

Lab-based Matrix Multiplication

CMP202 2023/24 Term 2

Mini-Project: Mini-Project 2, GPU

Jack Laundon

2202274

1	Introduction	1
2	Application Overview.....	1
3	Parallelization Strategy and Implementation	1
4	Performance Evaluation	2
5	Conclusion.....	3

1 Introduction

The task chosen for this project was a parallel version of non-square matrix multiplication. This is from the lab-based ideas sheet provided. This program contains six functions to carry out matrix multiplication, using the following methods:

- Buffers and Accessors
- Implicit USM Memory
- Explicit USM Memory
- Tiled Matrix Multiplication
- Subgroup Matrix Multiplication
- CPU Buffers and Accessors.

The application runs each of these and compares the performance (the time taken in nanoseconds).

2 Application Overview

As per the instructions of Task 1 and Task 2, each function was modified to accept two non-square matrices, $M \times N$ and $N \times P$, and multiply them together to produce a third matrix, $M \times P$. Following the instructions of Task 3, a function was added that utilises subgroups. This function closely follows the tiled matrix multiplication function because this was found to be the fastest function, but the subgroup function uses the subgroup operation “broadcast” and a subgroup barrier. The program includes the following features:

- Kernel Execution
 - The program contains functions that execute on the kernel to perform matrix multiplication. Task decomposition is shown in the tiled and subgroup functions when the “tiles” are used to distribute parts of the task. This is also shown whenever a parallel for loop is launched, as the `ND_Range` kernels distribute parts of the task across work groups.
- Resource Management
 - Resources are shared safely through the use of barriers and subgroup barriers to synchronise all work items to ensure all tasks are completed before moving on, the use of allocating local memory to a tile to share data between work items in a work group, and the “broadcast” operation to share data between work items in a sub-group.
- Performance Optimisation
 - Optimisation strategies are used including USM, local memory in tiles, and subgroups
- Analysis
 - The CPU selector is used to compare the GPU program to CPU. This was tested across multiple data sizes and the speedups were calculated.
- Vary thread count
 - The tile size can be changed which changes the size of the work groups, changing the number of work items employed on each work group

3 Parallelization Strategy and Implementation

Work items are launched using the parallel for loops in each function, and are managed in the following ways:

- Item barriers
 - When item barriers are used, this provides a point of synchronisation to ensure that all work items have completed their tasks before continuing with the program. Specifically, the synchronisation should take place in the local memory, as signified by “`access::fence_space::local_space`”

- Subgroup barriers
 - Subgroup barriers provide another point of synchronisation to ensure all threads have completed their tasks before continuing with the program. For these barriers, they synchronise all work items within the sub-group, as opposed to local memory like item barriers.
- Kernel Launches
 - When a kernel is launched, the task is split across different work groups to be executed by multiple work items
- Tile size
 - The tile size dictates the size of the work group to be used in the tiled matrix multiplication function, which in turns dictates the number of work items in each work group
- Wait
 - More generally, the “wait()” function is used to provide another point of synchronisation by making sure that all tasks submitted to the queue are complete before resuming with the program.

The program uses the following implementation specifics:

- Subgroups
 - Used in a similar fashion to tiled matrix multiplication
 - Subgroup barriers used to synchronise all items in the subgroup
 - Broadcast used to distribute tileA in the subgroup function between work items in the subgroup
- Item barriers
 - Used to synchronise items in the local memory
- Task decomposition
 - Demonstrated across the program to distribute tasks efficiently through sub groups, tiles, and when nd_range kernels are launched
- Resource Sharing
 - Work items access the same data at once when the data is loaded into local memory, shown through the tiled function and partially in the subgroup function.
 - The data is distributed between work items in the sub-group using the “broadcast” function

4 Performance Evaluation

The dimensions used to test this program ranged from $M = 3000$, $N = 2500$, and $P = 2000$ to $M = 16000$, $N = 14000$, $P = 12000$. This resulted in matrices ranging from a size of six million to 192 million. See Figure 1 for a graph of the performance of each function, and Figure 2 for a graph of the speedups obtained by each function compared to the CPU buffers and accessors function.

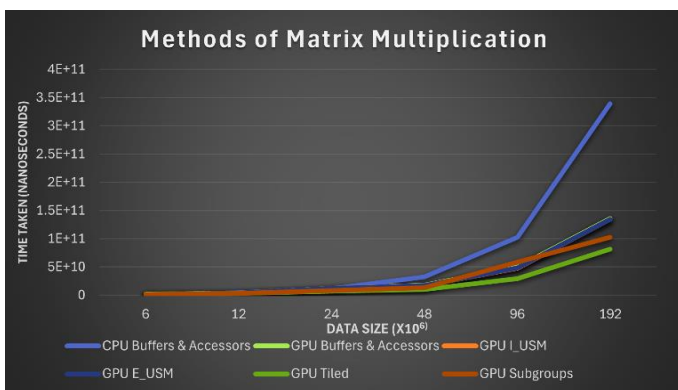


Figure 1 - Time taken

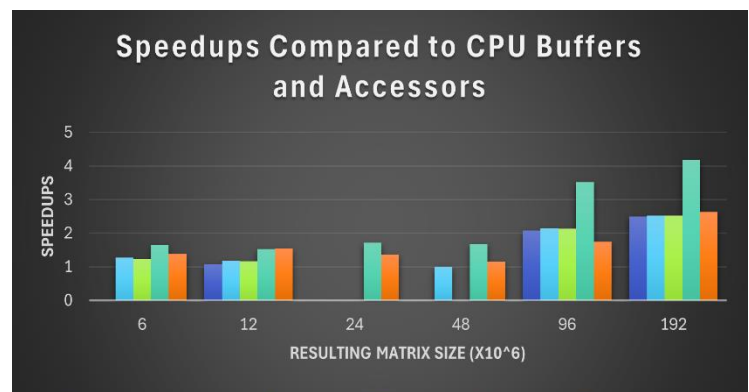


Figure 2 - Speedups

As shown by the graphs, the tiled method is the fastest, closely followed by subgroups, and the CPU function is far slower than all other functions, clearly highlighting the benefits of parallelism. Until a data size of 24×10^6 (24 million), each function takes a similar amount of time. This is due to the CPU having fewer cores than the GPU - when the data size becomes large, the GPU can efficiently use a substantially greater number of cores than the CPU.

Although the efficiency of the GPU functions compared to the CPU function is clear, it could possibly be made more efficient. Unlike the tiled function, the subgroup function does not use tile A and tile B, just tile A. This is because accessor B is used instead to utilise global memory. The reason for this is, although using global memory is not as efficient as using local memory, incorrect results were obtained when using the broadcast operation with local memory. The matrix was not multiplied correctly and thus the function was not running correctly. However, the correct results were obtained using global memory. If this program was repeated, a plausible method of increasing efficiency would be to use local memory if the correct result could be obtained. This was a challenge when developing the program and was addressed as stated by using global memory instead of local memory. This is not the most efficient way however correct results are obtained when using global memory. The performance could also be improved if run on a system that has a bigger maximum work size, as this means a bigger tile size can be used, meaning more work items can be running at once to execute tasks in parallel. In this program, a tile size of 16 was used as the maximum work size was 256, and 16×16 comes to 256.

Another challenge faced when developing this program was related to the tiled and subgroup functions. At apparently random points, the program ran but output a wall of error messages when it got the tiled function and then stopped prematurely. After careful examination, it found this was related to the tile size. If the tile size was bigger than the dimensions of the matrix, an error occurred. The tile size must be adjusted accordingly depending on the matrix dimensions. That being said, it is unlikely that the dimensions will be smaller than the tile size in a real-world scenario, it is more likely that this would occur during a testing phase. During development of the program, stack overflow errors were encountered. This was addressed by defining the matrices causing the issue as a "new float". To avoid memory leaks the matrices were freed or deleted after use.

5 Conclusion

In conclusion, this program effectively utilises the GPU of a computer to demonstrate the benefits of parallelism via matrix multiplication. The program uses buffers and accessors, explicit and implicit USM, tiled matrix multiplication using local memory, subgroup matrix multiplication in a similar fashion to tiled matrix multiplication, and buffers and accessors matrix multiplication running on the CPU to demonstrate the difference in efficiency. If the subgroup function was able to use local memory while obtaining the correct results, this function could run even faster, and if run on a system with a greater maximum work size, the efficiency of that function and the tiled function could be increased.