Assignment 4
Report

Submitted by :
Punit Kumar Jha
NETID: punit2
Graduate Student
Department of Chemistry

December 17, 2017

Contents

List of Figures

List of Tables

1 Introduction:

1.1 Objective

This assignment required use to a multi-core CPU using OpenMP and a GPU using OpenACC to implement a matrix-multiply routine for multiplying a band matrix by a dense matrix. The performance impact of varying the band and matrix size was studied . For this assignment, the Blue Water's supercomputer system was used.

1.2 Banded Matrix

In mathematics, a band matrix is a sparse matrix whose non-zero entries are confined to a diagonal band, comprising the main diagonal and zero or more diagonals on either side.

$$\begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & \ddots & & & & \vdots \\ 0 & a_{32} & a_{33} & a_{34} & \ddots & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & a_{76} & a_{77} & a_{78} & 0 \\ \vdots & & & & \ddots & a_{87} & a_{88} & a_{89} \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & a_{98} & a_{99} \end{bmatrix} \tag{1}$$

Bandwidth: Consider a $n \times n$ matrix $A = (a_{i,j})$. If all the elements are zero outside a diagonally bordered band whose range is determined by constants k_1 and k_2 :

$$(a_{i,j}) = 0 \text{ if } j < i - k_1 \text{ or } j > i + k_2 \text{ ; where } k_1 \text{ and } k_2 > 0$$

then the quantities k_1 and k_2 are called the lower bandwidth and upper bandwidth, respectively. The bandwidth of the matrix is the maximum of k_1 and k_2. Mathematically, it is the number k such that:

$$a_{i,j} = 0 \text{ if } |i - j| > k$$

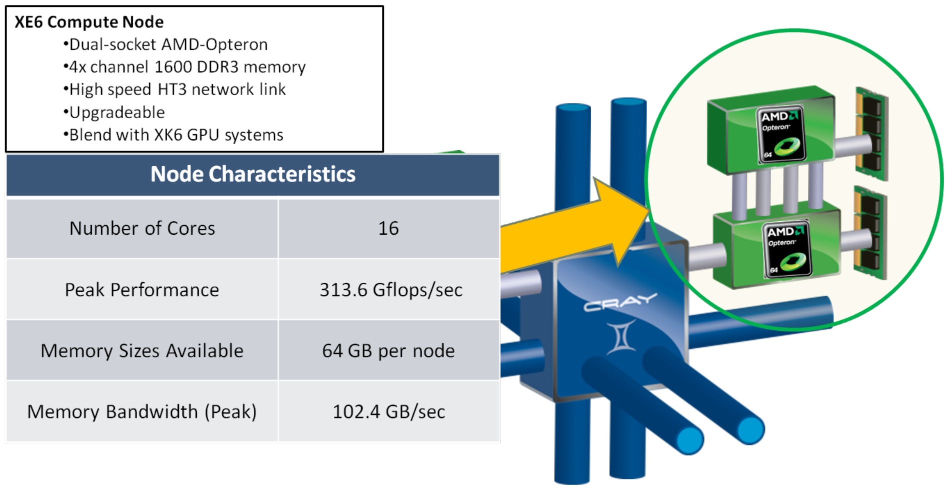
1.3 Brief Intro to Blue Waters Super Computing System

Blue Waters is a Cray XE6/XK7 system consisting of more than 22,500 XE6 compute nodes. Each XE6 node contains two AMD Interlagos processors. I also has more than 4200 XK7 compute nodes. Each XK7 has one AMD Interlagos processor and one NVIDIA GK110 *Kepler* accelerator in a single Gemini interconnection fabric.[1]

1.3.1 Blue Waters Nodes

The Blue Waters has three different types of nodes which are listed below:

Traditional Compute Nodes (XE6)
The XE6 dual-socket nodes have 2 AMD Interlagos model 6276 CPU processors (one per socket) with a clock speed of at least 2.3 GHz and 64 GB of physical memory. The Interlagos architecture employs the AMD Bulldozer core design in which two integer cores share a single floating point unit. The Bulldozer core has 16KB/64KB data/instruction L1 caches and 2 MB shared L2 . Each core is able to complete up to 8 floating point operations per cycle. The architecture supports 8 cores per socket with two die, each die containing 4 cores forming a NUMA domain. The 4 cores of a NUMA (Non-Uniform Memory Access) domain share an 8 MB L3 cache.



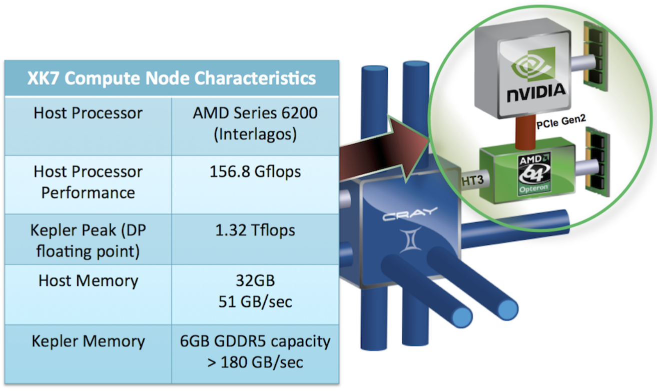
XE6 Compute Node
- Dual-socket AMD-Opteron
- 4x channel 1600 DDR3 memory
- High speed HT3 network link
- Upgradeable
- Blend with XK6 GPU systems

Node Characteristics	
Number of Cores	16
Peak Performance	313.6 Gflops/sec
Memory Sizes Available	64 GB per node
Memory Bandwidth (Peak)	102.4 GB/sec

Figure 1: Traditional Compute Nodes (XE6)

GPU-enabled Compute Nodes (XK7)

The accelerator nodes are equipped with one Interlagos model 6276 CPU processor and one NVIDIA GK110 "Kepler" accelerator K20X. The CPU acts as a host processor to the accelerator. In the current design, the NVIDIA accelerator does not directly interact with the Gemini interconnect so the date has to be moved to a node containing an accelerator. Each XK7 node has 32 GB of system memory while the accelarator has 6 GB of memory. The Kepler GK110 implementation includes 14 Streaming Multiprocessor (SMX) units and six 64bit memory controllers. Each of the SMX units feature 192 singleprecision CUDA cores.



XK7 Compute Node Characteristics	
Host Processor	AMD Series 6200 (Interlagos)
Host Processor Performance	156.8 Gflops
Kepler Peak (DP floating point)	1.32 Tflops
Host Memory	32GB 51 GB/sec
Kepler Memory	6GB GDDR5 capacity > 180 GB/sec

Figure 2: GPU-enabled Compute Nodes (XK7)

I/O and Service Nodes (XIO)

Each Cray XE6/XK6 XIO blade has four service nodes. These service nodes use AMD Opteron "Istanbul" six-core processors, with 16 GB of DDR2 memory, and the same Gemini interconnect processors as compute nodes. These XIO nodes perform on of the following functions:

- **esLogin Nodes**: provide user access and log in services for the system.
- **PBS MOM Nodes**: these are nodes which run the jobs (where the PBS script gets executed and aprun is launched) and where interactive batch jobs (qsub -I) place a user.
- **Network Service Node**: used to connect to external network setting.
- **LNET Router Nodes**: Manage file system metadata and transfer data to and from storage devices and applications.

1.3.2 Running Scripts on Blue Waters

The aprun command is used to specify the resources and placement parameters needed for the scripts at launch. At a high level, aprun is similar to mpiexec or mpirun. The following are the most commonly used options for aprun.

- -n: Number of processing elements(PEs) required for the application.
- -N: Number of PEs to place per node.
- -S: Number of PEs to place per NUMA node.
- -d: Number of CPU cores required for each PE and its threads.
- -j: Number of CPUs to use per compute unit .
- -cc: Bind PEs to CPU cores.

1.4 Programming with OpenACC

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator. The OpenACC interface is very close to OpenMP in its use of pragmas for compiler directives.

1.4.1 The `kernels` Construct

The `kernels` construct identifies a region of code that may contain parallelism, but relies on the automatic parallelization capabilities of the compiler to analyze the region, identify which loops are safe to parallelize, and then accelerate those loops

1.4.2 The `data` Construct

The `data` construct tells the compiler how to move the data from the host to device and vice versa. An example is provided below:

```
#pragma acc data copyin(b[0:br*bc],br,bc,bbw) copy(o[0:orr*occ],occ,orr)
```

The above command states “Copy `b` of size br*br starting at index 0 and along with other variables `br,bc,bbw` to the device. Once the created kernel has completed, copy `o` of size `orr*occ` starting at index 0 back out to the host device.” In C, arrays are pointers to the beginning of a sequence of elements. The compiler can find out the dimensions of locally declared arrays, but when arrays are dynamically allocated, the compiler only sees a pointer. We must specify the dimensions of the array to copy it properly, which is known as **array shaping**.

1.4.3 The `parallel` Construct

The `parallel` construct identifies a region of code that will be parallelized across OpenACC. By itself a parallel region is of limited use, but when paired with the `loop` directive the compiler will generate a parallel version of the loop for the accelerator. These two directives most often are, combined into a single `parallel loop` directive

2 PART A- Familiarize yourself with the code

The `src` folder of the tarball was provided to us with the `main.c` file. This was the main function which linked the 3 different sparse-dense matrix multiplication implementation in the files `serial_mm.cpp`, `omp.cpp` and `gpu_mm.cpp` The tarball was uploaded to Blue Waters and the relevant modules were loaded specially the `craype-accel-nvidia35` and `cudotoolkit` modules. After extracting the source code and compiling, one binariy was obtained as `mp3`. The run script was then submitted on the cluster and the output file was produced as `out-JOBID.txt`.

- **What does the application do ? What are the arguments to the binary ?**
  The codes provided to us do a dense-sparse matrix multiplication. In the `main.hpp` file the matrix structure is defined as well the functions are named. The `ERROR_THRESHOLD` is set to 1e-4 in the this file. In the `main.cpp` the functions that were named earlier and now defined. They are as follows:

  `matrix create_matrix(int rows, int cols)` - creates a new matrix taking the number of rows and columns as input. The details of the matrix structure are described in the answer below.

  `void init_matrix(matrix& mat, int bandwidth = -1)` - initializes the matrix with numbers based on the variable `bandwith`, if `bandwidth = -1` then a dense matrix is initialized by generating random numbers, otherwise based on the values of the bandwidth a banded sparse matrix is initialized.

  `void init_zero_matrix(matrix& mat)` - this is used to initialize the matrix in which the result of the matrix multiplication is to be stored.

  `void compare_results(matrix& ref, matrix& test)` - this function is used to compare the results of matrix multiplication obtained by serial, multi-threaded and GPU implementations.

  `void serial_mm(matrix& banded, matrix& dense, matrix& out)` - function to multiply the dense and banded matrix serially.

  `void omp_mm(matrix& banded, matrix& dense, matrix& out)`-function to multiply the dense and banded matrix using the multi-threading.

  `void gpu_mm(matrix& banded, matrix& dense, matrix& out)`-function to multiply the dense and banded matrix on the GPU.

  The `main.cpp` file also has the preprocessor directives to define the OpenMP and the GPU implementations on the matrix, as follows:

```
#if MP3_OMP
// OpenMP implementation.
start = get_time();
omp_mm(A, B, C_omp);
double omp_time = get_time() - start;
std::cout << "OMP time: " << omp_time << " s" << std::endl;
compare_results(C_serial, C_omp);
#endif

#if MP3_GPU
// GPU implementation.
start = get_time();
gpu_mm(A, B, C_gpu);
double gpu_time = get_time() - start;
std::cout << "GPU time: " << gpu_time << " s" << std::endl;
compare_results(C_serial, C_gpu);
#endif
```

The `Makefile` uses the directives while compiling them as:

```
mp3: *.cpp
        CC *.cpp -DMP3_GPU -O3 -hstd=c++11 -o mp3 -DMP3_GPU -DMP3_OMP
```

- **What configurations does the provided `run.sub` script run by default ? What kind of node does it run on? (You may check the Blue Waters users guide for this.)**
  It was observed that the arrays are declared dynamically and their sizes and their bandwidths, in case of an sparse matrix, are assigned only during the runtime by the `run.sub` file. A *for loop* is used as show below:

```
for size in 100 1000 2000 4000
do
for bandwidth in 5 10 50 100 300
do
echo "n=${size} bandwidth=${bandwidth}"
aprun -n 1 -N 1 -d 16 ./mp3 $size $bandwidth
done
done
```

  The matrix sizes are varied from 100 to 4000 and while keeping the matrix size constant the bandwidth is varied from 5 to 300, i.e, the sparsity is varied. The `run.sub` file specifies that `#PBS -l nodes=1:ppn=16:xk` uses the **XK** node as described above. It uses one node with 16 cores per node.

- **How is a matrix represented in the application ?**
  The matrix is represented as a one dimensional array that is store in the `structure` matrix. Initially, a set of pointers to pointers is created, which is equal to the number of rows of the molecule. Then the pointer at the 0th element of this row of pointer, is made to point at the whole of the one-dimensional array which has all the elements of the matrix. In the subsequent loop that follows the array of pointers is made to refer/point at the elements of the on dimensional array which make up the next rows – by using pointer arithmetics. A diagrammatic representation is shown below. This matrix representation helps in keeping the contents contiguous. However, it makes the later reallocation of individual rows more difficult.[4]

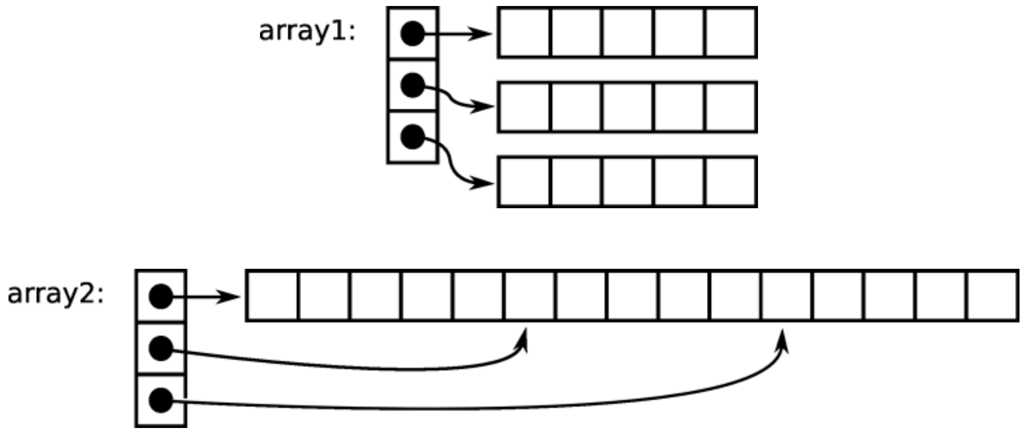


Figure 3: Structure of the matrix.

- **How is the memory for a dense and a band matrix laid out ?**
  The memory for the dense and the band matrix are laid out in the manner described above. Both the matrices have the same layout. However, the dense matrix has values in all of its rows/columns while the banded matrix had only a band of rows/columns that have any non-zero value.

- **Describe the basic algorithm being used to multiply the matrices.**

Algorithm 1 Algorithm to multiply a Banded matrix with a dense matrix.

```
for Each $int\ i = 0$ in rows of Banded Matrix do
    if $(i > Bandwidth)$ then
        int $start = i - Bandwidth$
    else
        int $start = 0$
    end if
    if $(i + Bandwidth <$ rows of Banded Matrix $)$ then
        int $end = i + Bandwidth$
    else
        int $end =$ rows of Banded Matrix
    end if
    for Each $int\ j = 0$ in cols of Banded Matrix do
        $double\ sum = 0$
        for Each $int\ k = start$ , $end$ do
            $sum += $ Banded_Matrix.$a[i][k] *$ Dense_Matrix.$a[k][j]$;
        end for
        Output_Matrix.$a[i][j]$;
    end for
end for
```

3 PART B- Multi-threaded OpenMP implementation

The OpenMP implementation of the dense-sparse matrix multiplication is shown below.

```
#include "main.hpp"

void omp_mm(matrix& banded, matrix& dense, matrix& out) {
        if (banded.bandwidth == -1) {
                // Fall-through case for when banded is actually dense.
                dense_mm(banded, dense, out);
                return;
        }
        #pragma omp parallel shared(banded,dense,out)
        {
         #pragma omp for collapse(2) schedule(static,2*banded.bandwidth)
        for (int i = 0; i < banded.rows; ++i) {
                for (int j = 0; j < banded.cols; ++j) {
                        double sum = 0.0;
                        int start = (i > banded.bandwidth) ? i - banded.bandwidth : 0;
                        int end = (i + banded.bandwidth < banded.rows) ? i +\
                         banded.bandwidth + 1 : banded.rows;
                        for (int k = start; k < end; ++k) {
                                sum += banded.a[i][k] * dense.a[k][j];
                        }
                        out.a[i][j] = sum;
                        }
                }
        }
}
```

3.1 Overview of Implementation Choices

While coding the OpenMP implementation, the following implementation choices were made so as to optimize our code

- The whole block of the multiplication code was enclosed in the `#pragma omp parallel shared (banded,dense,out)` construct. We specify that the three matrices are shared in the parallel region.
- The `#pragma omp for collapse(2)` construct was use to collapse the two for loops into one large loop.
- It was specifed to OpenMP that it statically assigns loop iterations to threads. This is because it was observed that all iterates do about the same amount of work (slightly less for the first and last one). Also a chunk size of `bandwidth*2+1` was specified as it will minimize cache misses and improve prefetch in both the matrices.

3.2 Optimization report on the OpenMP code

Using the `-h msgs` option in the Make file the compiler optimization report for the written OpenMP code were generated. The optimization details are presented below:

- The block of codes starting from `#pragma omp parallel` region to the end of its braces was multi-threaded by the compile.
- As per our directive the compiler collapsed the nested for loops.
- The compiler also carried out loop portioning and unrolling of the above loops.

3.3 OpenMP: Experiment with varying matrix sizes and bandwidths

The OpenMP code above was used for dense-sparse matrix multiplication using different matrix sizes and bandwidths. The time taken for different sizes and bandwidths are reproted in the table below. The plots for the same are reported.

Table 1: Time taken by the OpenMP code for sparse-dense matrix multiplication of different bandwidths.

No.	Size	Bandwidth	Serial Time Taken in sec	OpenMP Time Taken in sec
1.	100	5	0.0001487	0.0009268
		10	0.0003043	0.0009129
		50	0.001360	0.00100893
		100	0.001829	0.001033
		300	0.001823	0.001093
2.	1000	5	0.02030	0.0054
		10	0.03579	0.006523
		50	0.3658	0.0307919
		100	0.7434	0.0540922
		300	2.1204	0.334784
3.	2000	5	0.08029	0.01634
		10	0.1406	0.02013
		50	1.4834	0.118498
		100	3.32909	0.271123
		300	9.74946	1.68718
4.	4000	5	0.32033	0.059924
		10	0.567506	0.076257
		50	6.49324	0.528655
		100	13.6565	1.13025
		300	44.20870	7.87376

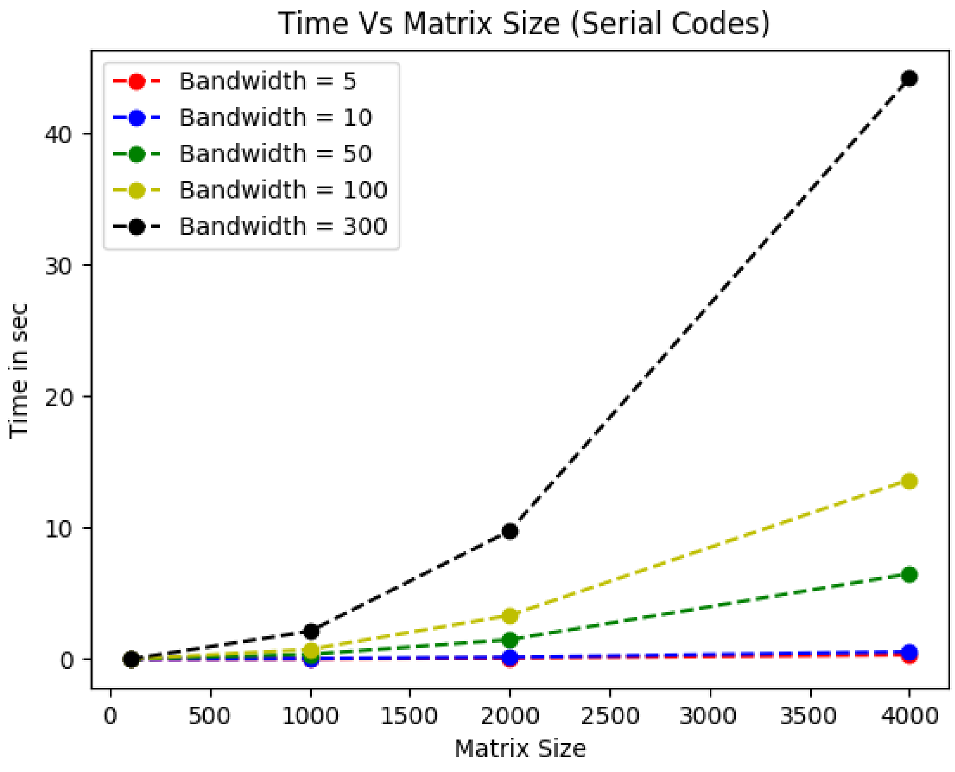


Figure 4: Time taken by Serial Codes

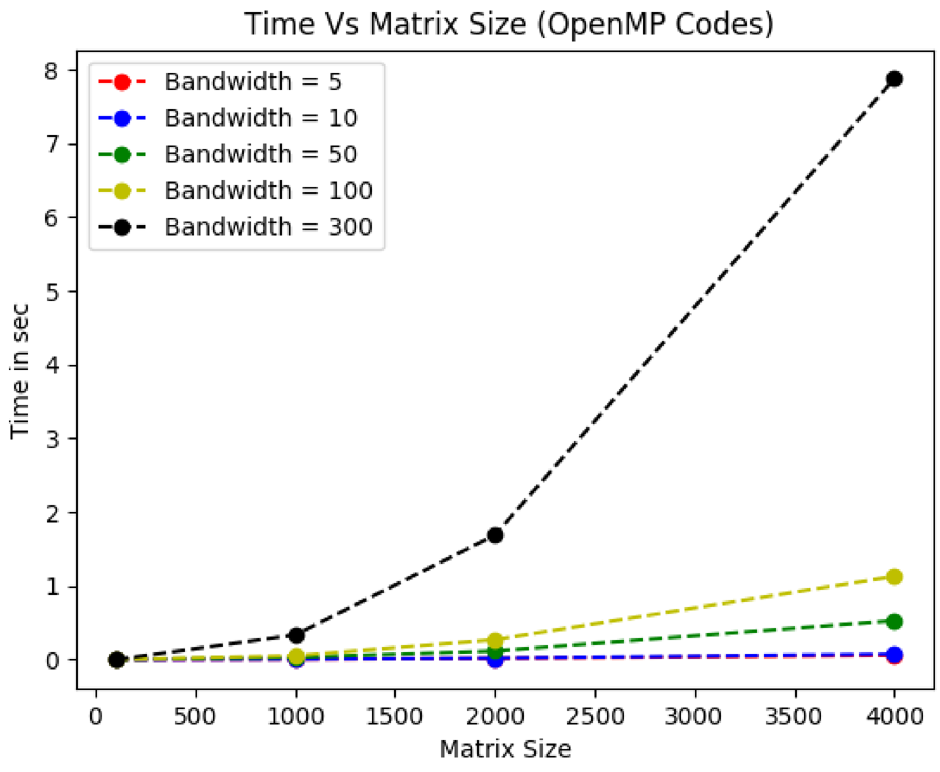


Figure 5: Time taken by OpenMP Codes

3.3.1 OpenMP Conclusion

- From the data and the graph presented above, it was seen that as the size of the matrix increases the time taken by both the serial version and the OpenMP increases. However, we observe that time taken by the parallel OpenMP code is $\sim$ 7-13 times faster than the serial version - when using 16 threads.
- It was observed that for a given constant matrix size as the bandwidth increase the time taken for multiplication increases.
- It was observed that the OpenMP implementation is always faster than the serial version. (This might not always be true for GPU implementation which was studied below.)

4 PART C- GPU implementation

Using the better implemented version in Part B , the tiled version was implemented. The code is shown below. It can be seen from the code below that every dimension was tiled. Different experiments were run using different matrix sizes and the results are shown below:

```
void gpu_mm(matrix& banded, matrix& dense, matrix& out) {
        double* b= banded.a[0];
        int br=banded.rows;
        int bc=banded.cols;
        int bbw=banded.bandwidth;
        double* d= dense.a[0];
        int dr=dense.rows;
        int dc=dense.cols;
        int dbw=dense.bandwidth;
        double* o= out.a[0];
        int orr=out.rows;
        int occ=out.cols;
        #pragma acc data copyin(b[0:br*bc],br,bc,bbw, d[0:dr*dc],dr,dc,dbw)\
         copy(o[0:orr*occ],occ,orr)
        #pragma acc kernels
        #pragma acc parallel loop
        for (int  i = 0; i < br; ++i) {
                int start = (i > bbw) ? i - bbw : 0;
                int end = (i + bbw < br) ? i + bbw + 1 : br;
                for (int j = 0; j < bc; ++j) {
                        double sum = 0.0;
                        for (int k = start; k < end; ++k) {
                                sum += b[i*bc+k] * d[k*bc+j];
                        }
                        o[i*occ+j]=sum;
                        }
        }
}
```

4.1 Optimization report on the GPU code

Using the `-h msgs` option in the Make file the compiler optimization report for the written GPU implementation code using OpenACC were generated. The optimization details are presented below:

- The compiler created a data region for the block of codes enclosed by the `pragma acc kernels` region.
- The line containing the `#pragma acc data copyin` and `copy` was placed on the accelerator by the compiler.
- The compiler allocated memory and and copied the following variables to the accelerator. These variables were also copied out at the end of the execution on the accelerator.
  - The `out` matrix.
  - The `out.cols` variable.
  - The `out.rows` variable.
- The compiler allocated memory and and copied the following variables to the accelerator. These variables were freed at the end of the execution on the accelerator.
  - The `dense` and `banded` matrices.
  - The `dense.cols` and `banded.cols` variables.
  - The `dense.rows` and `banded.cols` variables.

- The compiler also partition the loop across 128 threads within a thread block.
- The loop was partially vectorized by the compiler and was unrolled 6 times. It was also partially vector pipelined

4.2 Overview of Implementation Choices

While coding the GPU implementation using OpenACC, the following implementation choices were made so as to optimize our code:

- First of all, a deep copy was performed of the structure `matrix` was implemented on the accelerator. In C, arrays are pointers to the beginning of a sequence of elements. The compiler can find out the dimensions of locally declared arrays, but when arrays are dynamically allocated, the compiler only sees a pointer. We must specify the dimensions of the array to copy it properly, which is known as **array shaping**.
- Coping to and fro from the accelerator has a overhead and so the best possible way to decrease this was to copy `dense` and `banded` matrix structures only to the accelerator and not to copy them back. However, the structure `out` was to be copied in and out as well.
- Using the `#pragma acc parallel loop` directive it was explicitly specified to the compiler that a parallel region follows.
- It was noticed that the variable `start` and `end` need to be computed only for the outer loop so it was moved out of the second loop. This reduced computation overhead.

4.3 GPU: Experiment with varying matrix sizes and bandwidths

The GPU code above was used for dense-sparse matrix multiplication using different matrix sizes and bandwidths. The time taken for different sizes and bandwidths are reproted in the table below. The plots for the same are reported.

Table 2: Time taken by the GPU code for sparse-dense matrix multiplication of different bandwidths.

No.	Size	Bandwidth	Serial Time Taken in sec	GPU Time Taken in sec
1.	100	5	0.0001487	0.249207
		10	0.0003043	0.25349
		50	0.001360	0.2469453
		100	0.001829	0.25228
		300	0.001823	0.250533
2.	1000	5	0.02030	0.36073
		10	0.03579	0.357449
		50	0.3658	0.37794
		100	0.7434	0.405111
		300	2.1204	0.485001
3.	2000	5	0.08029	0.541098
		10	0.1406	0.554373
		50	1.4834	0.629734
		100	3.32909	0.726667
		300	9.74946	1.07188
4.	4000	5	0.32033	1.08689
		10	0.567506	1.12567
		50	6.49324	1.4302
		100	13.6565	1.81142
		300	44.20870	3.39229

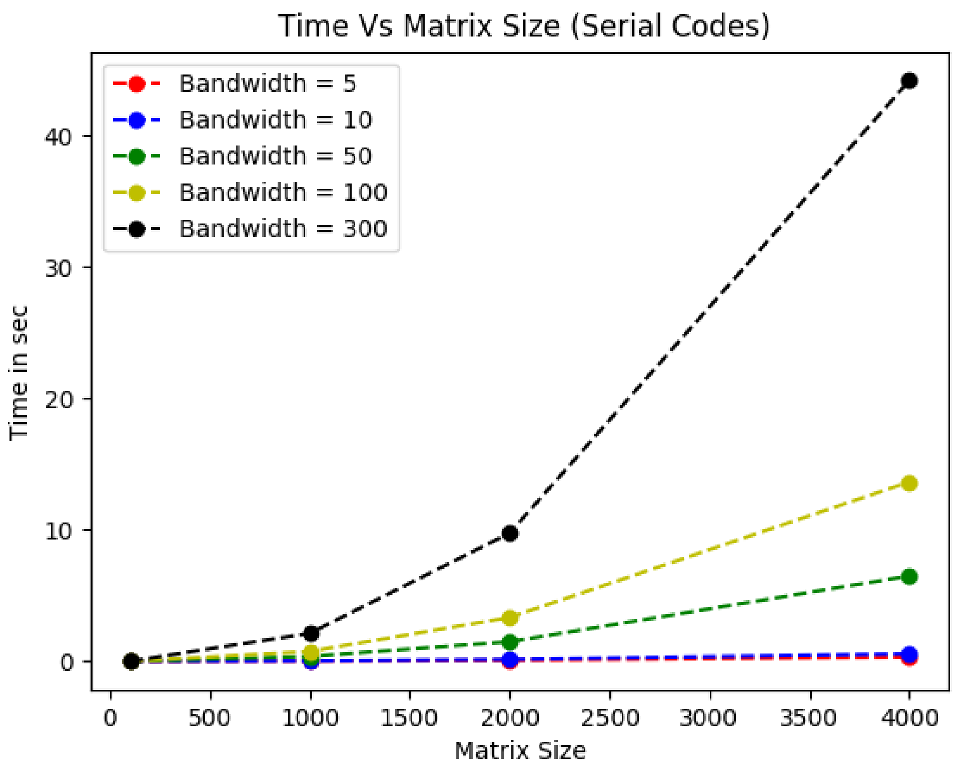


Figure 6: Time taken by Serial Codes

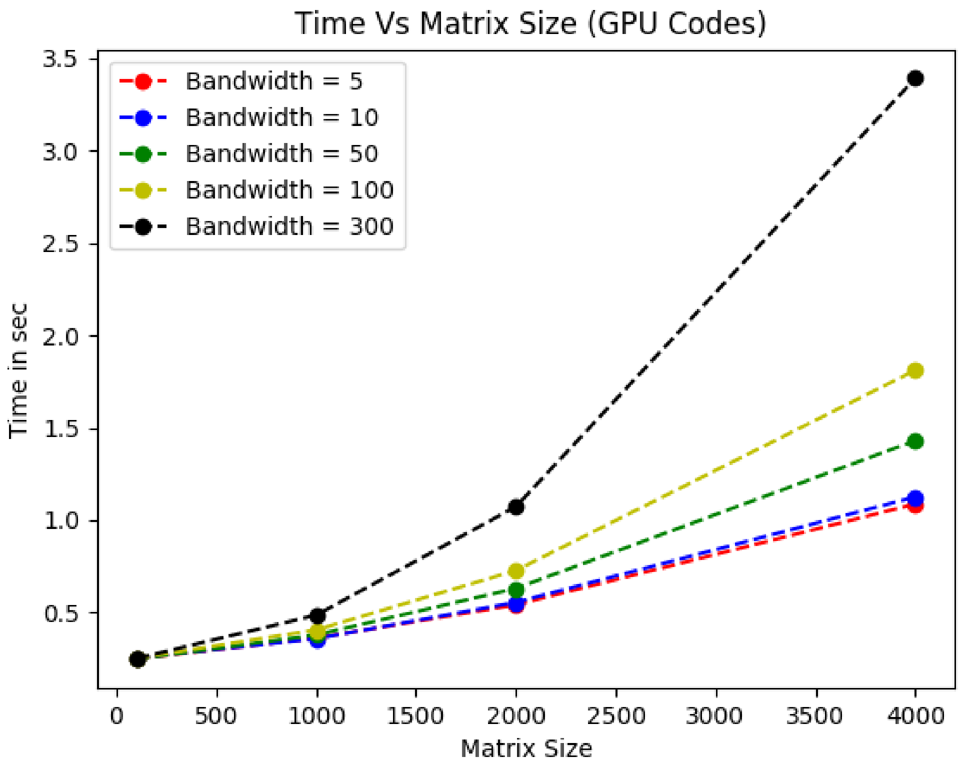


Figure 7: Time taken by GPU Codes

4.3.1 GPU Conclusion

- From the data and the graph presented above, it was seen that as the size of the matrix increases the time taken by both the serial version and the GPU implementation using OpenACC increases. However, it was observed that time taken by the GPU implementation was higher than both the serial and OpenMP implementation for matrices upto size 1000 and a bandwidth of 100.
- However, for matrices of size bigger that 2000 and of higher than 100 bandwidth the GPU was faster.
- Detailed comparisons and analysis are presented in the section below.

5 PART D- Evaluation and Conclusions

In this section we compare the performances of the Serial vs OpenMP vs GPU implementation of the codes. We ran the default configurations provided to us and the results obtained for each of the three cases are provided in the Table below for comparison. Following the tables are plots that compare the time taken by the three implementations as function of matrix size with constant bandwidth. Finally the speed up obtained over the serial version it plotted as a bar graph.

Table 3: Comparison of time taken for sparse-dense matrix multiplication of different bandwidths and matrix sizes.

No.	Size	Bandwidth	Serial Time in sec	OpenMP Time in sec	GPU Time in sec
1.	100	5	0.0001487	0.0009268	0.249207
		10	0.0003043	0.00091	0.25349
		50	0.001360	0.0010089	0.2469453
		100	0.001829	0.0010	0.25228
		300	0.001823	0.00109	0.250533
2.	1000	5	0.02030	0.0054	0.36073
		10	0.03579	0.006523	0.357449
		50	0.3658	0.0307919	0.37794
		100	0.7434	0.0540922	0.405111
		300	2.1204	0.334784	0.485001
3.	2000	5	0.08029	0.01634	0.541098
		10	0.1406	0.02013	0.554373
		50	1.4834	0.118498	0.629734
		100	3.32909	0.2767	0.726667
		300	9.74946	1.687	1.07188
4.	4000	5	0.32033	0.059924	1.08689
		10	0.567506	0.076257	1.12567
		50	6.49324	0.528	1.4302
		100	13.6565	1.13025	1.81142
		300	44.20870	7.87376	3.39229

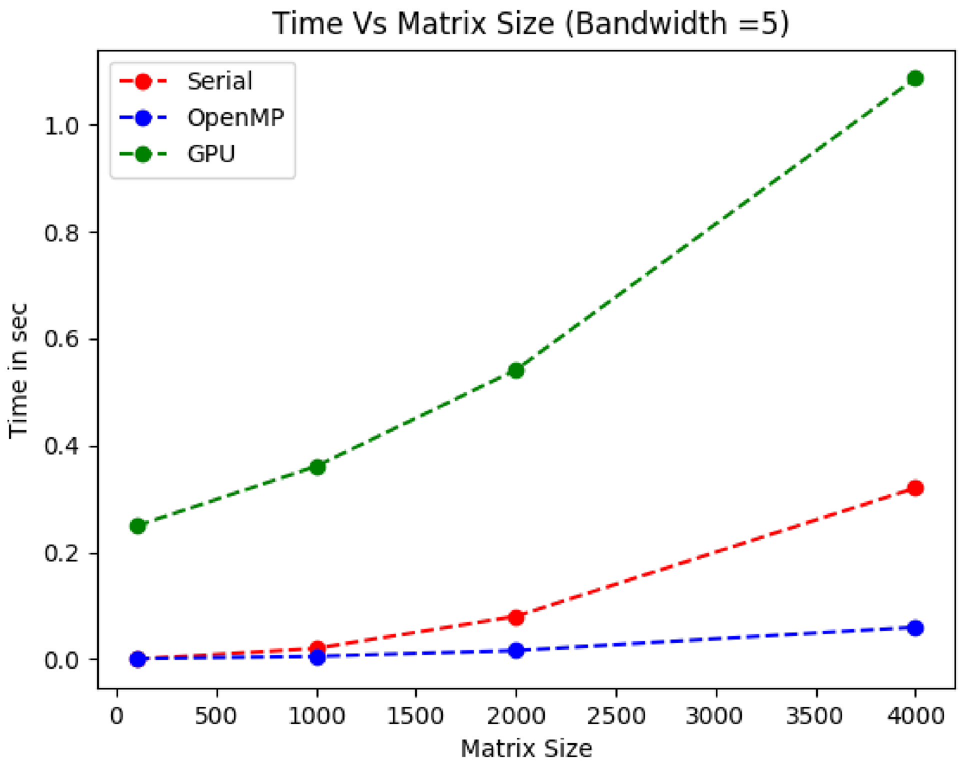


Figure 8: Serial Codes vs OpenMP vs GPU implementation. Bandwidth = 5

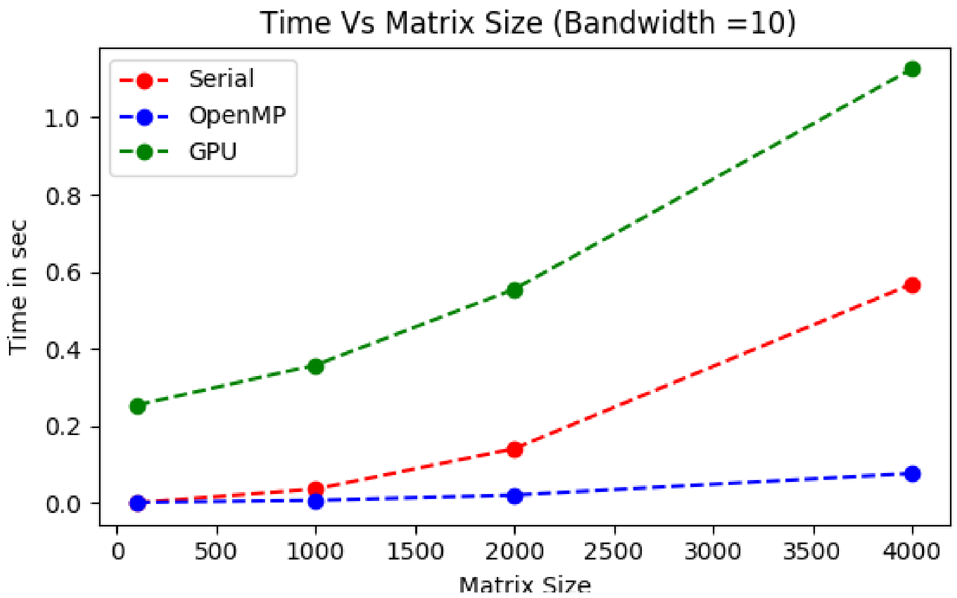


Figure 9: Serial Codes vs OpenMP vs GPU implementation. Bandwidth = 10

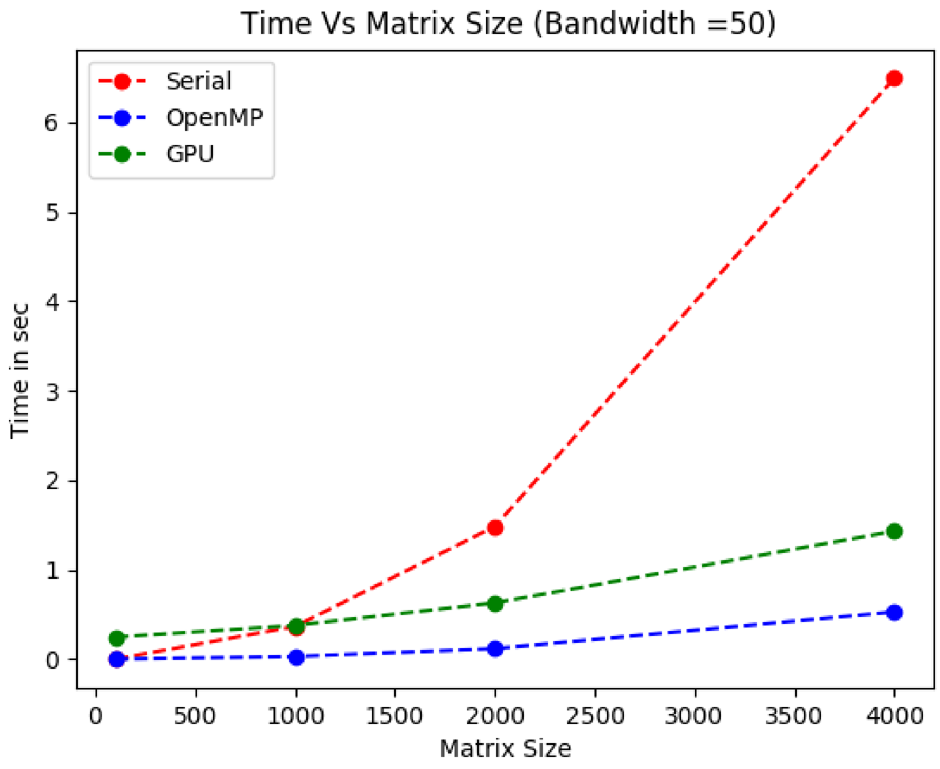


Figure 10: Serial Codes vs OpenMP vs GPU implementation. Bandwidth = 50

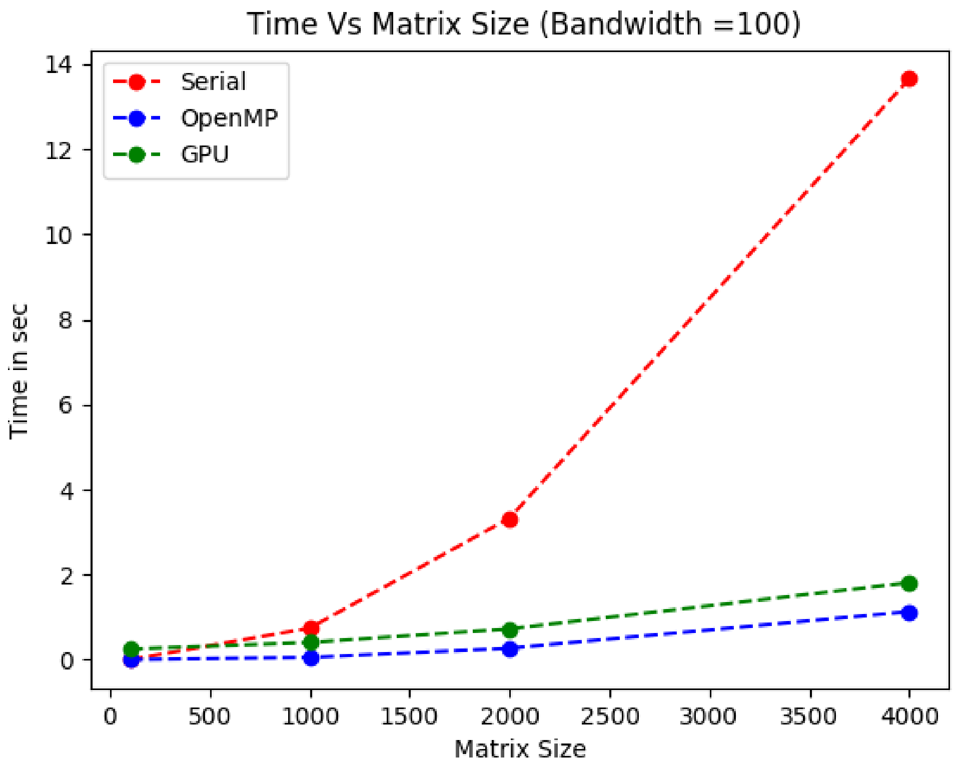


Figure 11: Serial Codes vs OpenMP vs GPU implementation. Bandwidth = 100

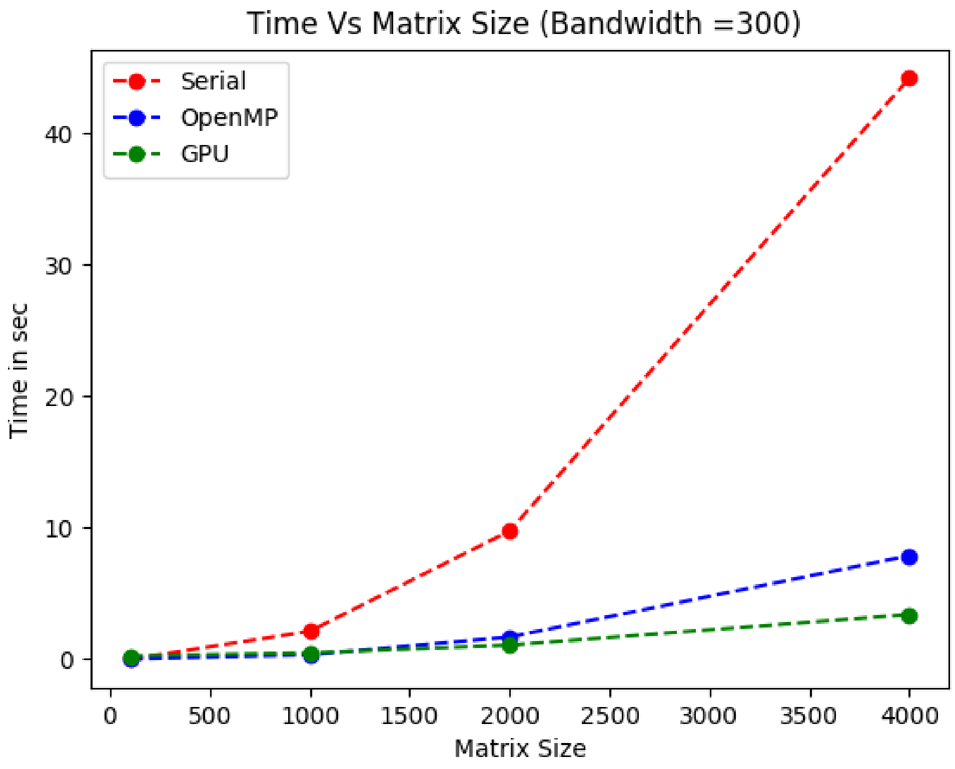


Figure 12: Serial Codes vs OpenMP vs GPU implementation. Bandwidth = 300

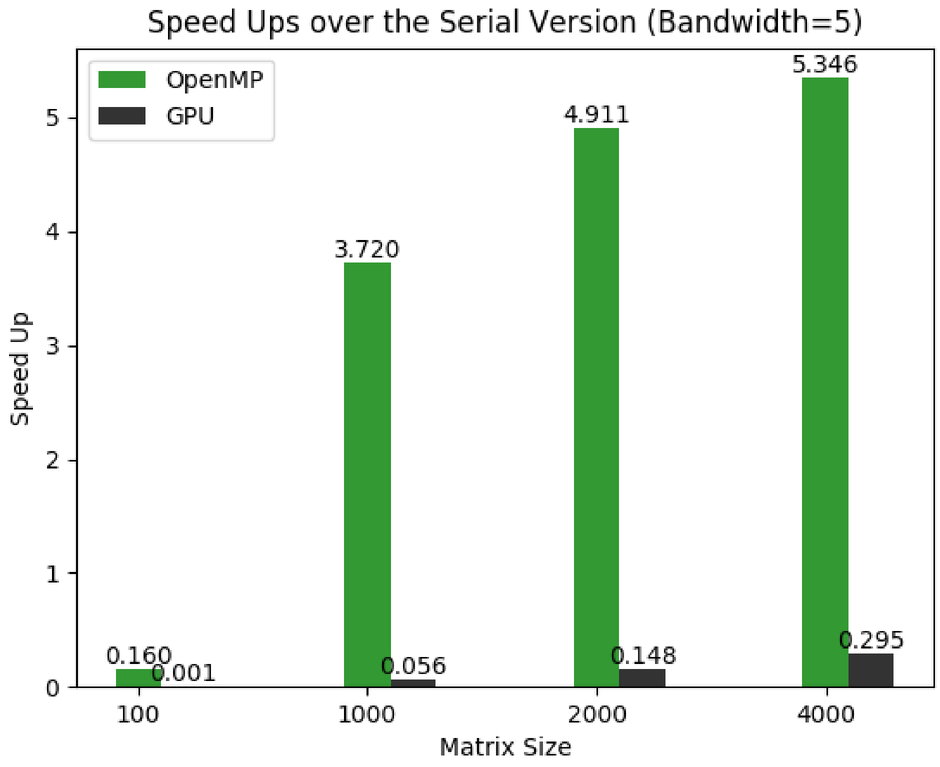


Figure 13: Speed Up over the Serial Version. Bandwidth = 5

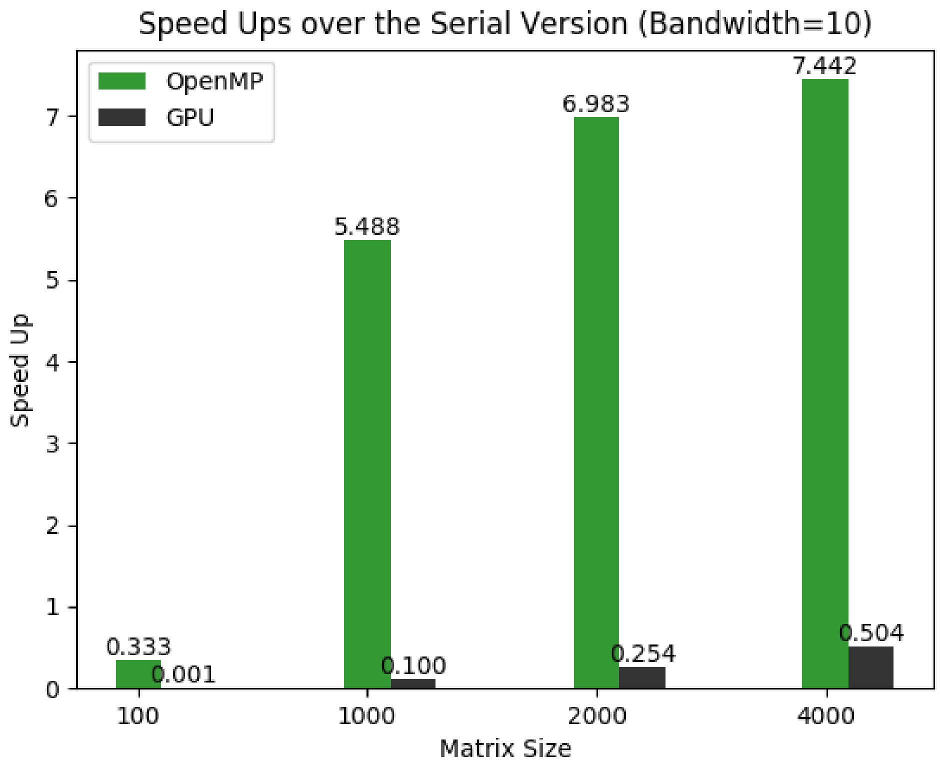


Figure 14: Speed Up over the Serial Version. Bandwidth = 10

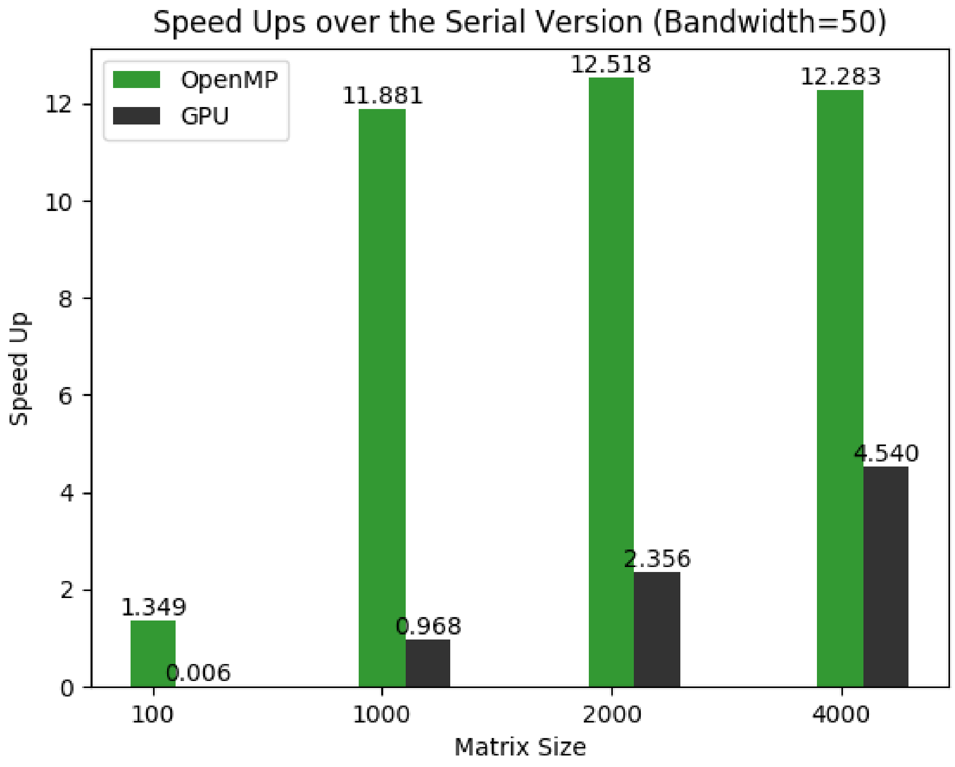


Figure 15: Speed Up over the Serial Version. Bandwidth = 50

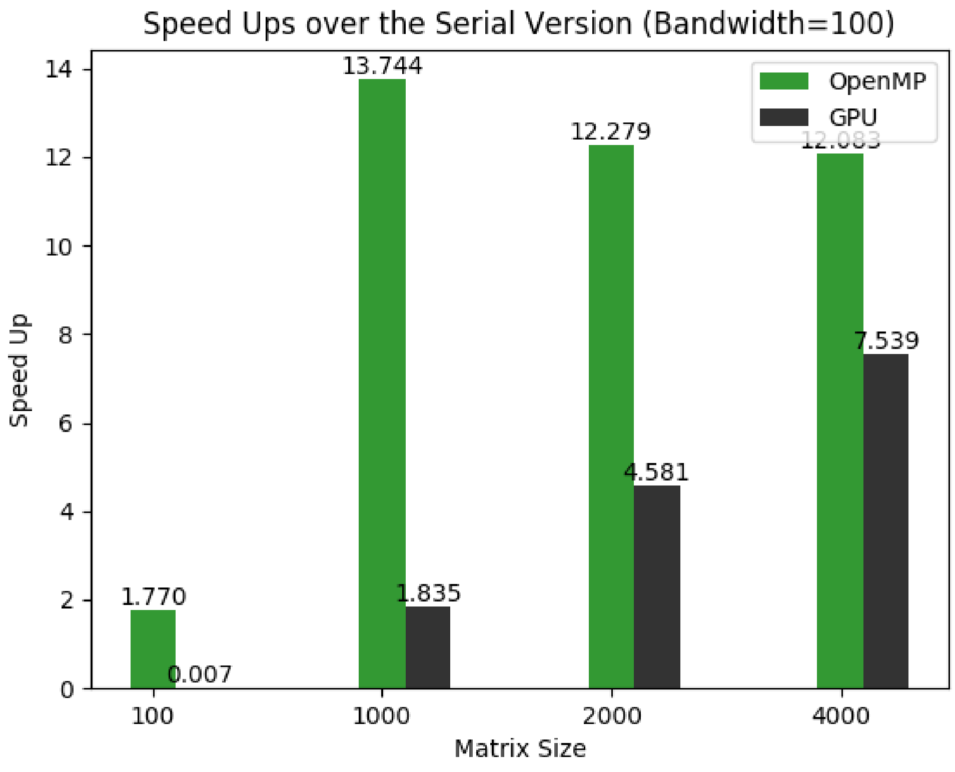


Figure 16: Speed Up over the Serial Version. Bandwidth = 100

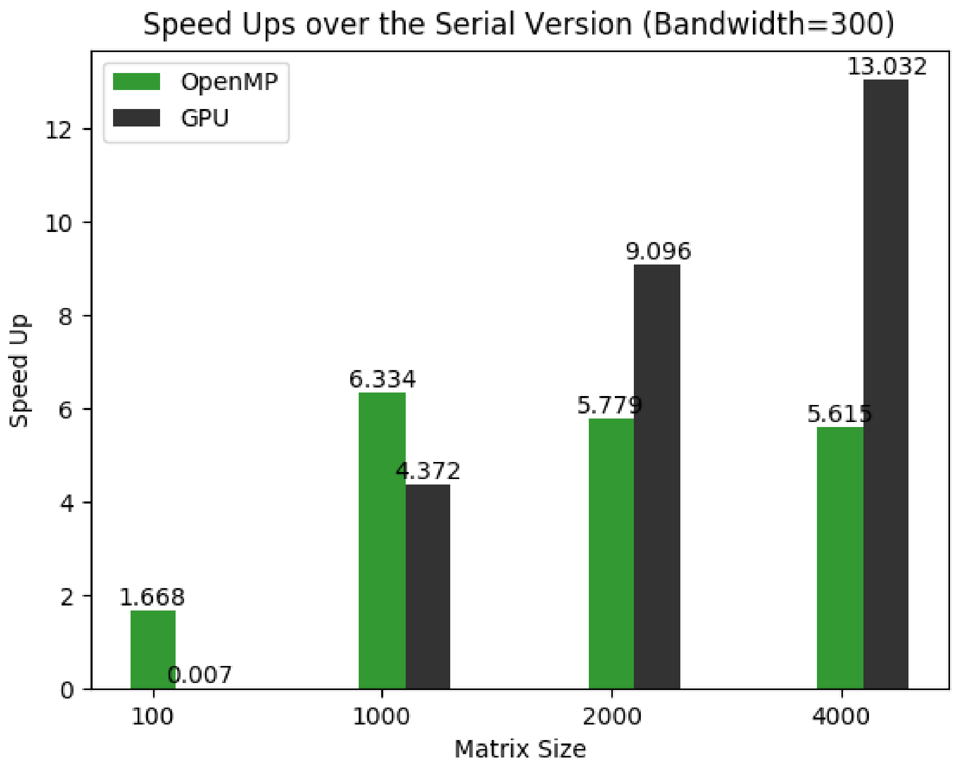


Figure 17: Speed Up over the Serial Version. Bandwidth = 300

5.0.1 Answers to Questions

- **When is the multi-threaded implementation fastest? When is the GPU implementation fastest?**

- It was observed from the plots above that the multi-threaded implementation is the fastest (~ 13 times faster than serial) when the matrix size is 1000 and 2000 for a bandwidth of 100. For bandwidth larger than 100 we actually observe that the speed up for the OpenMP is $\sim$ 6-7 times the serial codes.

- GPU implementation is the fastest for the largest matrix size and bandwidth used in this experiment of 4000 and 300 respectively. We observe a ~ 13 times speed up here as well. In general, as the matrix becomes bigger and denser the speed up increases (over the serial version). Since, it was obsereved that as the matrix size and bandwidth increased, the speed up observed by the GPU implementation also increased – the `run.sub` file was modified to carry out a few more experiments with increased bandwidth and matrix size. The modifications are listed below:

```
#!/bin/bash
#PBS -l walltime=00:300:00   -- The runtime was increase to 300
#PBS -l nodes=1:ppn=16:xk
#PBS -N cs420mp3
#PBS -o out-$PBS_JOBID.txt
#PBS -e err-$PBS_JOBID.txt

cd $PBS_O_WORKDIR

module load craype-accel-nvidia35
module load cudatoolkit

export OMP_NUM_THREADS=16

for size in  5000 7000
do
                for bandwidth in 500 1000 2000 5000
                do
                                echo "n=${size} bandwidth=${bandwidth}"
                                aprun -n 1 -N 1 -d 16 ./mp3 $size $bandwidth
                done
done
```

Using the above configurations the following results were obtained:

Table 4: Time taken by Serial vs OpenMP vs GPU code for sparse-dense matrix multiplication of different bandwidths and sizes.

No.	Size	Bandwidth	Serial Time in sec	OpenMP Time in sec	GPU Time in sec
1.	5000	500	154.183	27.018	9.57351
		1000	851.521	56.7169	20.6648
		2000	1561.44	103.772	39.3366
		5000	1561.44	103.772	39.3366
2.	7000	500	360.014	56.442	20.1272
		1000	1763.61	111.063	43.3871
		2000	3285.33	215.975	84.9625
		5000	6159.86	453.944	146.6

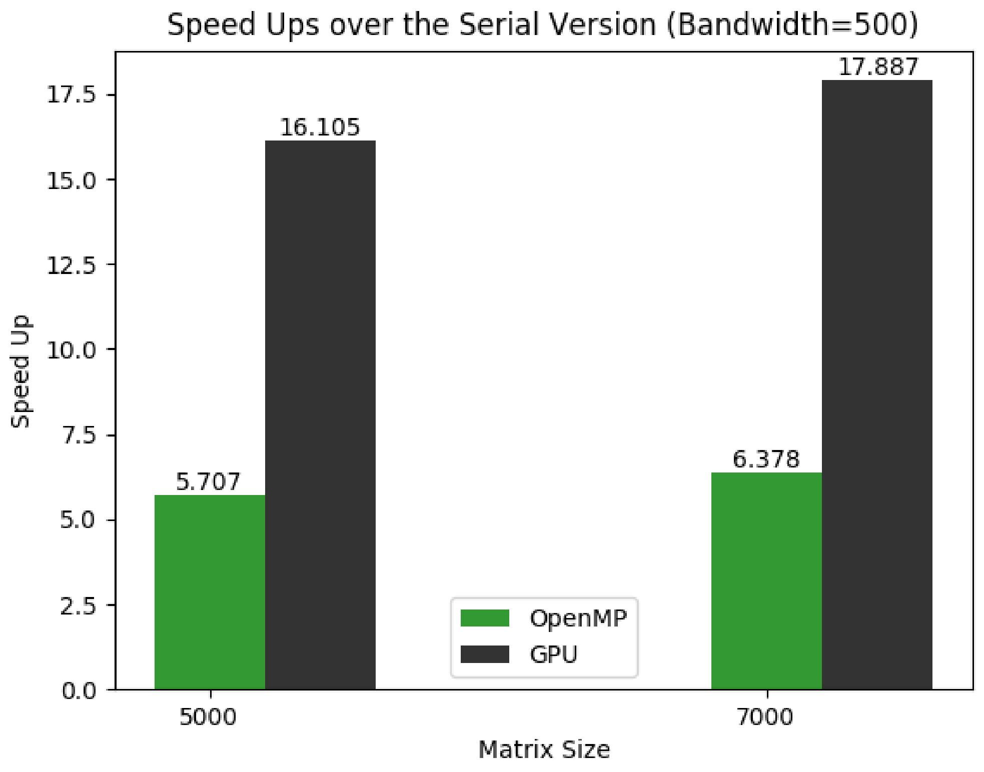


Figure 18: Speed Up over the Serial Version. Bandwidth = 500

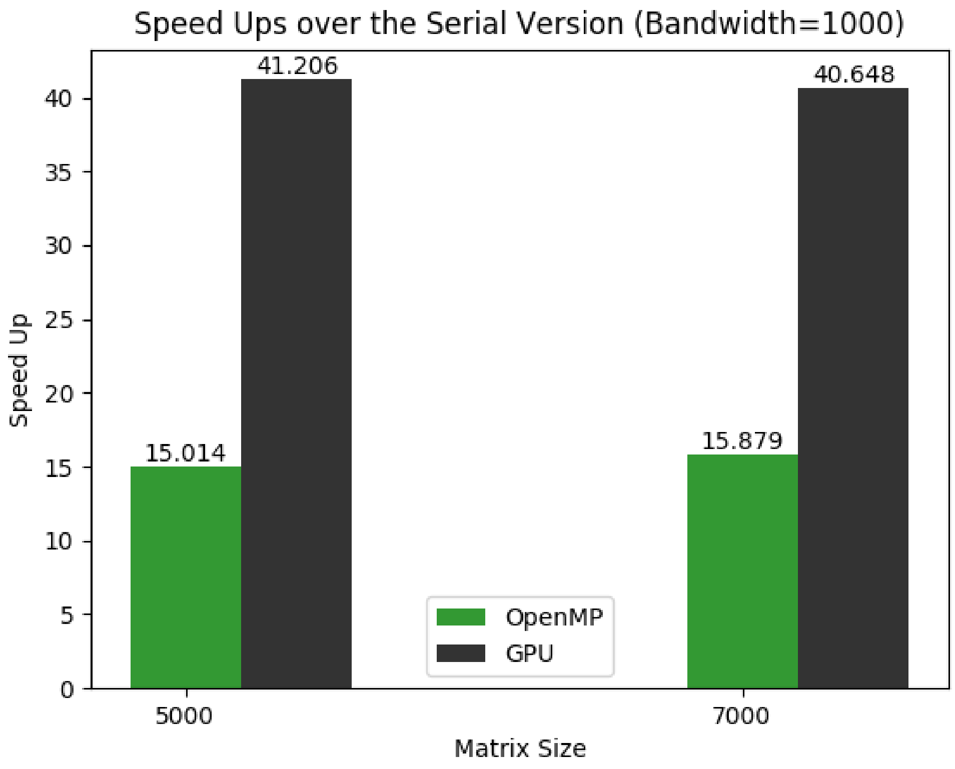


Figure 19: Speed Up over the Serial Version. Bandwidth = 1000

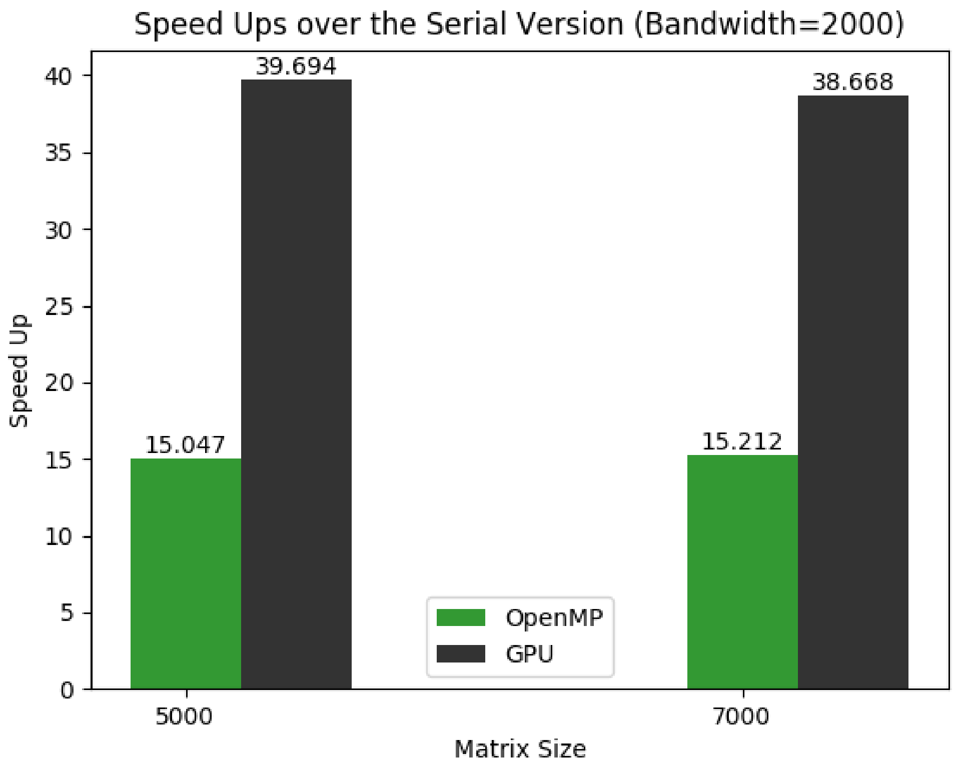


Figure 20: Speed Up over the Serial Version. Bandwidth = 2000

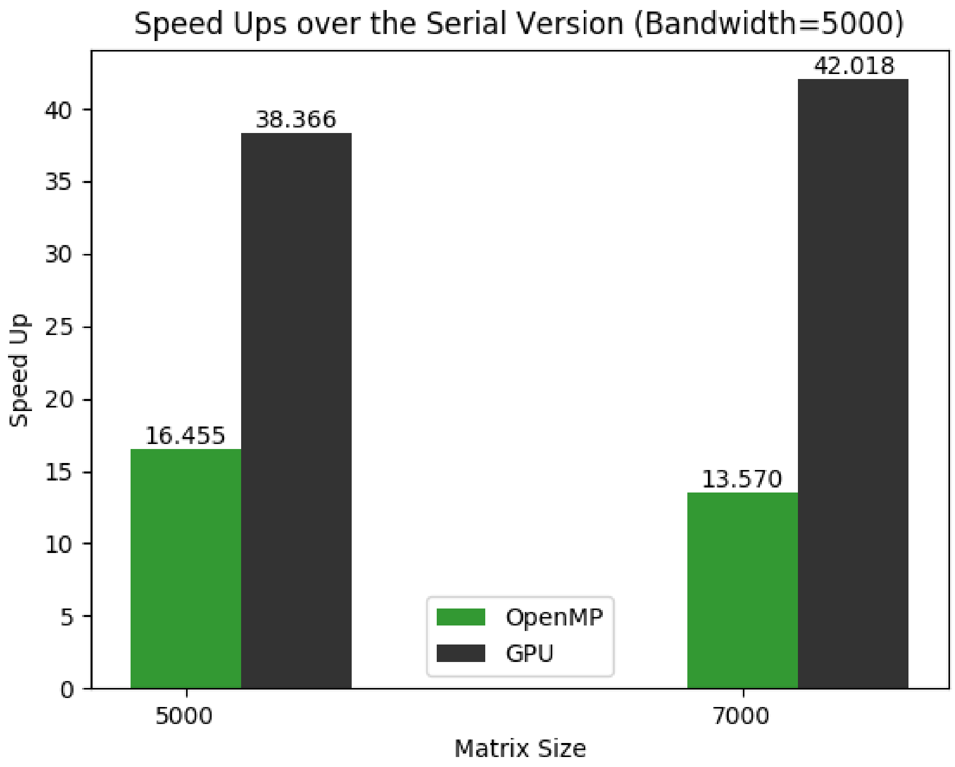


Figure 21: Speed Up over the Serial Version. Bandwidth = 5000

- We observe a speed up of over $\sim$ 40 times for the GPU implementation when the matrix size is 7000 and the bandwidth is 5000. However, the speed up over the serial version of the OpenMP code was always between $\sim$ 13-16 times.
- **What are the overheads involved in your multi-threaded implementation? In your GPU implementation?**
  The overheads associated with the multi-threaded implementation are:
  - overhead incurred in creating the threads
  - overhead incurred in terminating/merging threads
  - overhead incurred in switching the CPU between threads (load sharing).
  - overhead in saving register state of a thread when suspending it, and restoring the state when resuming it.
  - overhead incurred in saving and restoring a thread's cache state.

  The overheads associated with the GPU implementation are:
  - copying the data to and from the GPU (Memory Transfer Overhead)
  - allocating memory using on GPU can be more expensive than on CPU, especially when using certain CUDA functions– using `cudaMallocHost` is significantly more expensive than using `malloc` (Memory management Overhead)
  - conditionals/branching in the GPU code can affect performance.
  - executing a kernel on the GPU is a relatively expensive operation and kernel calls are generally asynchronous.

- **Explain the speedups you observe in your GPU implementation relative to your OpenMP implementation.**
  It was observed that GPU implementation became faster than the OpenMP implementation as the matrix size and bandwidth increased (when the matrix size > 1000 and bandwidth > 300, a speed up of ∼2 was observed), however initially for a matrix size of around <1000 the OpenMP was much faster than the GPU implementation. The GPU was faster in the compute intensive cases of the experiment, this was because:

  – SIMT execution model of GPU is faster than the SIMD model used my CPUs(OpenMP).

  – GPU's have comparatively a lot more number of ALUs and these are more heavily pipelined . They also have small caches to improve memory throughput.

  We observed that GPUs are basically massively parallel computers. They work well on problems that can use large scale data decomposition and GPU's offer orders of magnitude speedups on these problems. The speed up of the GPU over OpenMP from the experiment that was carried out using the modified `run.sub` file is reported below.

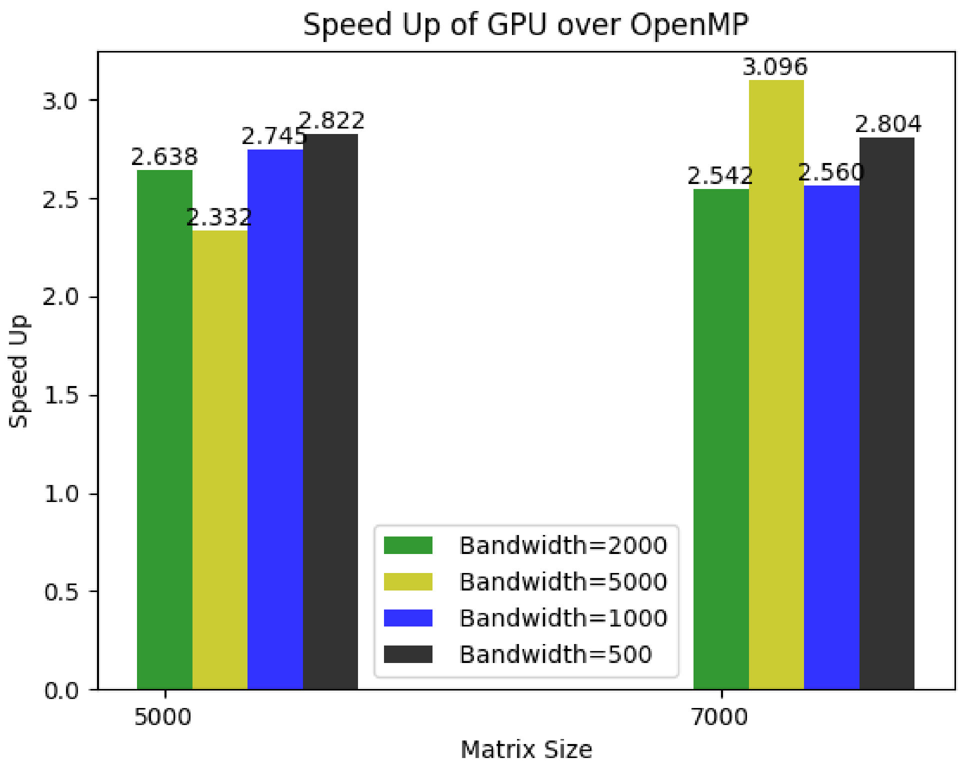


Figure 22: Speed Up over the OpenMP code by the GPU for different bandwidths and matrix sizes.

6 References

[1] https://bluewaters.ncsa.illinois.edu/user-guide accessed on 13th December.

[2] http://icl.cs.utk.edu/projects/papi/wiki/PAPIC:Overview.

[3] CS-420 Notes by Prof. Mark Snir

[4] http://c-faq.com/aryptr/dynmuldimary.html accessed on 13th December.