Heterogeneous Parallel Programming: Dive into the World of CUDA.

A previous article in this series 'Introducing NVIDIAs CUDA' covered the basics of the NVIDIA CUDA device architecture. This article covers parallel programming using CUDA C with sequential and parallel implementations of a vector addition program.

Parallel programming and general-purpose GPU computing are some of the hottest trends in computer science today due to the decreased prices of multi-core systems and the increase in compute efficiency. Various parallel programming languages like OpenCL and CUDA have been developed and evaluated over the years. This article will cover the basics of CUDA C, invoking kernels, threads and blocks with a vector addition program and it aims to give you an insight into beginning programming on your CUDA device.

A few important points before we begin. To use CUDA on your system, you will need to have the following installed:

1. CUDA-capable GPU hardware
2. A supported version of Linux with a GCC compiler and toolchain
3. NVIDIA CUDA toolkit and drivers

It is presumed that if you already have a CUDA device within your system, you probably have the latest toolkit and drivers installed and configured correctly. In case you do not have the NVIDIA CUDA drivers configured or you have recently upgraded your hardware with a CUDA device, you could follow the simple steps given below to configure your device.

1. Download the toolkit from the NVIDIA website, available at no cost from: *http://www.nvidia.com/content/cuda/cuda-downloads.html*. Select the correct product release depending on your operating system preferences. This download contains an all-in-one package, which includes the CUDA toolkit, SDK code samples and the required drivers. After downloading, follow the steps in the NVIDIA guide to install the drivers, CUDA samples and the toolkit, at: *http://developer.download.nvidia.com/compute/cuda/5_0/rel/docs/CUDA_Getting_Started_Guide_For_Linux.pdf*. This guide will help you set up the complete environment on the system.

In this article, I will cover the serial and parallel versions of a vector addition program. Once you have understood the basics of the parallel vector addition program, you could use the concepts pretty well in parallelising other algorithms as per your requirements.

First, let's write a simple C program to perform vector addition of two arrays. Open your favourite editor and write a simple vector addition code that looks like what's shown below:

```
#include<stdio.h>

static const int N=100;
```

```
//Add the vectors and store result in vector C
void vector_add(int *a,int *b, int *c)
{
     int i=0;
     for (i=1;i<=N;i++)
    {
         c[i]=a[i]+b[i];
         }
     }

     int main()

    {

          int a[N], b[N], c[N];
          int i=0;

         //Initialize the vectors with values from 1 to 100
and its double in another array
          {
             a[i]=i;
             b[i]=2*i;
          }

             //Call function vector_add to display the result
             vector_add(a,b,c);

              //Print the resultant array.
              for (i=1;i<=N;i++)
              {
                 printf("%d %d %d\n\n", a[i],b[i],c[i] );
                 }

             system("pause");
             return 0;

          }
```

This very simple serial vector addition program creates two arrays of integer values, and adds them using the *vector_add* function.

Compile the code using the following command:

```
gcc sequential_vector.c -osequential
```

Run it using the command given below:

```
./sequential
```

In the above program, the processor runs each task sequentially, one after the other. Looping in the above program works sequentially as it starts with the first index and computes consequentially till the last index, and then

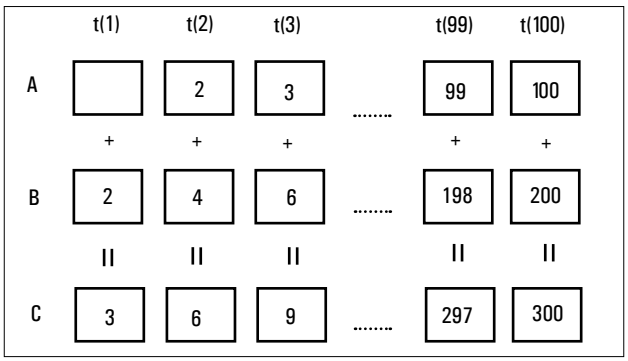

Figure 1: Sequential vector addition

exits the program. It is a single-threaded execution. The figure below would make the serial execution concept clear, where *t(x)* represents the time slot of execution.

Now, CUDA gives us the functionality to perform the same operation in parallel. What it does essentially is offload the data parallel sections to the GPU device and send the result back after computation. In what follows, you will get an insight into launching kernels, writing a device and host code, and performing the same serial vector addition program given above, in parallel.

Here is a simple vector addition code in CUDA.

```
#include <cuda.h>
#include<stdio.h>
#define N 100
#define numThread 1 // in this example we keep one thread in
one block
#define numBlock 100 // in this example we use 100 blocks

__global__ void vector_add( int *a, int *b, int *c ) {

   // keep track of the index
   int tid = blockIdx.x;

   while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid = tid+ numBlock; // shift by the total number of
blocks, i.e. 100 in our case
     }
}

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

    // allocate the memory on the CPU
   a = (int*)malloc( N * sizeof(int) );
   b = (int*)malloc( N * sizeof(int) );
   c = (int*)malloc( N * sizeof(int) );

  //Initialize the vectors with values from 1 to 100 and its
double in another array
 for (int i=0; i<N; i++) {
```

```
        a[i] = i;
        b[i] = 2 * i;
    }

    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) ) ;
   cudaMalloc( (void**)&dev_b, N * sizeof(int) );
cudaMalloc( (void**)&dev_c, N * sizeof(int) );

     // copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy( dev_a, a, N * sizeof(int),cudaMemcpyHostToDevi
ce );
    cudaMemcpy( dev_b, b, N * sizeof(int),cudaMemcpyHostToDevice
);

    vector_add<<<numBlock,numThread>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( c, dev_c, N * sizeof(int),cudaMemcpyDeviceToHost
) ;

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

     // free the memory we allocated on the CPU
    free( a );
    free( b );
    free( c );

    // free the memory we allocated on the GPU
    cudaFree( dev_a ) ;
    cudaFree( dev_b ) ;
     cudaFree( dev_c);

    return 0;
}
```

Let's begin with analysing each part of the code, and then compile the code to get our results.

From the previous article (Part 1 of this series), you know that CUDA programs execute in two places—the host (your CPU) and the device (GPU). You might be a bit surprised by the fact that writing the device code is much simpler than writing the CPU host code. Hence, let's begin with analysing the device code first.

Since we have 100 array values in this code, to simplify things, let's have 100 blocks (kernels) running simultaneously, where each kernel runs a single thread. Hence, let's set numThread to 1 and numBlock to 100, and use these variables later while calling the device from the host.

Device code:

```
__global__ void add( int *a, int *b, int *c ) {

     // keep track of the index
    int tid = blockIdx.x;

    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid = tid+ numBlock; // shift by the total number of
blocks, i.e. 100 in our case
    }
}
```

As shown above, add a *__global__* qualifier to the function *vector_add*. Notice that there are very few changes in the function *vector_add* of the serial and parallel sections. The *__global__* qualifier indicates that this is a device function that would be called from the host.

blockIdx is a built-in CUDA runtime variable, which is a three-component vector to identify threads in a one, two and three dimension index. Imagine a block as a 3-D matrix and to access the different components in this vector, use *blockIdx.x*, *blockIdx.y* and *blockIdx.z*. In this code, we will be using 100 blocks with a single thread on every grid, which will be seen while we analyse the host code, and hence we use only *blockIdx.x* which returns the current block number.

The condition *while tid<N* checks that the bounds for array computation have not been reached and computes the array sum taking the block value as an index, i.e., *tid*.

Add *numBlock* to the *tid* value, to shift the index by the number of blocks, as each block would be computing just one array index, and we have 100 blocks for 100 array indexes. This explanation pretty much sums up the device code for the program.

Now move on to the host code, which prepares the GPU for execution and invokes the kernel. It works by allocating memory to the GPU and CPU, transfers the input vectors to the GPU, launches the kernel and transfers the result back to the host (CPU).

```
int main( void ) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the CPU
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );

    // fill the arrays 'a' and 'b' on the CPU
   for (int i=0; i<N; i++) {
        a[i] = i;
       b[i] = 2 * i;
    }

    // allocate the memory on the GPU
   cudaMalloc( (void**)&dev_a, N * sizeof(int) ) ;
```

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

   // copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy( dev_a, a, N * sizeof(int),cudaMemcpyHostToDevi
ce );
  cudaMemcpy( dev_b, b, N * sizeof(int),cudaMemcpyHostToDevice );
```

As in the above code, similar to the allocation in C, variables a, b and c are allocated memory on the CPU. *cudaMalloc()* is a standard sub-routine of the CUDA API to allocate memory on the device. It works similar to the *C malloc()* function we used earlier.

Since you cannot modify the memory allocated to the device from the host directly in CUDA, *cudaMemcpy()* is used to transfer data to the device. This method takes a pointer to the local memory, a pointer to the GPU memory being copied to, the number of bytes that will be copied, and a flag which determines the direction of the memory transfer, respectively.

```
vector_add<<<numBlock,numThread>>>( dev_a, dev_b, dev_c );
```

This line is a call to the device kernel from the host to execute the function on the device. It is similar to the serial function call with some additional code. Blocks are organised in three dimensional grids and threads are organised in three dimensional blocks. numBlock and numThread are passed as arguments to let the device know about the structure to be adopted for computation.

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

This copies the resultant data back from the GPU to the host. As you can see, it is similar to the *cudaMemcpy()* we used above, with the last variable being changed to *cudaMemcpyDeviceToHost* to indicate data transfer is between device to host.

The rest of the code is pretty much self-explanatory, except that you use *cudaFree()* to free the memory allocated to the GPU.

Now that you understand the code well, compile the code to verify your results.

Save the code and *parallel_vector.cu*, and type the following command in the terminal:

```
nvcc parallel_vector.c  -o parallel
```

Now, to execute the code, run the following command:

```
./parallel
```

If you have followed this guide correctly, it will print the vectors 'a' and 'b' along with their additional resultant 'c'. This would verify that the first GPU code which you wrote has worked correctly. Still confused? Well, have a look at Figure 2, which will give you a clear understanding of how things work in parallel, in this case.

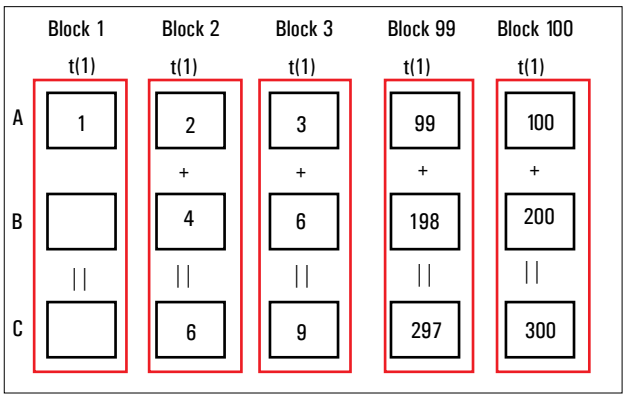

Figure 2: Parallel vector addition

Performance

You may wonder how the GPU code can perform about 100 times faster than the CPU code, since we have created 100 blocks that are executing in parallel. This is not the case, since there is an overhead involved in copying data between the CPU and the GPU and the resultant data back to the CPU. Hence, CUDA is generally used for computing algorithms that are significantly data intensive, as it would then make sense to spend some time for data transfer. GPUs are, therefore, generally known as data intensive computational devices.

As a next step, you could try programming matrix addition on the GPU in parallel to get a good grip on kernels, threads and parallel execution. Your best companion for this would be the links and the books mentioned in the 'References' section at the end of this article. I also recommend you visit the NVIDIA website and documentation, as it will give you a good idea about the power of CUDA if you are not already impressed by what this simple GPU device on your laptop can do. Next up in this series, I might cover an advanced CUDA program with multiple threads and blocks on a grid, and analyse the running time of the code. I will follow it up with a discussion on OpenACC and other simpler parallel programming models that have come up recently.

Till then, start thinking of algorithms in parallel. The world of parallel computing is here to stay! END

References

[1] 'CUDA C Programming Guide' by NVIDIA; *http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html*
[2] 'CUDA Application Design and Development' by Rob Farber
[3] 'Programming Massively Parallel Processors' by David B. Kirk and Wen-mei W. Hwu

By: Tejaswi Agarwal

A FOSS enthusiast, the author is passionate about compute power utilisation, run time and memory utilisation of algorithms. Some of his research areas are computer architecture, parallel programming and performance engineering. He can be contacted at *tejaswi.agarwal2010@vit.ac.in*