General Matrix-Matrix Multiplication

Small Scale Parallel Programming Report

M.Sc. in Software Engineering for Technical Computing

Aman Makkar
S321839

4th May 2021

Abstract

This report has been compiled on the work carried out to design, develop, test and implement two kernels using OpenMP and Cuda using the C programming language. The programs will be written to multiply two square matrices where the size of the matrix will be an argument of the kernel. Both kernels will include a serial program to test for the correctness of the results for the programs written in OpenMp and Cuda. Performance tests will be carried out on the Crescent High-Performance Computer and the results will be analysed for different matrix sizes.

Contents

1. Introduction

The purpose of this report is to provide a breakdown of the work carried out to write a kernel, that multiplies two General matrices that are both square. The kernels have been written using the C programming language in OpenMP and CUDA versions. A breakdown of the design has been provided along with the design specification and a test plan. Matrix-matrix multiplication is of great importance within the Scientific and High-Performance Computing with it gaining further importance in machine learning(Huang & van de Geijn, 2016). General matrix-matrix multiplication is a simple enough concept to explain, and it can be useful to understand how a piece of code can be optimised however, it is a challenging task to be able to fully optimise the code.
There are various ways of implementing a parallel matrix-matrix multiplication code, one of those methods is using the directive-based programming model. The loop to carry out the work multiplication work can be mapped in different ways, resulting in performance difference, the loop can be scheduled in different ways such as, letting the compiler do the loop scheduling or these can be explicitly provided by the developer when writing the code (Kunkel et al., 2016).
OpenMP is a shared memory parallel programming API, that supports C, C++ and fortran. It is a set of compiler directives that control the run time behaviour of the program and use all the available CPUs on a given PC. CUDA on the other hand, carries out the work on the GPU. The memory must be explicitly declared on the GPU, and the information must be explicitly exchanged between the CPU and the GPU however, although it is a cumbersome task the benefits of using GPU for high performance computing outweigh the inconvenience and time required to write the program.

2. Problem Statement

The task of this assignment is to develop a GEMM (General Matrix-Matrix) product kernel. The given equation is $C \leftarrow A \cdot B$ and all three are square matrices. The size of the matrix will be provided as the argument of the kernel and the code will need to be parallelised using OpenMP and CUDA APIs.

3. OpenMP

OpenMP is a set of directives that are inserted into the serial code, which leads the compiler to carry out the parallelisation. OpenMP works on multicore machines that have a shared memory, and it would not be accessible if the memory is not shared (*12 Introduction to OpenMP - Multicore DSP*, n.d.). The regions that can parallelised is for the developer to design and provide the directive in the code for the compiler to parallelise the code. The design of the OpenMP provides various features such as Standardisation, ease of use and portability (for C, C++, and Fortran).
OpenMP has three main components, as can be seen in figure 1 below (*12 Introduction to OpenMP - Multicore DSP*, n.d.):

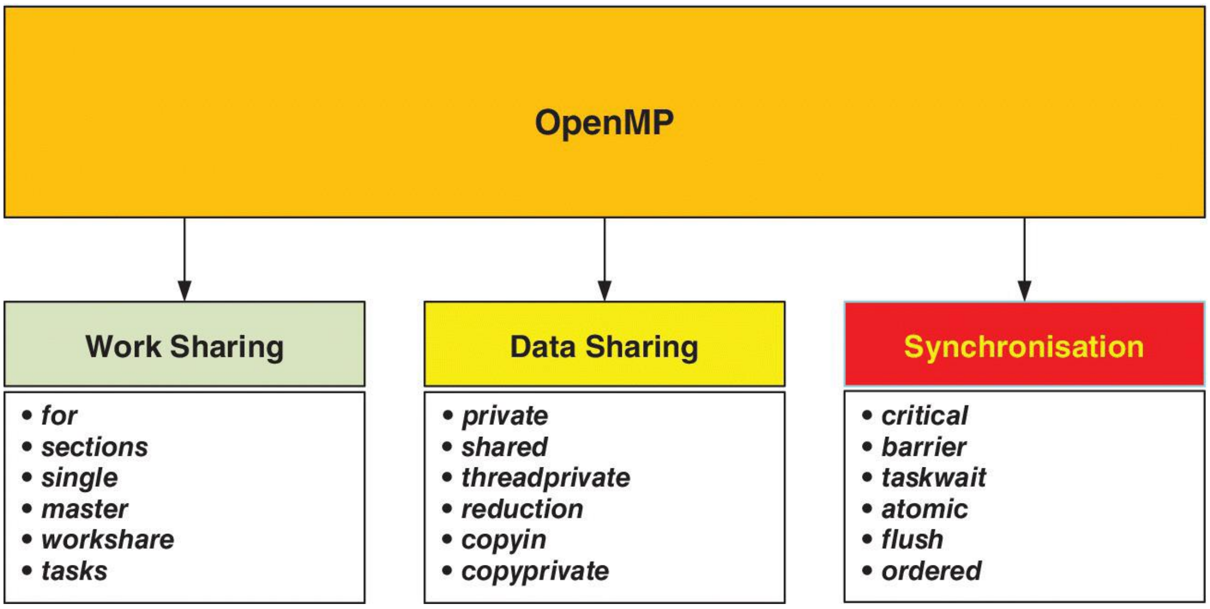

Figure 1: OpenMP components

4. CUDA

Cuda is a programming model with some extensions to the C programming language however, managing the memory explicitly can be quite cumbersome. It works in heterogenous environment where both a CPU and a GPU are available. Where the CPU memory is referred to as the host and the GPU as the device memory (Chapter 2: CUDA Programming Model - Professional CUDA C Programming, n.d.). The host can work independently of the device, once the kernel has been launched on the device, the control is returned to the host that can carry out tasks in parallel while waiting for the device to complete its tasks.

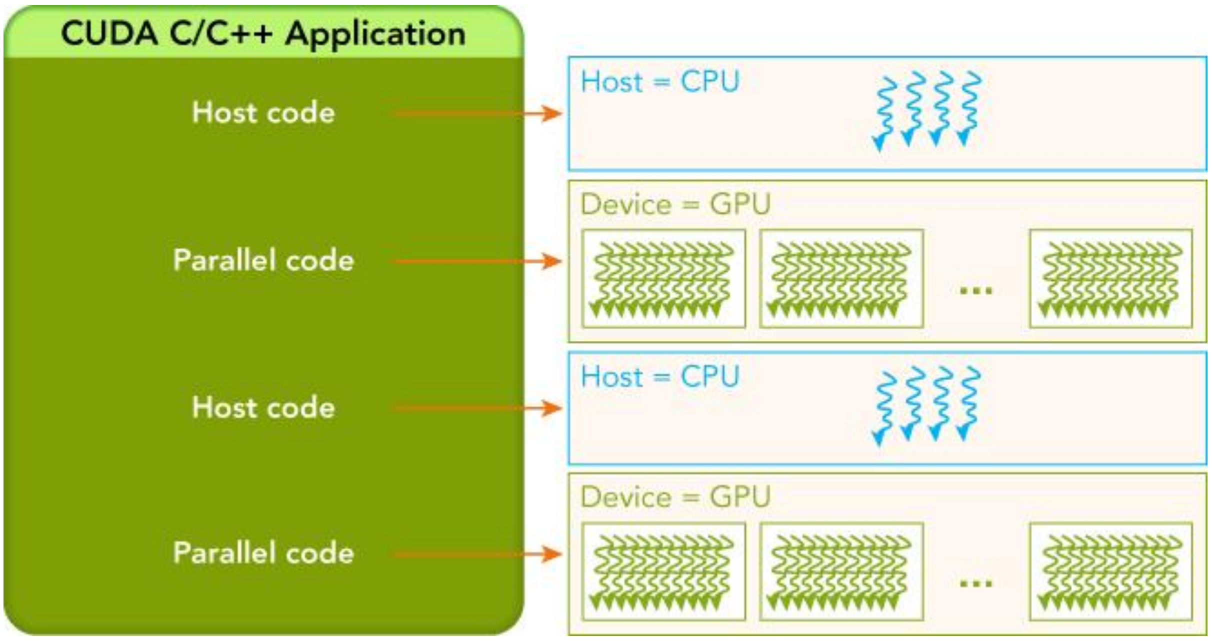

Figure 2: CUDA components

Figure two above shows a standard breakdown of how a program runs and passes control between the host and the device. Data is copied from the host to the device, the data then stored in the device memory is invoked and the data is then copied back on to the host memory. The device is made up of blocks, a group of blocks form a grid, and each block can have up to 1024 threads each (Filippone, n.d.-a). When the host has launched the kernel in the device, the execution moves to the device where many threads are generated that carry out the work (Chapter 2: CUDA Programming Model - Professional CUDA C Programming, n.d.). The figure below shows the organisation of the threads on the device.

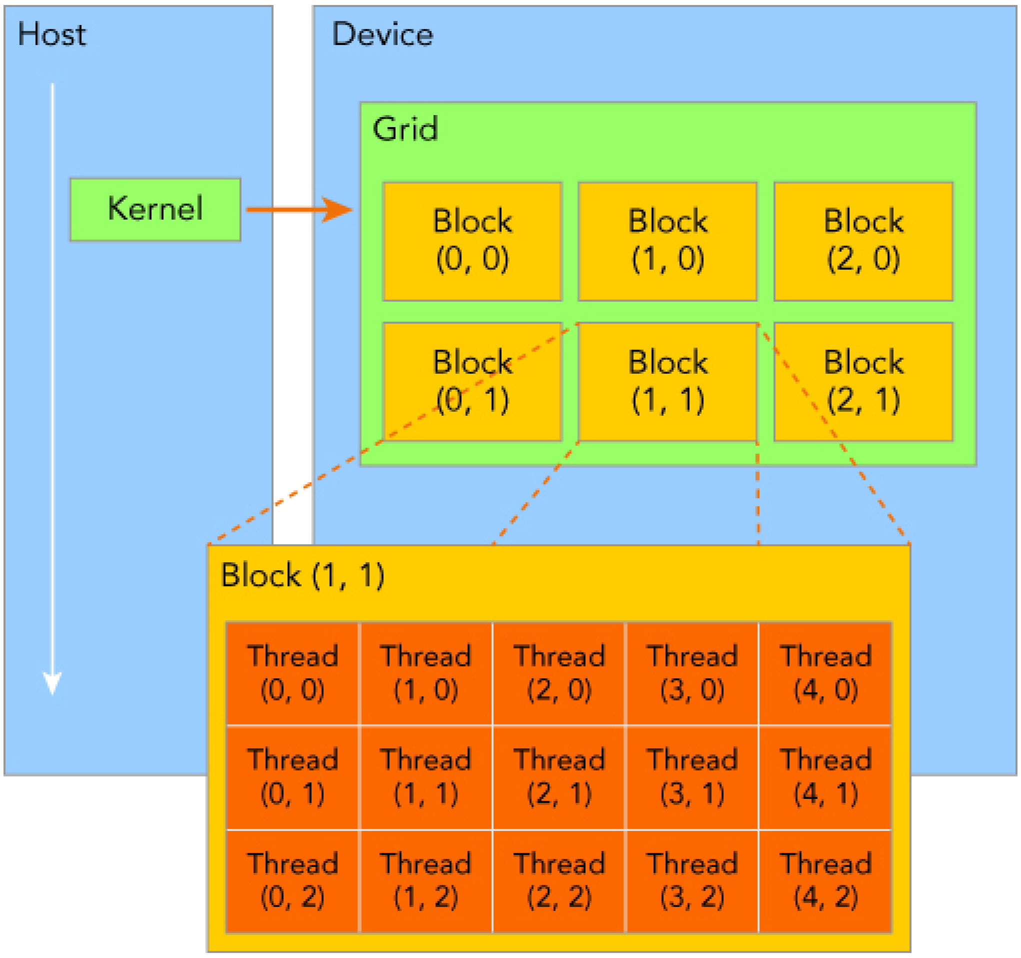

Figure 3: Threads and Block Organisation on the device (Chapter 2: CUDA Programming Model - Professional CUDA C Programming, *n.d.)*

5. Design Methodology

The section provides a breakdown of the design methodology used for the two programs using OpenMP and CUDA.

5.1. OpenMP

As described above, OpenMP is a set of directives that are added to a serial code to let the compiler parallelise the code. For the purposes of this work, where the problem statement is a GEMM, matrix-matrix multiplication the design of the program is as following. Initially, serial code will be written to ensure that it obtained correct results, the serial code is a straightforward algorithm where rows and columns loop are written alongside with two dimensional arrays. The C programming language stores matrices in row major order and that is the route that has been taken for the design of this algorithm. Furthermore, the matrices

have been declared using double point precision, over the years performance differences on the CPU between float and double precision has been reduced to the point now that it is almost negligible. Therefore, for better results double precision matrices have been used.
To parallelise the code, another function has been written that runs the code using OpenMP. The syntax of OpenMP require a #pragma to start and to use a for loop, the #pragma for directive will be used. In OpenMP, all variables are shared by default and which variable is shared and which variable is private needs to be declared explicitly as a clause. The three matrices will be shared across all the threads, as these will be required by all the threads when carrying out the multiplication. The variables used for the loops, i.e. I, j and k will be declared as private, as each thread will carry out the loop work individually, such as increasing the variable I by 1 after each iteration. The loops can be scheduled in OpenMP, there are a total of three options, static scheduling where set number of iterations are set for each thread in a round-robin fashion, Dynamic scheduling where a chunk of loop iteration is taken from a queue at run time and sent to the core and Guided scheduling where a large chunk is sent to the core and this gradually reduces until it reaches the chunk size defined in the code (*12 Introduction to OpenMP - Multicore DSP*, n.d.). For the purposes of this algorithm, Dynamic scheduling will be used where the provided chunk size will be of 5. Testing was carried out using static scheduling and dynamic scheduling, and it was noticed that the performance of dynamic scheduling with the chunk size 5 provided better results. Furthermore, a reduction operator will also be declared, the purpose of this is to ensure that the product of the matrices is carried out at the end of parallel region.

5.2. CUDA

As described above, CUDA works as an extension to the C programming language however, there are some syntax differences. Such as declaring dynamic memory on the host can be declared as “malloc” and the memory declared on the device must be declared as “cudamalloc” however, Nvidia have tried to keep CUDA as close to the C programming language as possible and in most cases, the syntax is similar to that of C.
For the purposes of this algorithm, the matrices have used the row-major order and they have been declared using the floating-point precision. It should be noted that floating point math is not associative and there will be some rounding off leading to somewhat different results, and even in the order that the operations are performed can lead to different results (*CUDA C BEST PRACTICES GUIDE Design Guide*, 2012).
To parallelise the code using CUDA, the tiling technique will be used. Using this technique makes the computation proceed by the size of the tiles (Filippone, n.d.-b). Matrix ax and ay use shared memory, the size of the matrix would be same as the size of the tile (in two dimensions). CUDA provides build in coordinate variables for the block and thread Ids, these are assigned to each thread during the CUDA runtime. These coordinates can then be used to assign data to different threads. The design purpose of using this method to see if it is possible to use one thread per row and one thread per column and this is how the row and column variables have been declared, using the coordinate variables provided by CUDA. A for loop is then called to initialise matrix A and B, following on from which the threads must be synched. Another loop is called to be able to carry out the addition and multiplication in a single instruction execution however, it should be noted that doing this operation together and separately will provide different results (*CUDA C BEST PRACTICES GUIDE Design Guide*, 2012). Following on from this operation the threads are again synchronised to provide correct results.
The CUDA methods are then called in the main function, where dynamic memory is declared on the device with “cudaMalloc” and matrix A and matrix B are copied from host to device using “cudaMemcpy”. To carry out the calculations on the GPU, the Grid dimension and the

Block dimension must be declared however, choosing the optimum block size is a difficult task and some good initial guesses include 16 x 16 or 32 x 8. In the case of this algorithm, the block size has been declared as 16 x 16.
The square matrix function is then called to carry out the product of two matrices inside the GPU with the Grid dimension and Block size provided, the operation will be carried out on the arrays declared dynamically on the device and then a device synchronisation. The results will be copied from the GPU to matrix C via the "cudaMemcpy" function again.
A simple matrix multiplication serial function will also be written which will be like the one described above for the OpenMP algorithm. The serial matrix and Cuda matrix results will be compared, and it will show the time it took execute, the Gflops each algorithm used and the difference between the results. Figure 4 below shows the GEMM implementation using the tile methodology (Filippone, n.d.-b).

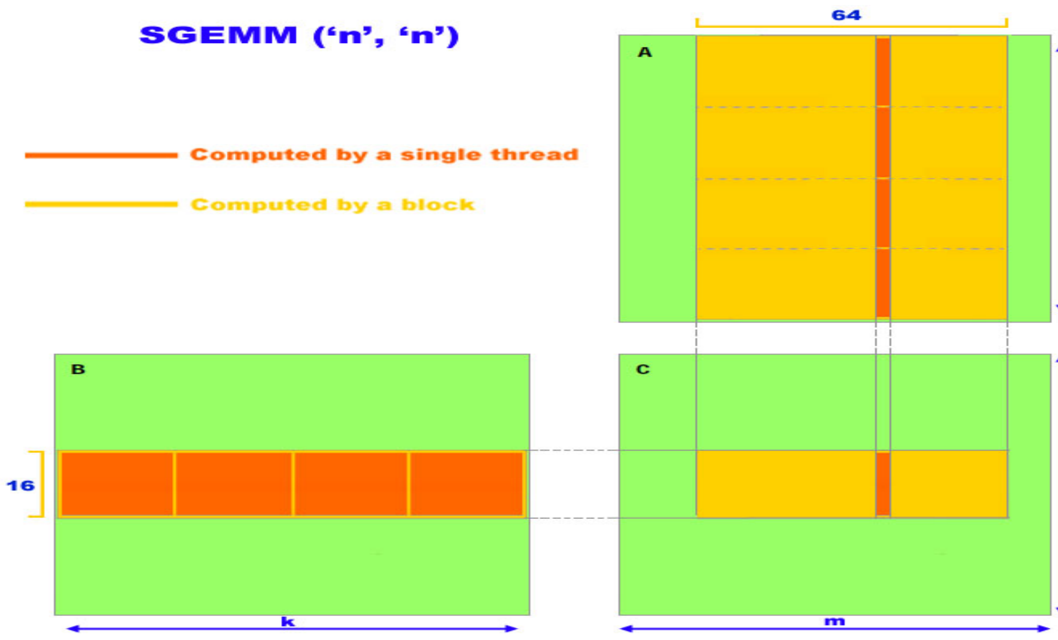

Figure 4: GEMM Implementation

6. Results

The solution to the given problem statement was carried out, and this included obtaining and comparing the results between various matrix sizes for both OpenMP and CUDA algorithms. The results are shown in the figures below and analysed to be able to determine how the performance changes depending on the size of the matrices. Both the OpenMP and CUDA versions were tested on Crescent and the results provided below are based on those tests.

6.1. OpenMP

OpenMP runs the program on the CPU, with each core having 2 threads. Running the code on Crescent shows that each node has 4 cores within Crescent and this equates to a total of 8 threads when running the tests. Graph 1 below shows the results of the tests run with various matrix sizes on Crescent.

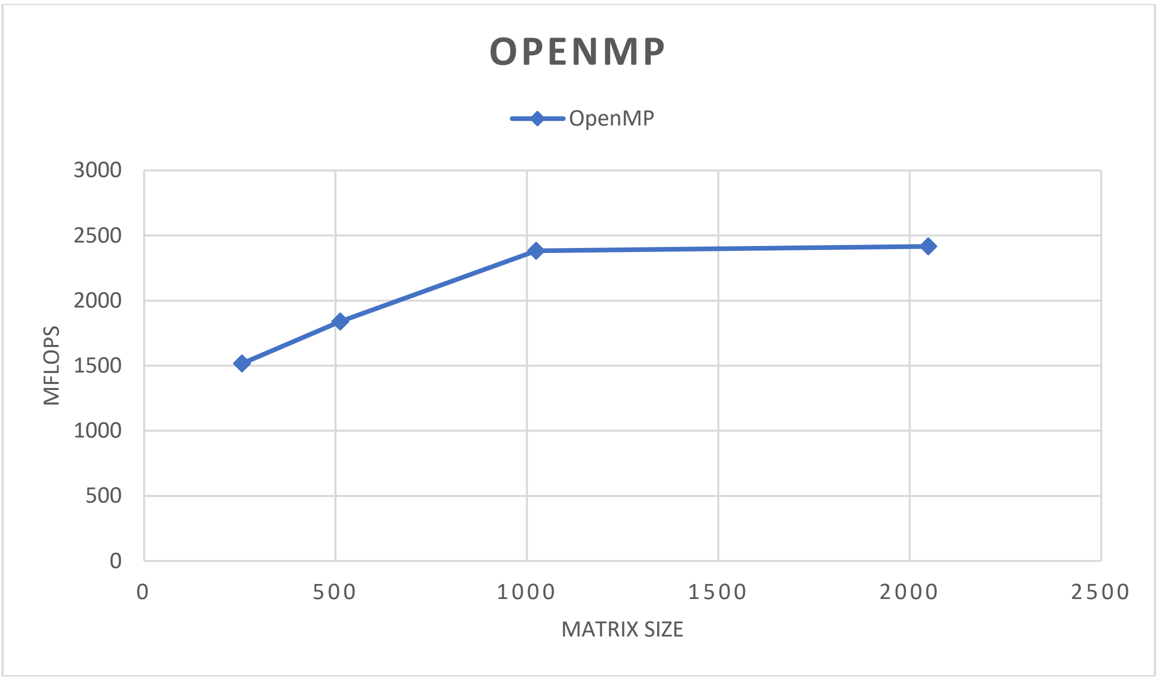

Figure 5: OpenMP performance

Graph 1 above shows the results obtained while carrying out the tests on Crescent, it shows how with the size of the matrices, the performance also increases. However, although OpenMP makes it easier to use systems that have shared memory, it is difficult to pinpoint where the performance has been lost. To complete the OpenMP algorithm for this matrix-matrix problem, care was taken to ensure that "ordered" schedule was not used, make the critical regions small where possible, parallel regions were avoided inside the loops and false sharing was avoided, these are some of the things that were taken into account when designing this algorithm (Wachsmann, 2007).

6.2. CUDA

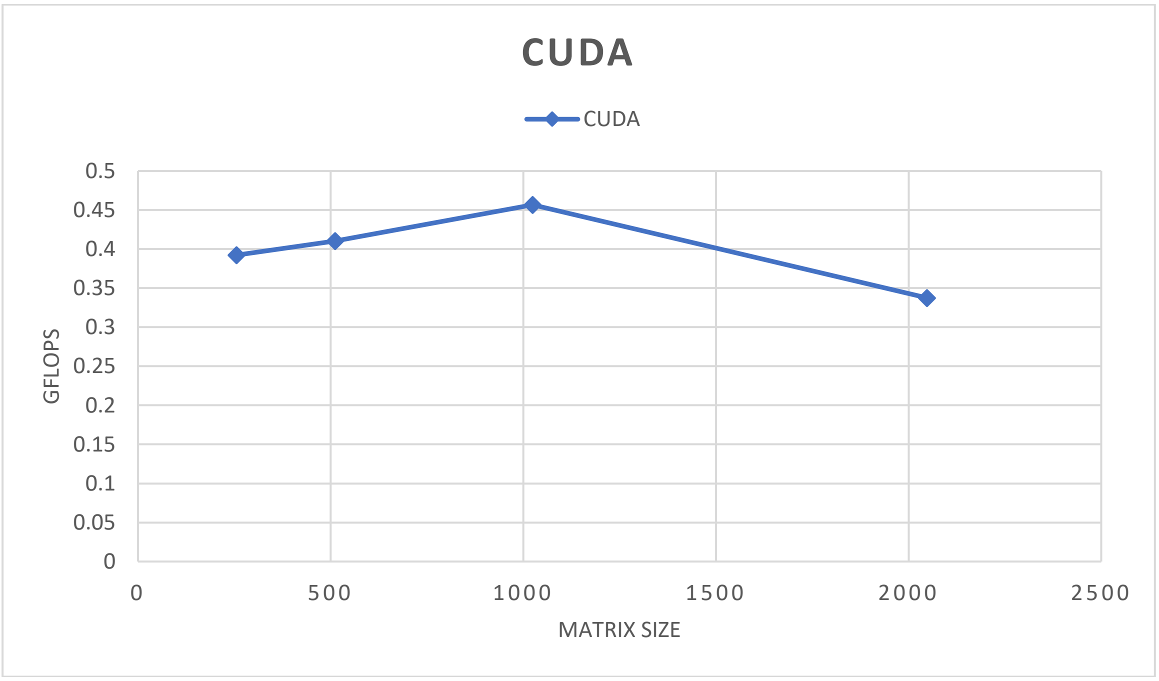

Figure 6: CUDA Performance

The initial design for the CUDA algorithm was based around the capabilities of the Tesla GPU that is available on Crescent. As there is no standard way of knowing which block size would be the optimal, the algorithm was tested with various block sizes and it was determined that the optimal size for this algorithm is 16 x 16. As can be seen from the graph above, that as the size of the matrices increase so does the performance however, after it reaches the size of 1024, the performance drops considerably. This is due to the reason that each thread block size 1024 and threads within the same block share the same memory however, if threads must communicate from between blocks, the communication is not as efficient as it is within the same block and that is the reason the performance drops (Filippone, n.d.-a).

7. Testing

In order to ensure that the performance seen above by the OpenMP and the CUDA algorithms perform correctly on other hardware and that they are portable, they were also tested on the local machine which is an Alienware Laptop with Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 2801 Mhz, 4 Core(s), 8 Logical Processor(s) and NVIDIA GeForce GTX 1060 Mobile 6GB GPU.

7.1. Parts to be tested

The types of tests that have been carried out are:

- Acceptance testing
- Integration testing
- Performance testing
- Correctness testing

7.1.1. Acceptance testing

Smoke testing will be carried out to set out the criteria of the tests. This test is carried out at a high level to determine if further testing is required at this stage, the following will be tested:

- The Kernels compile and run on local machine
- The Kernels compile and run on the University machine
- The kernels compile and run on Crescent

7.1.2. Integration testing

The serial algorithm and the parallel algorithms will be tested together an individually. Initially, the serial algorithm will be tested, then the parallel algorithm tested with various sizes the resulting matrices of both algorithms will be printed to ensure that the results match. However, it should be noted that in the case of CUDA kernel, it is not possible for the CPU (i.e. serial algorithm) and the GPU results to be exactly the same, as floating-point arithmetic is not associative (*CUDA C BEST PRACTICES GUIDE Design Guide*, 2012).

7.1.3. Performance testing

The code will be tested on Crescent and the results are recorded and analysed above in the results section.

7.1.4. Correctness testing

In order to ensure that the results provided by both algorithm is correct, the results were compared to that of the serial algorithm.

In the case of the OpenMP algorithm, the checkResults function compares the results between the OpenMP algorithm and the serial algorithm to within the tolerance of 0.1. If the results do not match then an error is thrown to advise the user of incorrect results. It is expected that the results would be the same for the OpenMP and the serial algorithm as both of these algorithm run on the CPU, i.e using same resources, this of course is only true if both the algorithms are correct.

To check the correctness of the CUDA algorithm, the result for the Serial and the CUDA algorithms are compared however, is it expected that the results will not be exactly the same as the serial algorithm runs on the CPU and the CUDA algorithm runs on the GPU and differences are expected due to rounding off.

8. Discussion

Following on from the work carried out on this project, it should be noted that the CUDA algorithm does not perform to the full capability of what is available on Crescent. The purpose of this project was to be able use the OpenMP and CUDA algorithms to parallelise the multiplication of two square matrices which has been achieved however, due to time constraints it has not been possible to improve the performance to make the full use of available power on Crescent.
Carrying out the work on this project has led to a good understanding of how to parallelise programs that can make use of all the available threads on the CPU and the GPGPU. When running the code, the multiplication provides correct results when it has been parallelised by both OpenMP and CUDA.
The future work to be carried out on this project is to be able to improve the performance of the parallelised algorithms. The CUDA algorithm can perform much better that it has due to the capability available on the Crescent high-performance computer. Although both the algorithms provide the correct results, the purpose of parallelising the work is to have better performance and that is something that will need to be improved.

9. Conclusion

A detailed amount of work has been carried out for the completion of this project, including the design of the kernels, and writing the source code. Although the performance of the algorithms is not where it can be based on the capabilities of Crescent, the product of the matrices provides the correct results, even though the CUDA GPU results differ from the CPU results due to rounding off differences.

The amount of work carried out to design, write the source and test is quite substantial and the algorithms did have to be tested individually with varying matrix sizes. The results were then compared and analysed with a breakdown provided with the differences in performance.

10. Bibliography

- *12 Introduction to OpenMP - Multicore DSP*. (n.d.). Retrieved April 8, 2021, from https://learning.oreilly.com/library/view/multicore-dsp/9781119003823/c12.xhtml
- *Chapter 2: CUDA Programming Model - Professional CUDA C Programming*. (n.d.). Retrieved April 8, 2021, from https://learning.oreilly.com/library/view/professional-cuda-c/9781118739310/c02.xhtml
- *CUDA C BEST PRACTICES GUIDE Design Guide*. (2012). www.nvidia.com
- Filippone, S. (n.d.-a). *GPGPU Programming Fundamentals — 1*.
- Filippone, S. (n.d.-b). *GPGPU Programming Fundamentals — 3*.
- Huang, J., & van de Geijn, R. A. (2016). *BLISlab: A Sandbox for Optimizing GEMM*. http://arxiv.org/abs/1609.00076
- Kunkel, J. M., Conference, I., Performance, I. S. C. H., & Hutchison, D. (2016). Disc_2016. In *Technological Forecasting and Social Change* (Vol. 110). https://doi.org/10.1016/s0040-1625(16)30193-7
- Wachsmann, A. (2007). Shared Memory Programming with OpenMP LECTURE 6: TASKS. *Spring*, 1–21.

11. Appendix A

11.1 OpenMP

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#include <stdbool.h>

#define CHUNK 5

void MatrixMultiply(int matrix_size, double A[matrix_size][matrix_size], double
transposeB[matrix_size][matrix_size], double C[matrix_size][matrix_size]);
void transposed(int matrix_size, double transpose[matrix_size][matrix_size], double
B[matrix_size][matrix_size]);
void print_matrix(int matrix_size, double A[matrix_size][matrix_size], double
B[matrix_size][matrix_size], double C[matrix_size][matrix_size]);
void MatrixMultiplyOmp(int matrix_size, double A[matrix_size][matrix_size], double
transposeB[matrix_size][matrix_size], double C[matrix_size][matrix_size]);
void checkResults(int matrix_size, double C[matrix_size][matrix_size], double
SerialC[matrix_size][matrix_size]);

int main(int argc, char** argv)
{
   int i, j, matrix_size;
   //To ensure that the right amount of arguments are provided when running the program
   if(argc < 2)
      fprintf(stderr, "Usage: %s  row/col, it is a square matrix\n",argv[0]);
   matrix_size = atoi(argv[1]);

   int tid, nthreads;

   //Dynamic memory allocation for the matrices
   double (*A)[matrix_size] = malloc(sizeof(double)*matrix_size*matrix_size);
   double (*B)[matrix_size] = malloc(sizeof(double)*matrix_size*matrix_size);
   double (*C)[matrix_size] = malloc(sizeof(double)*matrix_size*matrix_size);
   double (*transposeB)[matrix_size] = malloc(sizeof(double)*matrix_size*matrix_size);
   double (*SerialC)[matrix_size] = malloc(sizeof(double)*matrix_size*matrix_size);

   //Randomising the A and B matrices
   srand(time(0));
   for(i = 0; i < matrix_size; i++){
      for(j = 0; j < matrix_size; j++) {
         A[i][j] = 100.0f * ((double) rand()) / RAND_MAX;;
         B[i][j] = 100.0f * ((double) rand()) / RAND_MAX;
      }
   }

// Only left here if the user wants to print the randomised matrices.
//   printf("Matrix A: \n");
//   for(i = 0; i < matrix_size; i++) {
//       for (j = 0; j < matrix_size; j++) {
//           printf("%2.2f ", A[i][j]);
//       }
//       printf("\n");
//   }
//   printf("Matrix B: \n");
//   for(i = 0; i < matrix_size; i++) {
//       for (j = 0; j < matrix_size; j++) {
//           printf("%2.2f ", B[i][j]);
//       }
//       printf("\n");
//   }

   //Transposing matrix B
   transposed(matrix_size, transposeB, B);
   //To start measuring the time of the parallel execution
   double startTime = omp_get_wtime();
   //To carry out the multiplication of A * B and save it into matrix C using OpenMP.
   MatrixMultiplyOmp(matrix_size, A, transposeB, C);
   //To end measuring the time of the parallel execution
   double endTime = omp_get_wtime();
   //Only left here if the user wants to print the resultant matrix C.
//   print_matrix(matrix_size, A, transposeB, C);
   //To start measuring the time of the serial execution
   double startTimeSer = omp_get_wtime();
   //To carry out the multiplication of A * B and save it into matrix C using serial algorithm.
   MatrixMultiply(matrix_size, A, transposeB, SerialC);
   //To end measuring the time of the serial execution
   double endTimeSer = omp_get_wtime();
   //Only left here if the user wants to print the resultant matrix C.
//   print_matrix(matrix_size, A, transposeB, SerialC);

   // To measure the total execution time of the parallel algorithm.
   double totTime = endTime - startTime;
   // To measure the total execution time of the serial algorithm.
   double totTimeser = endTimeSer - startTimeSer;

   //To compare the results of the parallel and serial algorithm, ensuring they match.
   checkResults(matrix_size, C, SerialC);
   printf("Parallel time = %f and Serial time = %f\n", totTime, totTimeser);

   //To measure the performance of the two algorithms, in Mega flops.
   double mflops = (2.0e-6)*matrix_size*matrix_size*matrix_size/totTime;
#pragma omp parallel
  {
```

```
#pragma omp master
    {
      fprintf(stdout,"Multiplying matrices of size %d x %d (%d) (block_42) with %d threads:
time %lf  MFLOPS %lf \n",
                 matrix_size,matrix_size,matrix_size,omp_get_num_threads(),totTime,mflops);
    }
  }
  // To free the allocated memory.
    free(A);
    free(B);
    free(C);
    free(transposeB);
    free(SerialC);
    return 0;

}
//Serial algorithm to carry out the multiplication of matrix A and B and saves it matrix C.
void MatrixMultiply(int matrix_size, double A[matrix_size][matrix_size], double
transposeB[matrix_size][matrix_size], double C[matrix_size][matrix_size])
{
    int i,j,k;
    for (i = 0; i < matrix_size; i++) {
        for (j = 0; j < matrix_size; j++) {
            C[i][j] = 0;
            for (k = 0; k < matrix_size; k++) {
                C[i][j] = C[i][j] + A[i][k] * transposeB[j][k];
            }
        }
    }
}

//Transposes a matrix.
void transposed(int matrix_size, double transpose[matrix_size][matrix_size], double
B[matrix_size][matrix_size])
{
    int j, k;
    for (j = 0; j < matrix_size; j++) {
        for (k = 0; k < matrix_size; k++) {
            transpose[k][j] = B[j][k];
        }
    }
}

//To print a matrix.
void print_matrix(int matrix_size, double A[matrix_size][matrix_size], double
B[matrix_size][matrix_size], double C[matrix_size][matrix_size])
{
    int i, j, k;
    for(i = 0; i < matrix_size; i++)
    {
        for(j = 0; j < matrix_size; j++)
        {
            printf("%2.2f   ", C[i][j]);
        }
        printf("\n");
    }
}
//Parallel algorithm to carry out the multiplication of matrix A and B and saves it matrix C,
uses OpenMP.
void MatrixMultiplyOmp(int matrix_size, double A[matrix_size][matrix_size], double
transposeB[matrix_size][matrix_size], double C[matrix_size][matrix_size])
{
    int i,j,k;
    double dot = 0.0;

#pragma omp parallel for shared(A, transposeB, C) private(i, j, k) schedule(dynamic,
CHUNK) reduction(+: dot)
    for (i = 0; i < matrix_size; i++) {
        for (j = 0; j < matrix_size; j++) {
            C[i][j] = 0;
            for (k = 0; k < matrix_size; k++) {
                dot += (A[i][k] * transposeB[j][k]);
                C[i][j] = dot;
            }
        }
    }
}

//To check the results of two resultant matrices match.
void checkResults(int matrix_size, double C[matrix_size][matrix_size], double
SerialC[matrix_size][matrix_size])
{
    int i, j;
    bool answer = true;
    for(i = 0; i < matrix_size; i++)
    {
        for(j = 0; j < matrix_size; j++)
        {
            if ((SerialC[i][j] - C[i][j]) > 0.1)
                answer = false;
        }
    }
    if (answer) {
        printf("The values for Serial and Parallel matrices match!\n");
    } else
        printf("The values for Serial and Parallel matrices do not match!\n");
}
```

CMakeLists

```
cmake_minimum_required(VERSION 3.17)
project(oomp C)

set(CMAKE_C_STANDARD 99)

add_executable(oomp main.c)
```

11.2 CUDA

cuda.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include <iostream>
#include <cuda_runtime.h>  // For CUDA runtime API
#include "helper_cuda.h"  // For checkCudaError macro
#include "helper_timer.h"  // For CUDA SDK timers

#define BLOCK_SIZE 16
const dim3 BLOCK_DIM(BLOCK_SIZE, BLOCK_SIZE);

//Serial algorithm to carry out the multiplication of matrix A and B and saves it matrix C.
void MatrixMultiply(int matrix_size, float *A, float *B, float *C)
  {
     int i,j,k;
     // printf("Resultant Matrix:\n");
     for (i = 0; i < matrix_size; i++) {
        for (j = 0; j < matrix_size; j++) {
           float t = 0.0f;
           for (k = 0; k < matrix_size; k++) {
              int idxA = i * matrix_size + k;
              int idxB = k * matrix_size + j;
              t += A[idxA] * B[idxB];
           }
           C[i] = t;
           // printf("%f   ", C[i]);
        }
        // printf("\n");
     }
  }

//Transposes a matrix.
void transposed(int matrix_size, float *transpose, float *B)
{
     int j, k;
     for (j = 0; j < matrix_size; j++) {
            for (k = 0; k < matrix_size; k++) {
               transpose[k * matrix_size + j] = B[j * matrix_size + k];
            }
         }
      }

//To print a matrix.
void print_matrix(int matrix_size, float *A, float *B, float *C)
   {
      int i, j;
      printf("The resultant Matrix is:\n");
      for(i = 0; i < matrix_size; i++)
      {
         for(j = 0; j < matrix_size; j++)
         {
           int idx = i * matrix_size + j;
           printf("%f   ", C[idx]);
         }
         printf("\n");
      }
   }

//Parallel algorithm to carry out the multiplication of matrix A and B and saves it matrix C,
uses CUDA.
__global__ void square_matrix_mult_gpu (float *A, float *B, float *C, int matrix_size)
   {
     //Declaring shared memory space for ax and ay.
     __shared__ float ax[BLOCK_SIZE][BLOCK_SIZE];
     __shared__ float ay[BLOCK_SIZE][BLOCK_SIZE];

     int row = blockIdx.y*BLOCK_SIZE + threadIdx.y;
     int col = blockIdx.x*BLOCK_SIZE + threadIdx.x;
     float t = 0.0f;
     int idx;

     for(int i =0; i<gridDim.x; i++){
       idx = row * matrix_size + i * BLOCK_SIZE + threadIdx.x;

       ax[threadIdx.y][threadIdx.x] = A[idx];
       ay[threadIdx.y][threadIdx.x] = B[idx];

        __syncthreads();

       for(int k = 0; k <BLOCK_SIZE; k++)
       {
         t += ax[threadIdx.y][k] * ay[k][threadIdx.x];
       }
         __syncthreads();
     }
     if(row < matrix_size && col < matrix_size){
       C[row * matrix_size + col] = t;
     }
   }

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

    int matrix_size;
    if(argc < 2)
        fprintf(stderr, "Usage: %s  matrix_size, it is a square matrix\n",argv[0]);
    matrix_size = atoi(argv[1]);

    float* A = new float[matrix_size * matrix_size];
    float* B = new float[matrix_size * matrix_size];
    float* C = new float[matrix_size * matrix_size];
    float* transposeA = new float[matrix_size * matrix_size];
    float* transposeB = new float[matrix_size * matrix_size];
    float* SerialC = new float[matrix_size * matrix_size];

    srand(time(0));
    // printf("Matrix A:\n");
    for (int row = 0; row < matrix_size; ++row) {
      for (int col = 0; col < matrix_size; ++col) {
        int idx = row * matrix_size + col;
        A[idx] = 100.0f * static_cast<float>(rand()) / RAND_MAX;
        // printf("%2.2f   ", A[idx]);
      }
      // printf("\n");
    C[row] = 0.0f;
}
    // printf("Matrix B:\n");
    for (int row = 0; row < matrix_size; ++row) {
    for (int col = 0; col < matrix_size; ++col) {
      int idx = row * matrix_size + col;
      B[idx] = 100.0f * static_cast<float>(rand()) / RAND_MAX;
      // printf("%2.2f   ", B[idx]);
  }
  // printf("\n");
}

    transposed(matrix_size, transposeB, B);
    transposed(matrix_size, transposeA, A);
    float *d_a, *d_x, *d_y;

     checkCudaErrors(cudaMalloc((void**) &d_a, matrix_size * matrix_size * sizeof(float)));
     checkCudaErrors(cudaMalloc((void**) &d_x, matrix_size * matrix_size * sizeof(float)));
     checkCudaErrors(cudaMalloc((void**) &d_y, matrix_size * matrix_size * sizeof(float)));

     checkCudaErrors(cudaMemcpy(d_a, transposeA, matrix_size * matrix_size *
sizeof(float), cudaMemcpyHostToDevice));
```

```
    checkCudaErrors(cudaMemcpy(d_x, transposeB,  matrix_size * matrix_size *
sizeof(float), cudaMemcpyHostToDevice));

    // ----------------------- Calculations on the CPU ------------------------ //
    float flopcnt=2.e-6*matrix_size*matrix_size;

    // Create the CUDA SDK timer.
    StopWatchInterface* timer = 0;
    sdkCreateTimer(&timer);

    timer->start();
    MatrixMultiply(matrix_size, transposeA, transposeB, SerialC);
    // print_matrix(matrix_size, A, B, SerialC);

    timer->stop();
    float cpuflops=flopcnt/ timer->getTime();
    printf("CPU time = %f ms, GFLOPS = %f\n", timer->getTime(), cpuflops);

    // ----------------------- Calculations on the GPU ------------------------ //

  // Calculate the dimension of the grid of blocks.
  unsigned int grid_matrix = (matrix_size + BLOCK_SIZE - 1) / BLOCK_SIZE;
  const dim3 GRID_DIM(grid_matrix, grid_matrix);

  timer->reset();
  timer->start();
  //Carry out the multiplication on GPU.
  square_matrix_mult_gpu<<<GRID_DIM, BLOCK_DIM >>>(d_a, d_x, d_y, matrix_size);
  checkCudaErrors(cudaDeviceSynchronize());

  timer->stop();
  float gpuflops=flopcnt/ timer->getTime();
  printf("GPU time = %f ms, GFLOPS = %f\n", timer->getTime(), gpuflops);

  // Download the resulting vector d_y from the device and store it in C.
  checkCudaErrors(cudaMemcpy(C, d_y, (matrix_size * matrix_size) *
sizeof(float),cudaMemcpyDeviceToHost));
  // for(int i = 0; i < matrix_size; i++){
  //   for(int j = 0; j < matrix_size; j++) {
  //     printf("%f   ", C[i * matrix_size + j]);
  //   }
  //   printf("\n");
  // }

  // Now let's check if the results are the same.
  float reldiff = 0.0f;
  float diff = 0.0f;

  for (int row = 0; row < matrix_size; row++) {
    float maxabs = max(abs(SerialC[row]), abs(C[row]));
    if (maxabs == 0.0f) maxabs = 1.0f;
    reldiff = max(reldiff, abs(SerialC[row] - C[row])/maxabs);
    diff = max(diff, abs(SerialC[row] - C[row]));
  }
  printf("Max diff = %2.2f, Max rel diff = %2.2f\n", diff, reldiff);

// ------------------------------- Cleaning up ------------------------------ //

  delete timer;

  checkCudaErrors(cudaFree(d_a));
  checkCudaErrors(cudaFree(d_x));
  checkCudaErrors(cudaFree(d_y));

  delete[] A;
  delete[] B;
  delete[] C;
  delete[] SerialC;
  delete[] transposeA;
  delete[] transposeB;
  return 0;
}
```