HPC - Exercise 6

Eduardo Gonnelli

10 July 2019

1 Introduction

The aim of this report is to employ the *CUDA* programming language to solve three problems on the *GPU* hardware. Firstly, to introduce the *CUDA*, was developed a code that reversed the order of an array. Secondly, it was implemented a matrix transpose code using threads and blocks. Finally, the matrix multiplication was created for matrix sizes $2048\hat{2}$ and for threads x block of max 512.

The GPU Tesla model K20m was used for running the codes. The specifications of the GPU can be seen on Figure 1. To access these information, the command *nvidia-smi* was executed. The NVIDIA System Management Interface (nvidia-smi) is a command line utility, based on top of the NVIDIA Management Library (NVML), intended to aid in the management and monitoring of NVIDIA GPU devices. This utility allows administrators to query GPU device state and with the appropriate privileges, permits administrators to modify GPU device state.

```
[egonnell@gn05-01 ex06]$ nvidia-smi
Wed Jul 10 18:05:21 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.48                 Driver Version: 410.48                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K20m          Off  | 00000000:20:00.0 Off |                    0 |
| N/A   26C    P0    46W / 225W |      0MiB /  4743MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K20m          Off  | 00000000:8B:00.0 Off |                    0 |
| N/A   27C    P0    54W / 225W |      0MiB /  4743MiB |     98%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Figure 1: NVIDIA System Management Interface (*nvidia-smi*).

To load the GPU-accelerated libraries, debugging and optimization tools, a C/C++ compiler and a runtime library, command line *module load cudatoolkit10.0* was used. The NVIDIA CUDA Toolkit provides a development environment for creating high performance GPU-accelerated applications. The compiler command of the *CUDA* is called: *NVCC*, that means, Nvidia CUDA Compiler. To compile a file with extension *.cu*, the command line *nvcc* < *File.cu* > was applied.

2 Results

The results for the Reverse Array, Matrix Transpose and Matrix Multiplication can be seen in this section. The codes developed to solve those problems are in the Appendix A.

The Figure 2 shows the output for the executable *reverse_array.o*. It can be seen that the array of size N = 10 have had its order reversed by the *cuda* code.

```
[egonnell@gn05-02 ex06]$ ./reverse_array.o
This program prints the reverse array of size N = 10
Array on the normal order:
 0 1 2 3 4 5 6 7 8 9
Array on the reverse order:
 9 8 7 6 5 4 3 2 1 0
```

Figure 2: Output of the *reverse_array.o* of size N = 10. It can be seen the sorted order array and it after been reversed by the GPU.

The Figure 3 shows the transpose matrix by the GPU. The output of the executable *transpose_matrix.o* was printed on the terminal screen.

```
[egonnell@gn05-01 ex06]$ ./matrix_transpose.o
This Program Transposes a Matrix of size N = 10

 0  0  0  0  0  0  0  0  0  0
 1  1  1  1  1  1  1  1  1  1
 2  2  2  2  2  2  2  2  2  2
 3  3  3  3  3  3  3  3  3  3
 4  4  4  4  4  4  4  4  4  4
 5  5  5  5  5  5  5  5  5  5
 6  6  6  6  6  6  6  6  6  6
 7  7  7  7  7  7  7  7  7  7
 8  8  8  8  8  8  8  8  8  8
 9  9  9  9  9  9  9  9  9  9

The Transpose matrix is:

 0  1  2  3  4  5  6  7  8  9
 0  1  2  3  4  5  6  7  8  9
 0  1  2  3  4  5  6  7  8  9
 0  1  2  3  4  5  6  7  8  9
 0  1  2  3  4  5  6  7  8  9
 0  1  2  3  4  5  6  7  8  9
 0  1  2  3  4  5  6  7  8  9
 0  1  2  3  4  5  6  7  8  9
 0  1  2  3  4  5  6  7  8  9
 0  1  2  3  4  5  6  7  8  9
```

Figure 3: Output of the *transpose_matrix.o* of size N = 10. It can be seen the transposed matrix was obtained successfully by the GPU.

The matrix multiplication of sizes N = 5 and N = 10 were executed on GPU. The Figure 4 shows the outputs for these to sizes. In the case for the matrix size of 2048 it was not possible to print the result in the terminal screen. However, it was not a problem, because the operation was successfully executed. The Figure 5 shows the result for the matrix multiplication of sizes N = 100 and N = 2048.

```
[egonnell@gn05-02 ex06]$ ./multi_matrix.o 5

Result of matrix multiplication on GPU:
        0       0       0       0       0
        10      10      10      10      10
        20      20      20      20      20
        30      30      30      30      30
        40      40      40      40      40

The Matrix multiplication of size n = 5 was completed

[egonnell@gn05-02 ex06]$ ./multi_matrix.o 10

Result of matrix multiplication on GPU:
        0       0       0       0       0       0       0       0       0       0
        45      45      45      45      45      45      45      45      45      45
        90      90      90      90      90      90      90      90      90      90
        135     135     135     135     135     135     135     135     135     135
        180     180     180     180     180     180     180     180     180     180
        225     225     225     225     225     225     225     225     225     225
        270     270     270     270     270     270     270     270     270     270
        315     315     315     315     315     315     315     315     315     315
        360     360     360     360     360     360     360     360     360     360
        405     405     405     405     405     405     405     405     405     405

The Matrix multiplication of size n = 10 was completed
```

Figure 4: Output of the *mult_matrix.o* of sizes N = 5 and N =10. Both cases were executed on GPU.

```
[egonnell@gn05-02 ex06]$ ./multi_matrix.o 100

Result of matrix multiplication on GPU:

The Matrix multiplication of size n = 100 was completed

[egonnell@gn05-02 ex06]$ ./multi_matrix.o 2048

Result of matrix multiplication on GPU:

The Matrix multiplication of size n = 2048 was completed
```

Figure 5: Output of the *mult_matrix.o* of sizes N = 100 and N = 2048. Both cases were executed on GPU.

References

[1] Jason Sanders and Edward Kandrot. *CUDA by Example - An IntroductIon to General-Purpose GPU Programming*. Addison-Wesley, 2011.

[2] Gregory Ruetsch and Massimiliano Fatica. *CUDA Fortran for Scientists and Engineers - Best Practices for Efficient CUDA Fortran Programming*. ELSEVIER, 2014.

[3] Cyril Zeller, NVIDIA Corporation - *CUDA C/C++ Basics - Supercomputing 2011 Tutorial*
https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf

[4] Andreas W Götz, San Diego Supercomputer Center, University of California, San Diego - *GPU Computing and Programming*
https://www.sdsc.edu/Events/training/webinars/gpu_computing_and_programming_2019/sdsc-gpu-webinar-goetz-2019-04-09.pdf

Appendices

A Codes

Listing 1: C++ code - Reverse Array

```
#include<iostream>

      __global__ void reverse( int* d_out, int* d_in, const int N )
      {
       int tid = ( blockIdx.x * blockDim.x ) + threadIdx.x;
       if ( tid < N ){d_out[ tid ] = N - d_in[ tid ] - 1;}
      };

int main( int argc, char** argv)
{
      std::cout << "This␣program␣prints␣the␣reverse␣array␣of␣size␣N␣=␣10" << "\n";

      const int N = 10;
      int* h_in  = new int [ N ];
      int* h_out = new int [ N ];

      //Host and Device memory
      const int bytes = N * sizeof(int);

      // Pointer for Device memory + Allocation
      int *d_in, *d_out;
      cudaMalloc( (void **) &d_in, bytes );
      cudaMalloc( (void **) &d_out, bytes );

      // Threadblock size
      const int NUM_THREADS = 256;
      // Grid size
      const int NUM_BLOCKS = ( N + NUM_THREADS - 1 ) / NUM_THREADS;

      //Host initialization
      for ( int i = 0; i < N ; ++i )
          {
            h_in[i] = i;
          }

      //Print the array - normal order
      std::cout << "\n";
      std::cout << "Array␣on␣the␣normal␣order:␣" << "\n";
      for ( int i = 0; i < N ; ++i )
      {
      std::cout << "␣" << h_in[i];
      }

      std::cout << "\n";
      std::cout << "\n";

      //Host --> Device
      //h_in to d_in
      cudaMemcpy( d_in, h_in, bytes, cudaMemcpyHostToDevice );

      // Reversing the array on GPU
      //CUDA kernel - Reverse Array
      reverse<<< NUM_BLOCKS, NUM_THREADS >>>( d_out, d_in, N );
      //Copy: Device to Host ( d_out -> h_out )
      cudaMemcpy( h_out, d_out, bytes, cudaMemcpyDeviceToHost );

          std::cout << "\n";
          std::cout << "Array␣on␣the␣reverse␣order:␣" << "\n";
          for ( int i = 0; i < N ; ++i )
          {
            std::cout << "␣" << h_out[i];
          }

      //Free Device and Host memory
      cudaFree( d_in );
      cudaFree( d_out );
      delete[] h_in;
      delete[] h_out;


   return 0;
}
```

Listing 2: C++ code - Transpose Matrix

```
#include<iostream>

      __global__ void matrixTranspose( int* d_in, int* d_out, const int n )
      {
      //Row index
      int row = ( ( blockIdx.y * blockDim.y ) + threadIdx.y );
      //Column index
      int col = ( ( blockIdx.x * blockDim.x ) + threadIdx.x );
      //Boundary protection
      if ( ( row < n ) && ( col < n ) )
        {
          d_out[ col * n + row ] = d_in[ row * n + col ];
        }
      };

int main(int argc, char** argv)
{
   std::cout << "\n";
   std::cout << "This␣Program␣Transposes␣a␣Matrix␣of␣size␣N␣=␣10" << "\n";

   //Size of matrix n x n | n = 10
   const int N = 10;

   size_t bytes = N * N * sizeof( int );

   int* h_in  = new int [ N*N ];
   int* h_out = new int [ N*N ];
   int *d_in, *d_out;

   cudaMalloc( &d_in, bytes );
   cudaMalloc( &d_out, bytes );

// Initialize matrix | each line has the same value
    for (int row = 0; row < N; row++)
    {
      for (int col = 0; col < N; col++)
      {
        h_in[row * N + col] = row ;
      }
    }

    std::cout << "\n";
    for (int row = 0; row < N; row++)
    {
      for (int col = 0; col < N; col++)
      {
        std::cout << "␣" << h_in[row * N + col] << "␣";
      }
        std::cout << "\n";
    }

    // Host --> Device
    cudaMemcpy( d_in, h_in, bytes, cudaMemcpyHostToDevice );
    const int BLOCK_SIZE = 16;
    const int GRID_SIZE = ( N + BLOCK_SIZE - 1) / BLOCK_SIZE;

    dim3 blocks( GRID_SIZE, GRID_SIZE, 1 );
    dim3 threads( BLOCK_SIZE, BLOCK_SIZE, 1 );

    // Transposing matrix on GPU
    matrixTranspose <<< blocks, threads >>> ( d_in, d_out, N );

    // Device --> Host
    cudaMemcpy( h_out, d_out, bytes, cudaMemcpyDeviceToHost );

    // Print the Transpose
    std::cout << "\n";
    std::cout << "The␣Transpose␣matrix␣is:␣" << "\n";
    std::cout << "\n";

    for (int row = 0; row < N; row++)
    {
      for (int col = 0; col < N; col++)
      {
        std::cout << "␣" << h_out[row * N + col] << "␣";
      }
    std::cout << "\n";
    }

    // Free memory
    delete[] h_in;
    h_in = NULL;
    delete[] h_out;
    h_out = NULL;
    cudaFree(d_in);
    cudaFree(d_out);

  return 0;
 }
```

Listing 3: C++ code - Matrix Multiplication

```
#include <iostream>

      __global__ void matrix_multiply( const int* d_a, const int* d_b, int* d_c, const int n )
      {
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;
        int helper_multiplication = 0;
        if ( ( row < n ) && ( col < n ) )
          {
            for ( int k = 0; k < n; k++ )
                {
                  helper_multiplication += d_a[ row * n + k ] * d_b[ k * n + col ];
                }
            d_c[ row * n + col ] = helper_multiplication;
          }
      };

int main( int argc, char** argv )
 {

      const int n = atoi( argv[1] );

      size_t bytes = n*n*sizeof( int );

      int* h_a = new int [ n*n ];
      int* h_b = new int [ n*n ];
      int* h_c = new int [ n*n ];;

      int *d_a, *d_b, *d_c;

      cudaMalloc( &d_a, bytes );
      cudaMalloc( &d_b, bytes );
      cudaMalloc( &d_c, bytes );

      //Matrix initialization

      for ( int row = 0; row < n; row++ )
      {
        for ( int col = 0; col < n; col++ )
            {
                h_a[ row * n + col ] = row;
            }
      }

      for ( int row = 0; row < n; row++ )
      {
        for ( int col = 0; col < n; col++ )
            {
                h_b[ row * n + col ] = row;
            }
      }
      //Host ---> Device ( h_a -> d_a )
      cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice );
      cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice );

      const int BLOCK_SIZE = 32;
      const int GRID_SIZE = ( n + BLOCK_SIZE - 1 ) / BLOCK_SIZE;
      dim3 grid( GRID_SIZE, GRID_SIZE, 1 );
      dim3 threads( BLOCK_SIZE, BLOCK_SIZE, 1 );

      // Matrix Multiplication - CUDA KERNEL
      matrix_multiply <<< grid, threads >>> ( d_a, d_b, d_c, n );

      //Device --> Host (d_c -> h_c)
      cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

      std::cout << "\nResult␣of␣matrix␣multiplication␣on␣GPU:␣\n";

      //Print only if n < 20
       if( n < 20 && h_c != NULL ){

             for ( int row = 0; row < n; row++ )
                   {
          for ( int col = 0; col < n; col++ )
                       {
                         std::cout << "\t" << h_c[ row * n + col ] ;
            }
                    std::cout << "\n";
             }
                                 }

       std::cout << "\n";
       //Validation of the GPU computation
      std::cout << "The␣Matrix␣multiplication␣of␣size␣n␣=␣" << n << "␣was␣completed␣\n\n";

    // Free Host memory
    delete[] h_a;
     h_a = NULL;
   delete[] h_b;
```

```
    h_b = NULL;
  delete[] h_c;
    h_c = NULL;

    // Free Device memory
  cudaFree( d_a );
  cudaFree( d_b );
  cudaFree( d_c );

 return 0;
}
```