMalloc((void **) &dev_a, N*sizeof(int));
        cudaMalloc((void **) &dev_b, N*sizeof(int));
        cudaMalloc((void **) &dev_c, N*sizeof(int));

        // Fill Arrays
        for (int i = 0; i < N; i++)
        {
                a[i] = i,
                b[i] = 1;
        }

        cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);

        add<<<N,1>>>(dev_a, dev_b, dev_c);

        cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);

        for (int i = 0; i < N; i++)
        {
                printf("%d + %d = %d\n", a[i], b[i], c[i]);
        }
        return 0;
}
```

blockIDx.x gives us the Block ID, which ranges from 0 to N-1. What if we used `add<<<1,N>>>` instead? Then we can access by the ThreadID which is the variable threadIDx.x.

As another example, let's add two 2D arrays. We can define a 2D array of ints as follows:

int c[2][3];

The following code illustrates how the 2D array is laid out in memory:

```
for (int i=0; i < 2; i++)
  for (int j=0; j< 3; j++)
    printf("[%d][%d] at %ld\n",i,j,&c[i][j]);
```

Output:

```
[0][0] at 140733933298160
[0][1] at 140733933298164
[0][2] at 140733933298168
[1][0] at 140733933298172
[1][1] at 140733933298176
[1][2] at 140733933298180
```

We can see that we have a layout where the next cell in the j dimension occupies the next sequential integer in memory, where an int is 4 bytes:

c[0][0] at &c	c[0][1] at &c + 4	c[0][2] at &c + 8
c[1][0] at &c + 12	c[1][1] at &c + 16	c[1][2] at &c + 20

In general, the address of a cell can be computed by:

```
&c + [(sizeof(int) * sizeof-j-dimension * i] + (sizeof(int)) * j
```

In our example the size of the j dimension is 3. For example, the cell at c[1][1] would be combined as the base address + (4*3*1) + (4*1) = &c+16.

C will do the addressing for us if we use the array notation, so if `INDEX=i*WIDTH + J`
then we can access the element via: `c[INDEX]`

CUDA requires we allocate memory as a one-dimensional array, so we can use the mapping above to a 2D array.

To make the mapping a little easier in the kernel function we can declare the blocks to be in a grid that is the same dimensions as the 2D array. This will create variables blockIdx.x and blockIdx.y that correspond to the width and height of the array.

```
#include "stdio.h"
#define COLUMNS 3
#define ROWS 2

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

int main()
{
        int a[ROWS][COLUMNS], b[ROWS][COLUMNS], c[ROWS][COLUMNS];
        int *dev_a, *dev_b, *dev_c;

        cudaMalloc((void **) &dev_a, ROWS*COLUMNS*sizeof(int));
        cudaMalloc((void **) &dev_b, ROWS*COLUMNS*sizeof(int));
        cudaMalloc((void **) &dev_c, ROWS*COLUMNS*sizeof(int));

        for (int y = 0; y < ROWS; y++)          // Fill Arrays
         for (int x = 0; x < COLUMNS; x++)
         {
                a[y][x] = x;
                b[y][x] = y;
         }

        cudaMemcpy(dev_a, a, ROWS*COLUMNS*sizeof(int),
cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, b, ROWS*COLUMNS*sizeof(int),
cudaMemcpyHostToDevice);

        dim3 grid(COLUMNS,ROWS);
        add<<<grid,1>>>(dev_a, dev_b, dev_c);

        cudaMemcpy(c, dev_c, ROWS*COLUMNS*sizeof(int),
cudaMemcpyDeviceToHost);

        for (int y = 0; y < ROWS; y++)          // Output Arrays
        {
         for (int x = 0; x < COLUMNS; x++)
         {
                printf("[%d][%d]=%d ",y,x,c[y][x]);
         }
         printf("\n");
        }
        return 0;
}
```