***Note: Part 1 was done in OPENCL and Part 2/3 was done in CUDA***

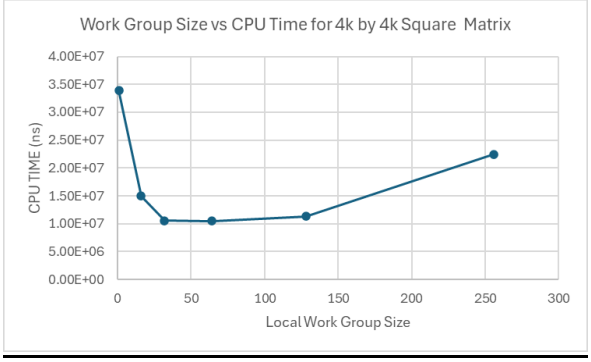
# Part 1: Matrix-Matrix Multiplication on GPU

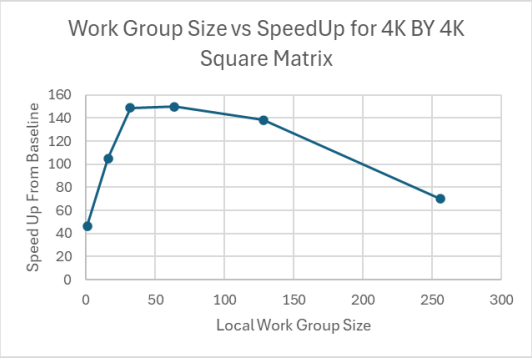
## Implementation 1: Single Row Based Decomposition ( Of Matrix A)

In single-row decomposition for matrix multiplication, matrix A is decomposed row-wise, with each work-item responsible for computing one row of the resultant matrix C. The global ID of each work-item corresponds to a specific row of A. This ensures that the work-item calculates all elements in the corresponding row of C by multiplying the entire row of A with all the columns in B.

Within the code, the outer loop than iterates over the shared dimensions of the matrix(K) and the inner loop iterates over the columns of C (N). Each row of C (C[I * N + j]) is added to the resultant matrix. Since the decomposition is row-based, a 1D workgroup is sufficient. Each thread in a work-group processes one row of C. The local work group size will affect how many threads/work items execute together in a group. Threads in the same work group can share local memory and synchronize with each other.

Within the code, the CLGETINFO function returned the maximum work group size as 256 in the cluster. Thus, this is the maximum number of work items that can be grouped into a single work group. This is what the upper limit I set my work group size sweeps too. With work group sizes past 256, the resultant matrix is not correct.

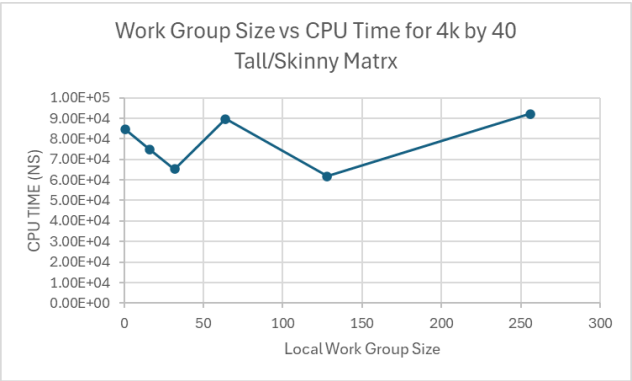


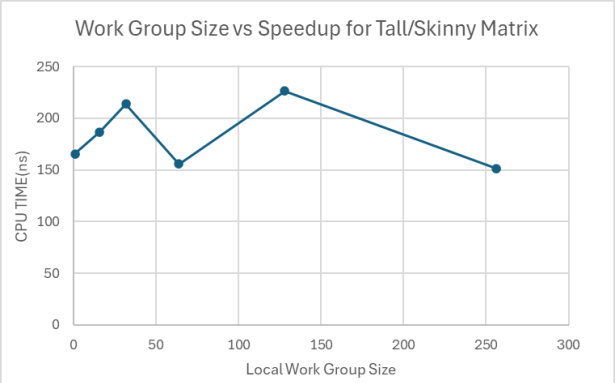




## Square Matrix Analysis:

At very small local work group sizes (up to around 32), the CPU time decreases as the work group size increases. This is due to the improved workload distribution and lower thread synchronization overhead, as more threads are available to share the computational load efficiently. Around a local work group size of **64**, the CPU time reaches its minimum value. At this point, the work group size manages trade-offs effectively, ensuring maximum utilization of hardware resources without excessive contention for these resources (register, shared memory). However, beyond a work group size of 64, the CPU time begins to increase due to increased contention for shared resources within the local memory.

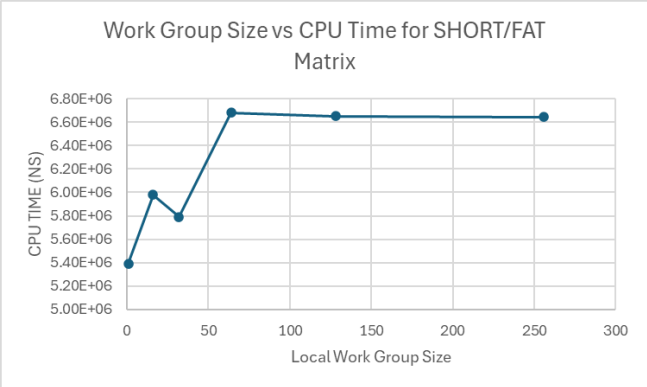
## Tall + Skinny Matrix Analysis

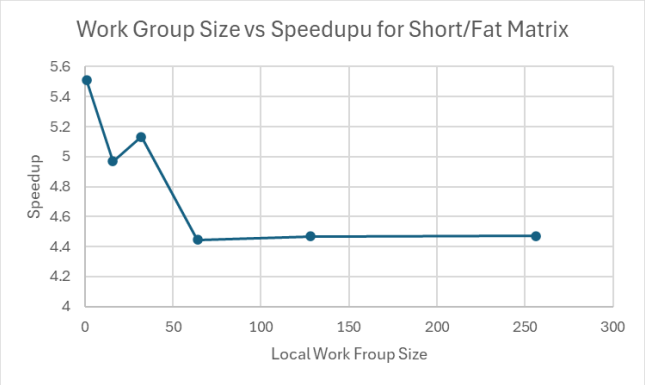


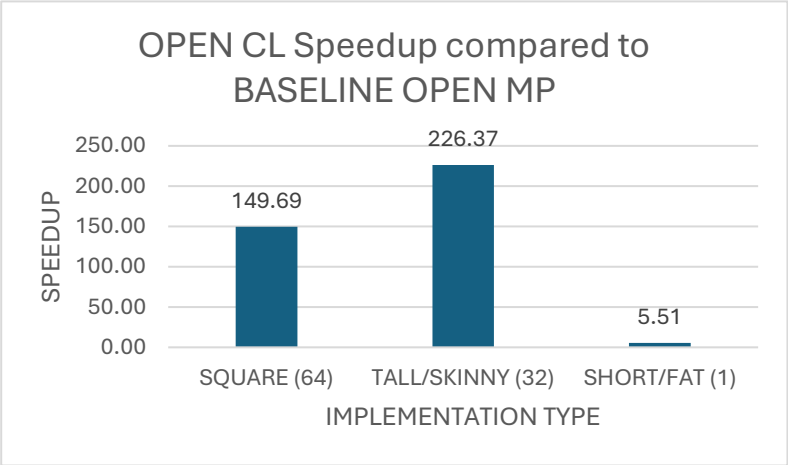

For a tall and skinny matrix, the performance pattern observed with a square matrix cannot be observed. This is because in a tall and skinny matrix, the number of rows is significantly larger than the number of columns. As a result, while each thread processes an entire row of C, each row involves processing far fewer elements compared to the square matrix. This leads to less computational work per thread, and many threads within a work group may remain idle or underutilized, wasting GPU resources. Thus, the workload is too small to fully leverage the GPU's parallelism. This inefficiency explains why, in general, the best results for tall and skinny matrices are achieved with relatively small work group sizes, such as around 32, where fewer threads can balance the limited workload more effectively.

## Short + Fat Matrix Analysis



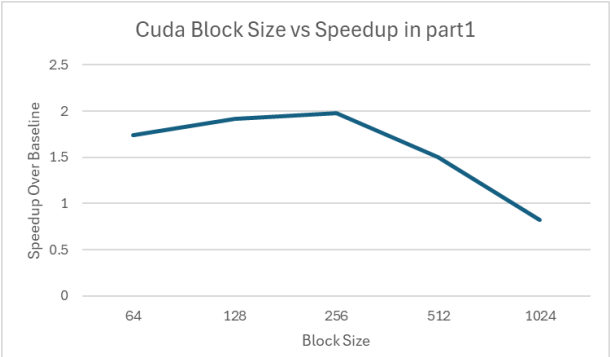

For the short/fat matrix, the best local work group size is the smallest (1). This is because, in a short/fat matrix, the number of rows is much smaller than the number of columns. With the current decomposition method, where one thread is assigned per row of C, only a small number of threads are active in each work group. The remaining threads remain idle, leading to significant underutilization of GPU resources. As the work group size increases, CPU time increases due to synchronization and shared memory overhead, which adds to the execution time without improving performance. Eventually, a plateau is reached as the work group size continues to grow. This happens because the active threads (one per row) perform all the necessary computations, while the additional threads remain idle.



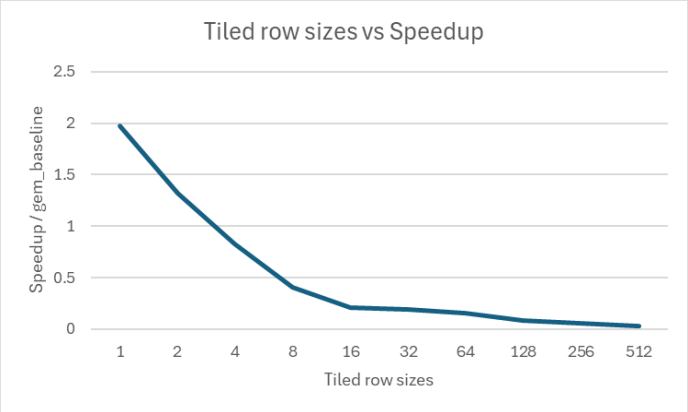

It can be observed that different matrix types achieve varying speedup values. The local work group size was optimized for each matrix type to minimize execution time. The best performance is achieved with the tall and skinny matrix, as this matrix type benefits significantly from the row-based decomposition. With many rows, there are a large number of threads or work items, enabling better parallelization and efficient GPU utilization. In contrast, the short and fat matrix achieves minimal speedup with the row-based decomposition because the limited number of rows restricts the number of active threads.

**Implementation 2: Multiple-row-based decomposition (1D tiling on rows of A)**

By further exploring the GPU programming, we use CUDA to perform the rest experiments. The CUDA platform support max block size up to 1024 in x dimension. To perform multiple row-based decomposition, we need to correctly identify the boundary of each thread (which row should each thread process and which row should each thread end). This can be simply done by correctly finding the index of each thread and multiply it by the row tilling size. Moreover, we want to verify part1 implementation if the block size 256 is the ideal size after we switch to CUDA device. By quickly swapping the block size, we get the left graph and the block size 256 can still provide the best speedup which is around 2x. The reason why the GPU didn't provide a large speedup is may due to the GPU device we are using – RTX4060 where it is a consumer device, the floating-point operation may not be as powerful as the other dedicated design GPUs. But the trend as we seen and the behaviour of the GPU should be enough to support the implementation. Another thing we observed is that the provided 'cudaEventElapsedTime' runtime API would provide the result in milliseconds, and the google benchmark would give the result in nanoseconds, so after each iteration, we have to multiply the CUDA runtime by 1e-3 to get the actual measurement of the CUDA runtime.




After finding the best block size 256 in the GPU for 4096x4096 gemm, we then swap the row sizes from 2 to 512 to see the effects on how the tilled row size affect the performance. From the graph below, we can find that when tilled row sizes increase from 1 to 512, the speedup over baseline will decrease. This is reasonable as we allocate more rows to a thread, a thread will perform more floating point computations than before which will eventually cause a increased CPU time.

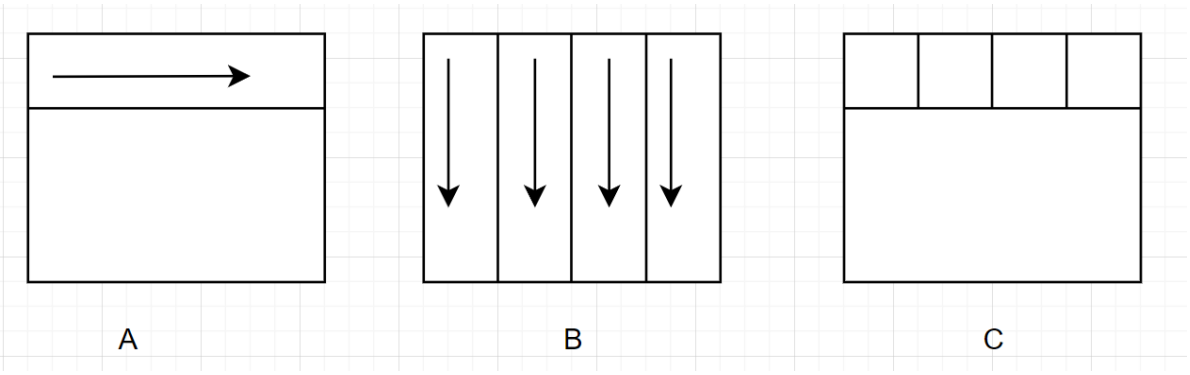



Similar to the first part, the tall/skinny matrix would provide a good speedup compared to the gemm_baseline since each thread is doing multiplication on each row and the original gemm_baseline have the avx enabled on row multiplication which it may not work well when the column size is small. From table on the left, we can see that the initial speedup 1 block_size is 8, and when the block size increases, the speedup will decrease as expected. For fat/Wide matric, the block size of m=32, n=4096 is chosen, but since we didn't do any optimization on the columns, which a wide column result the speedup slow compared to the AVX accelerated gemm code.

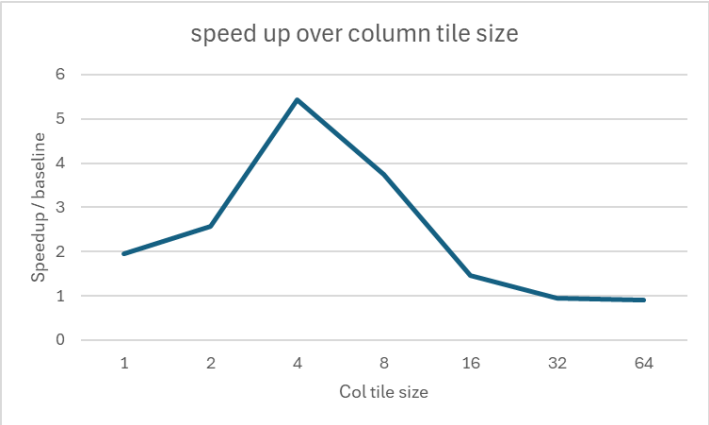
| Block_Size | Tall/Skinny | Fat/Wide |
|---|---|---|
| 1 | 8.27450419 | 0.105176177 |
| 2 | 7.500811623 | 0.065355038 |
| 4 | 4.332250871 | 0.035851412 |
| 8 | 2.23681532 | 0.018378916 |
| 16 | 1.079667143 | 0.009451212 |
| 32 | 1.051245214 | 0.004761068 |
| 64 | 0.795172329 | 0.004719161 |
| 128 | 0.481678598 | 0.004755906 |

## Implementation 3: 1D tiling on rows of A + 1D tiling on rows or columns of B

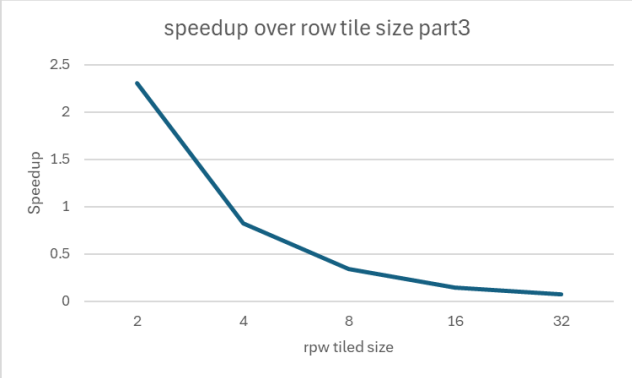
There are several ways to tile both rows and columns such as use a 2d dimension from CUDA, but in this case, we will explore the method by setting the boundaries correctly based on the thread Index. Below is a simple chart to display how we can assign each thread to perform the computation. Basically, for each row, it has to multiply by the vertical columns of B, so we can put each column of the matrix B into a thread and let it compute with the corresponding tilled row. We don't really need to spawn any new threads on matrix A since each tilled column thread of matrix B will multiply the A row, we just need to correctly set the tilled row boundary and assign correct number of threads based on the tilled row size of tilled column size in the kernel.




Based on what we have seen before, the increasing tilling on row will decrease the speedup, so firstly we want to explore the effect when the row tilling size is 1, what will happen by increasing the column tile size. By swapping the column tilling size, we observe that when the column tiles size starts increasing, the speedup gets increased as well, but once the column tile size reaches 4, the speedup starts decreasing. This may due to the increased memory usage and inefficient global memory access patterns. The large tile size could also reduce GPU occupancy and introduce thread divergence, therefore it's really important to find a balanced tile size.




To verify the increasing row tilling size would decrease the speedup, we fix the column size at 4 where it provides the best speed up and swap the tiled column size. From the graph below, we can verify that when the tiled row size increases, the speedup will decrease.




Finally, to investigate the effects on tall/skinny and short/wide matrices, with the patterns we observed before, we can estimate that for tall/skinny matrices, it will perform well when the row tile size is small as it will assign more threads to each row since it's row heavy computation, the column size may not impact performance that much but a balanced column tile size chosen would still improve the overall speed. And for the short wide matrices, since it is column heavy computation, assign more threads to each column would help increase the speedup.

By verifying the assumption, for tall/skinny matrices, the size 4096, 32 is chosen and column size 4 is fixed where we are going to swap the row size. And for short/wide matrices, the size 32, 4096 is chosen and row size 1 is fixed where we are going swap the column size.

| tilled row size | Speedup | tiled col size |
|---|---|---|
| 1 | 28.87906 | 4 |
| 2 | 13.57824 | 4 |
| 4 | 16.67046 | 4 |
| 8 | 25.81203 | 4 |
| 16 | 12.96626 | 4 |
| 32 | 6.364921 | 4 |
| 64 | 3.196587 | 4 |

| tiled col size | Speedup | tilled row size |
|---|---|---|
| 1 | 0.108115 | 1 |
| 2 | 0.225326 | 1 |
| 4 | 0.402811 | 1 |
| 8 | 0.579976 | 1 |
| 16 | 0.773338 | 1 |
| 32 | 0.920724 | 1 |
| 64 | 1.796143 | 1 |
| 128 | 3.545775 | 1 |
| 256 | 6.945244 | 1 |

By comparing these two result tables, we find that the result matches our assumptions where for tall skinny matrices, a small tilled row size would lead a large speedup and for short wide matrices, a large tiled column size would improve the speedup.

In conclusion, we learned that use both OpenCL and Cuda in this lab where we achieve the largest speedup 5x by using 1D tiling on rows of A + 1D tiling on rows or columns of B with the row tilling size 1 and the column tilling size 4. We learned that how to effectively use GPU architecture to speedup the program and also the limitations when working with the GPU.