Bilkent University

Department of Computer Engineer

CS426 Parallel Computing

Project 4 Report

Serhat Aras

21401636

Problem Definition

In this final project, I am required to implement a CUDA implementation of sparse matrix vector multiplication. In order to do this, I used per-initialize Vector and various Compressed Sparse Row matrix. The operation is basically to do a normal matrix multiplication on very large sparse matrix with a vector and store the result of the matrix-vector multiplication on to the vector itself at the end of every iteration.

File I/O

First thing to handle is to read the matrix dimension's and data in to a C compatible format. In order to do this, I created a SparseMatrix structure to store the required implementation.

In order to represent the sparse matrix, I created the following data structure.

```
struct SparseMatrix{
int *row_ptr;
int *col_ind;
float *values;
};
```

Computation

The first line of the input file is the dimensions and the number of elements that are non zero in the matrix. This data is used while checking and processing the boundaries of the matrices and the creation of the Vector X. After reading the values from the file and writing to the corresponding arrays in the SparseMatrix instance. This instance will later passed to Host and Kernel parts of the CUDA. Note that all of the row and column values are reduced by one in order to match the C Language array indexing. After finishing the Data read for the file passed from last the console parameter the Vector X is initialize, allocated and set it to all 1. Then we call our host function in order to start CUDA process of the project.

The host function has the following function signature.

```
__host__ void outer_VecMatMult_Started (struct SparseMatrix* sparse_Matrix, float *x, int row_size, int
col_size, int value_size, int NUMBER_OF_CUDA_THREADS ,int NUMBER_OF_REPETITION , int OUTPUT_FLAG);
```

The main function has passed all of the relevant data that the function needs. In the outer_VecMAtMult_Started fucntion, we handle the cudaMalloc, cudaMemcpy kernel(matrixVectorMultCuda) call, cudaFree and print operation of the Sparse Matrix Vector Multiplication. After initializing the device copies of the row_ptr, col_ind, value, and dev_x arrays, arrays except dev_x array copied to the device using cudaMemcpy function.

In order to implement the multiple iteration, I used the following fol loop to maintain the x Memcopy from both device and host, and also kernel call to actually start multiplication of the Matrix and Vector.

Iteration For Loop:

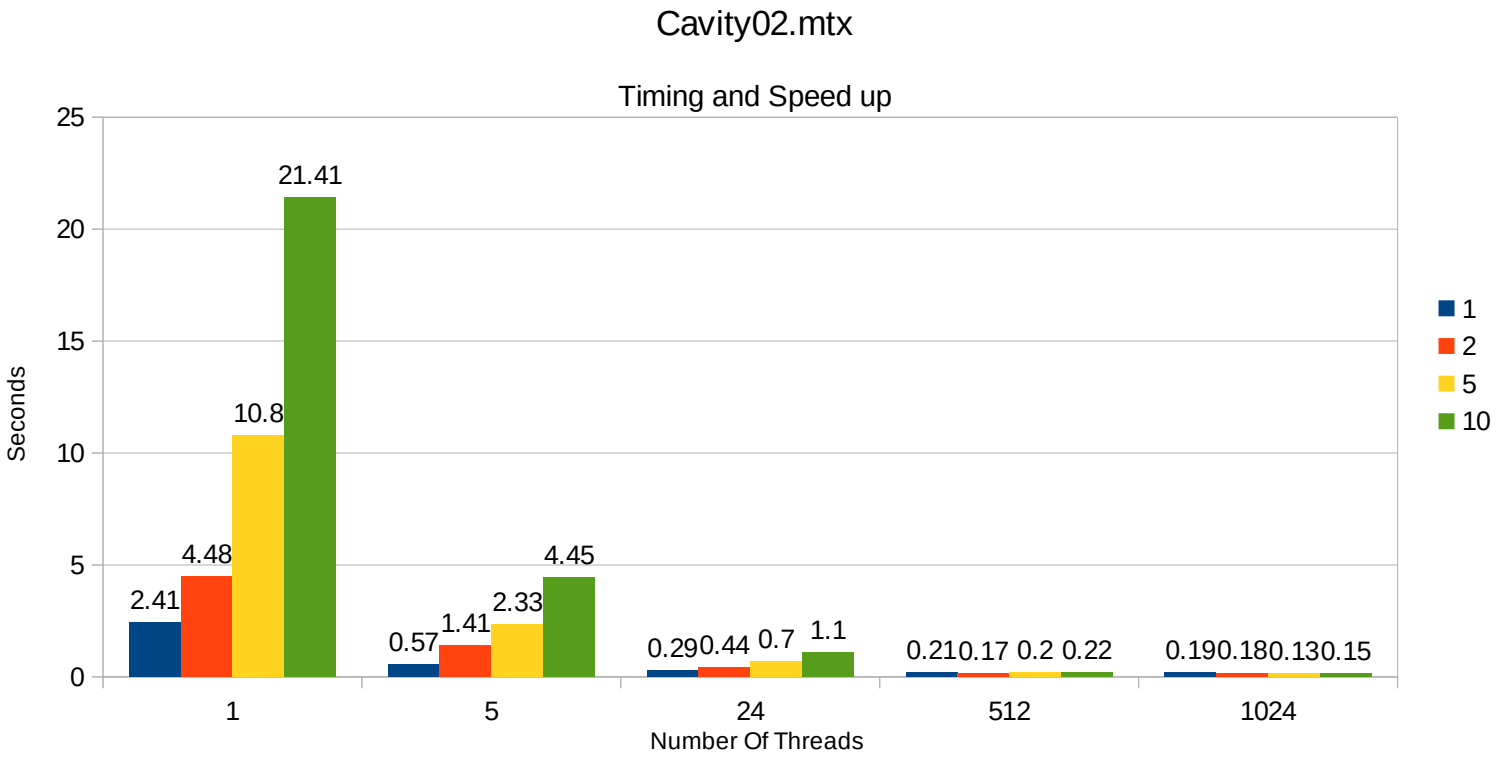
```
for(int i=0; i<NUMBER_OF_REPETITION; i++){
cudaMemcpy(dev_x, x, size_f_x,cudaMemcpyHostToDevice);
matrixVectorMultCuda<<<1,NUMBER_OF_CUDA_THREADS>>>(dev_row_ptr,dev_col_ind,dev_values,dev_x,row_size,co
l_size,value_size,dev_threadsID_ptr_Mapper,dev_numberOfIndexesToProcess);
cudaMemcpy(x, dev_x, size_f_x,cudaMemcpyDeviceToHost);
}
```

After finishing the iterations, the final x will be printed and other valuable information is may printed based on the command line input for output format.

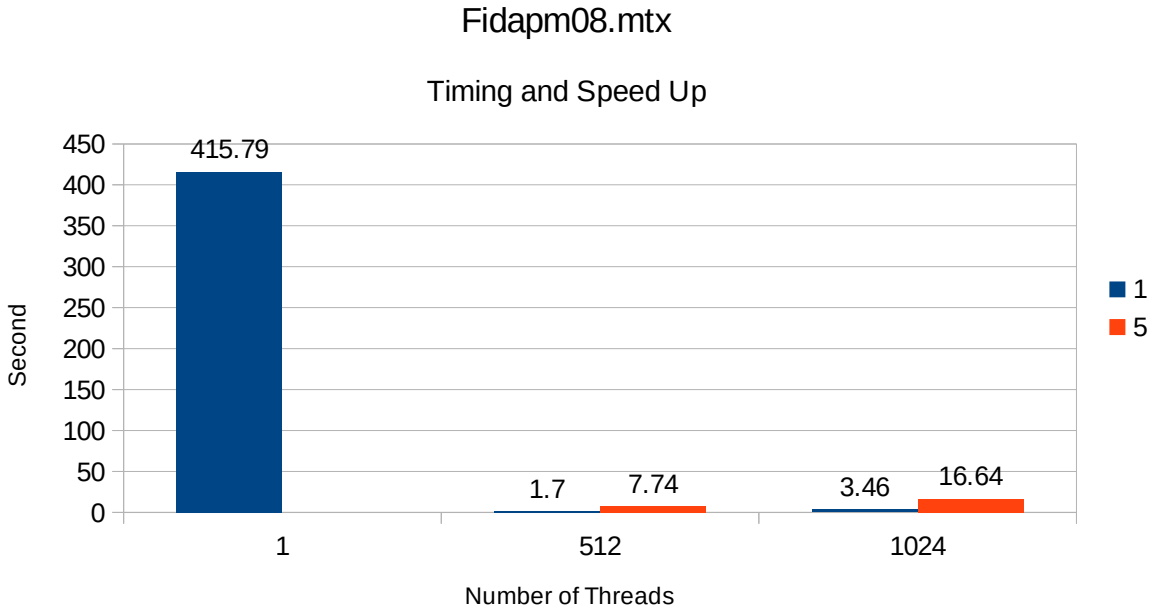
Parallelization Strategy:

The parallelization strategy used in the implementation is basically dividing the Sparse Matrix to row for each Cuda Thread. The division is basically depends on the number of threads and number of rows in the Sparse matrix. Only row value is enough since the matrix is expected to be in square matrix form. In the division part, the program generates 2 primary look up arrays in order to keep track of which thread must compute which row on Vector Matrix Multiplication. The arrays passed to the devices since each of the copy of the kernel on the device equipped with unique ThreadID which can be computed internally. For one thread, after knowing which row to make computation, the thread basically search rows from the input matrix and do the vector multiplication. Since we don't have any race condition on vector or SparseMatrix read/write, each of the thread can actually read from the SparseMatrix and vector x, write the computed value back to x, for example, for thread responsible from the multiplication of the row zero, travels through the pointed rows and find non zero columns(we actually have only the non zero information on all matrix) and computes an internal sum, after thread zero finishes its values, it waits to get sync with other threads. After synchronization is completed, the sum is placed on the X Vector to its corresponding index. After all the multiplication is completed, the X vector is posted back to the host, and if an iteration is waiting to get executed, the x vector is feed back to the devices.

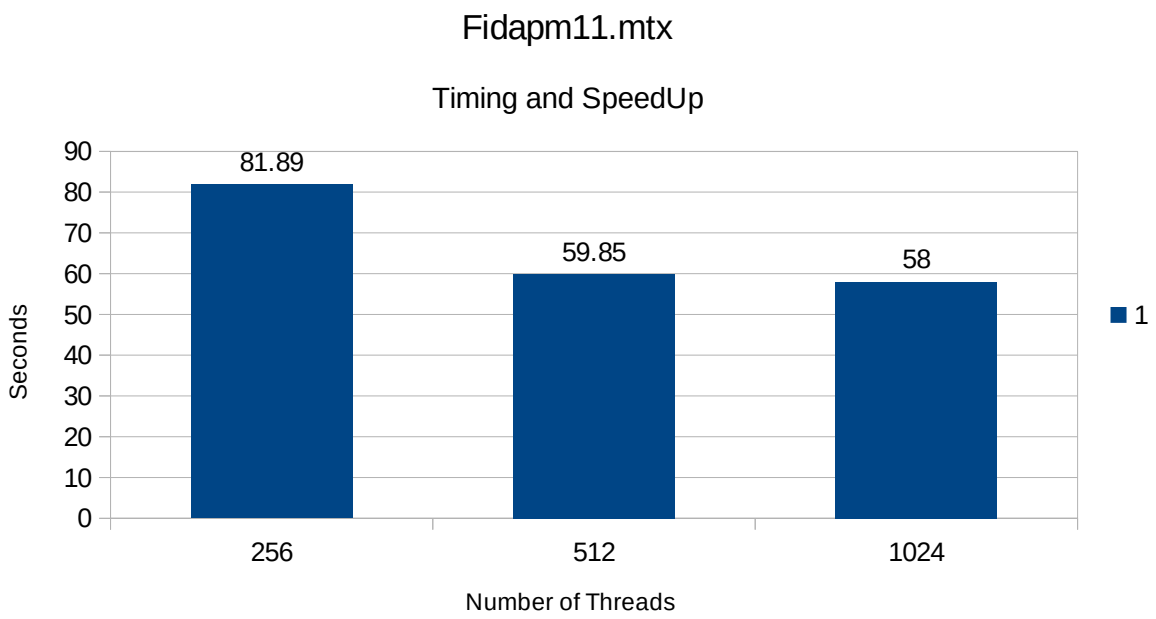
Results and Discussions:



In the cavity example, I did the most elaborate examination. The program runned with 1,5,24,512,1024 threads. All of these threads also runned with 1,2,5 and 10 iterations and their timing is recorded. In the above graph, we see that if the thread is 1, it took maximum amount of time among others, and naturally if we increase the number of iterations, the execution time is also increased. The Number of threads affects the execution time directly. If the threads increased, more workers introduced to the problem, and the sub problem size is getting smaller. After passing some threshold, the only thing affecting the execution time is memory copy between device and host. Therefore, increasing the thread size is not always beneficial for the problem.



On the Fidapm08 example, the matrix is much more bigger than the cavity02 example. Thus, the data is more bigger and there is lot to process than before. This growth on the data size can be examined though execution time of the program. However, the number of cores is expected to reduce the execution time as its gets bigger, in this case, it turns out didn't. These kind of performance losts can be happen due to the position of the non-zero values in the array and the number of operations per second is increased when matrix is more bigger or dense.



The same effect can be examined in the much more bigger Fidapm11 example also. Since it took too much time on 1 thread and 1 iteration compared to fidapm08 example (450+ seconds), I didn't wait the result because getting the 1-1 execution configurations result will not prove anything new that is already have been proved. Therefore, I kept it simple this time to show the affect of the matrix size. The algorithm is exactly the same and the amount of time that 256-512-1024 threads took is given in the above graph. As we can see, the number of threads helps us to some point but after that point we have to wait the bottleneck of the cuda applications which is the memory copy of between cpu and gpu.

In this project, we can experience the affect of the parallel algorithm that ran on the GPU using CUDA. The matrix size is heavily affects the performance of the program. However, this heavy toll can be reduced using cuda cores. The application parellized the matrix vector multiplication operation in a way that, each thread is responsible from the smaller portion of the big sparse matrix. Thus, after examining the results and evidence provided and discussed above, the benefits of the cuda is understandably great. But to not to waste the resourses on the computer, mainly the GPU and Memory, the nunber of threads must choosen carefully in order to have peak efficiency.

Example Execution Commands and Outputs :

- 64 threads/5 iterations/ a test matrix with smaller size/using 1st output option

```
(base) captainpicard@NCC1701:~/Desktop/CS426/P4$ ./matrixMul 64 5 1 test.mtx
Initial Matrix:
    Values Array: [ 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00]
    Col_Ind Array: [ 0, 0, 2, 2, 3, 3, 5, 6, 8, 7, 12]
    Row_Ptr Array: [ 0, 7, 4, 7, 0, 3, 5, 1, 4, 7, 12]
Vector: [ 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00, 1.000000e+00]
Resulting Vector: [ 3.200000e+01, 1.000000e+00, 0.000000e+00, 1.000000e+00, 3.200000e+01, 1.000000e+00, 0.000000e+00, 2.430000e+02, 0.000000e+00, 0.000000e+00, 0.000000e+00, 0.000000e+00]
----
Time taken: 0 seconds 56 milliseconds
```

- 64 threads/5 iterations/ a test matrix with smaller size/using 2nd output option

```
(base) captainpicard@NCC1701:~/Desktop/CS426/P4$ ./matrixMul 64 5 2 test.mtx
Resulting Vector: [ 3.200000e+01, 1.000000e+00, 0.000000e+00, 1.000000e+00, 3.200000e+01, 1.000000e+00, 0.000000e+00, 2.430000e+02, 0.000000e+00, 0.000000e+00, 0.000000e+00, 0.000000e+00]
----
Time taken: 0 seconds 53 milliseconds
```

The System Information:

Hardware Info of the Computer:

Processor	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Memory	16381MB
Graphic Card	NVIDIA Gtx1070 (8 GB)
Operating System	Ubuntu 19.04
Kernel Version	5.0.0-13-generic
GCC Comp.	Using built-in specs.

Cuda Information:

```
(base) captainpicard@NCC1701:~$ nvidia-smi
Sun Jun  2 20:18:59 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1070    On   | 00000000:01:00.0  On |                  N/A |
| N/A   51C    P0    38W /  N/A |   1406MiB /  8111MiB |      9%      Default |
+-------------------------------+----------------------+----------------------+
```