**Applied Parallel Programming With GPU** **Lab 1 Exercises**

**Semih Kekül**

**1347657**

**Vector - Vector Inner Product**

1. **Problem definition:**

Implementation of production of two vectors with CUDA. Parallelization would fasten up the element by element multiplication, however, it would not effect the summation speed.

1. **Algorithm:**

Product elements of vectors wrt to their indices and write it to shared memory. The thread indexed 0 sums all of the shared memory. I have used shared memory because it is faster then global memory. Also, thread 0 waits a \_\_syncthreads(); because maybe the other threads had not been finished producting the corresponding elements.

**Grid and Block Size**

The block size is 512 x 1 because 512 is the maximum size of a block. The grid size is by 1 X 1 because I am using one block for dot product of two vectors which are 512 element each.

1. **Benchmarking:**

**Memory**

Memory need is the global memory of the vectors plus the the shared memory.

**Time**

|  |  |  |
| --- | --- | --- |
| Matrix size | Time Without I/O | Time With I/O |
| 20 x 20 | ~0.01 ms | ~ 6 ms(higher as expected) |
| 200 x 200 | ~0.013 ms | ~ 6 ms(higher as expected) |
| 512 x 512 | ~0.02 ms | ~ 6 ms(higher as expected) |

1. **Pros and** **Cons**

The algorithm is not scalable it is fixed to at most 512 sized vectors. Moreover, the algorithm does not utilize the GPU. It is using just one block. Because of using one block the speed do not change according to matrix size.

1. **Discussion**

It is possible to create more scalable algorithm such as dividing bigger matrice to more blocks.

1. **Environment**:

|  |  |
| --- | --- |
| GPU processor | GeForce GTX 760M |
| CUDA Cores: | 768 |
| Core clock: | 627 MHz |
| Memory data rate: | 4008 MHz |
| Total available graphics memory: | 4096 MB |
| Memory interface: | 128-bit |
| Memory bandwidth: | 64.13 GB/s |
| Processor: | Intel Core i7 2.20 GHz |
| Memory | 32 GB |
| OS | Windows 7 64-bit |

**Matrix - Matrix Summation**

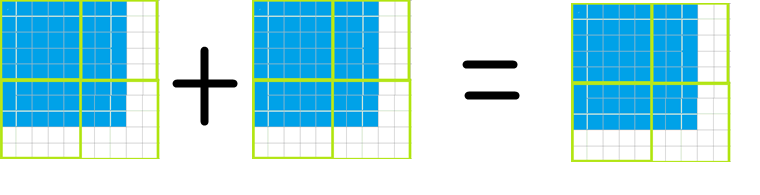
1. **Problem definition:**

Implementation of summation of two matrices with CUDA. Parallelization would fasten up the element by element summation, which is the entire job.

1. **Algorithm:**

The matrix data is 1D array however the blocks are located 2D in my algorithm.

elements. Each thread’s idx is calculated by a 2D indexing mechanism and each thread adds correspoding two numbers coming from each matrix.

 Figure 1; The blue ones are the cells of the output matrix and the whole structure is the grid. The empty kernels are padding at the end of blocks which are present because the matrix is not small enough to fit in a block and not big enough to fit in 4 blocks.

**Grid and Block Size**

Grid is 2D and block is also 2D. The size of the grid is determined by block size and matrix sizes.

1. **Benchmarking:**

**Memory**

Memory usage is the global memory for the two matrices.

**Time**

|  |  |  |  |
| --- | --- | --- | --- |
| Matrix size | Block size | Time Without I/O | Time With I/O |
| {20 x 20} x {20 x 20} | 16x16 | ~0.01 ms | ~ 1.4 ms(higher as expected) |
| {200 x 200} x {200 x 200} | 32 x 32 | ~0.02 ms | ~ 1.7 ms(higher as expected) |
| {2000 x 2000} x {2000 x 2000} | 32 x 32 | ~1.3 ms | ~ 15 ms(higher as expected) |

1. **Pros and** **Cons**

The algorithm is scalable and fast, however there are unused threads if matrix size is less then the k \* block size.

1. **Environment**:

|  |  |
| --- | --- |
| GPU processor | GeForce GTX 760M |
| CUDA Cores: | 768 |
| Core clock: | 627 MHz |
| Memory data rate: | 4008 MHz |
| Total available graphics memory: | 4096 MB |
| Memory interface: | 128-bit |
| Memory bandwidth: | 64.13 GB/s |
| Processor: | Intel Core i7 2.20 GHz |
| Memory | 32 GB |
| OS | Windows 7 64-bit |

**Matrix - Vector Product**

1. **Problem definition:**

Implementation of summation of a matrix and a vector with CUDA. Parallelization would fasten up the element by element production, however, it would not effect the summation speed.

1. **Algorithm:**

The matrix data and vector data is 1D array and also the blocks are located 1D in my algorithm.

Each thread’s idx is calculated by a 1D indexing mechanism wrt to multiple blocks and each thread products correspoding two numbers one is coming from matrix and the other is coming from the vector and write to the shared memory.

After \_\_syncthreads() the 0th thread of each block sums the corresponding column size 1D memory using the shared memory. And the sum is written to output matrix.

**Grid and Block Size**

Grid is 1D and block is also 1D. The dimension of the grid is determined by block dimension and matrix dimension (matrix row size / block size)

1. **Benchmarking:**

**Memory**

Memory usage is the global memory for the an input matrix and an input vector and an output vector and a shared memory vector;

**Time**

|  |  |  |  |
| --- | --- | --- | --- |
| Matrix size | Block size | Time Without I/O | Time With I/O |
| {128 x 128} x {128} | 128 | ~0.17 ms | ~ 1.0 ms(higher as expected) |
| {256 x 256 } x {256 } | 256 | ~1.1 ms | ~ 1.4 ms(higher as expected) |
| {512 x 512 } x {512 } | 512 | ~7.2 ms | ~ 11 ms(higher as expected) |

1. **Pros and** **Cons**

The algorithm is scalable and fast, however the summation may be done with reduction.

1. **Environment**:

|  |  |
| --- | --- |
| GPU processor | GeForce GTX 760M |
| CUDA Cores: | 768 |
| Core clock: | 627 MHz |
| Memory data rate: | 4008 MHz |
| Total available graphics memory: | 4096 MB |
| Memory interface: | 128-bit |
| Memory bandwidth: | 64.13 GB/s |
| Processor: | Intel Core i7 2.20 GHz |
| Memory | 32 GB |
| OS | Windows 10 64-bit |