Osama Shamout Dr. Hamdan Abdellaef
201906271 CSC447 – Parallel Prog.
Assignment 3

Performance Analysis of Matrix Multiplication Using GPU-based Parallel Computing

Introduction

Graphics Processing Units (GPUs), originally designed for rendering fast gaming graphics, have emerged as an attractive alternative due to their massive parallel processing capabilities. They have turned out to be exceptionally efficient at performing computations that can be parallelized, such as matrix multiplication.

In this report, we aim to analyze the performance of matrix multiplication using GPU-based parallel computing. We will assess different computational techniques, specifically regular and tiled matrix multiplication approaches, and analyze their efficiency on a Tesla T4 GPU. The implications of the GPU's hardware limitations, including shared memory size and thread block configuration, will be thoroughly discussed to better understand the observed performance metrics.

By leveraging GPU's capabilities, we aim to demonstrate how the right balance of computational complexity and parallel processing can lead to significant improvements in the speed and efficiency of matrix multiplication. Furthermore, this report seeks to provide valuable insights into optimizing GPU resource utilization and achieving optimal performance in large-scale matrix computations.

Parallelization of the Computation

Both implementations of the matrix multiplication were parallelized using CUDA's block and thread hierarchy. Each element of the output matrix is calculated independently, which gives us the opportunity to calculate multiple output elements concurrently. We use CUDA’s SM provided variables threadidx and blockIdx to manipulate and control the paralellizations.

Basic Matrix Multiplication

The basic CUDA implementation follows a straightforward approach: each thread calculates a single element of the output matrix by performing the dot product of the corresponding row from the first matrix and column from the second matrix.

```
function BasicMatrixMulti(M, N, P, height, width, depth)
Calculate row and col based on thread and block index
If(row < height AND col < depth)
    Initialize pvalue to 0 Loop over width
    Multiply corresponding elements of row and col and add to pvalue
    Write pvalue to P[row * depth + col]
```

Tiled Matrix Multiplication

The tiled version is a more sophisticated approach where each block of threads loads a small block (a tile) of the input matrices into shared memory. Shared memory is much faster than global memory, and by utilizing it, we're able to speed up the computation.

```
function TiledMatrixMultiplication(M, N, P, width, height, depth)
Define shared matrices Ms and Ns with TITLE_SIZE
    Calculate Row and Col
        Initialize Pvalue to 0 Loop over tiles
            Load elements from M and N into shared memory Ms and Ns
                Synchronize threads
                    Loop over TITLE_SIZE
                        Multiply corresponding elements of Ms and Ns and add to
Pvalue
                            Synchronize threads
                                Write Pvalue to P[Row * depth + Col]
```

GitHub Code

This code implementation and its accompanying data chart and python notebook are available at the following GitHub Repository:

[*https://github.com/OsamaShamout/MatrixMultiplication*](https://github.com/OsamaShamout/MatrixMultiplication)

Performance

In this study, we systematically gathered and analyzed various parameters, including matrix size, computation technique (regular or tiled), number of blocks, block/tile size, and parallel execution time in seconds. These metrics allowed us to critically assess the performance of our GPU implementation.

Central to our analysis is the total number of threads launched, computed as the product of the number of blocks in the grid and the size of each block. This metric is particularly significant in the context of GPU performance, as it offers a more representative measure of computational capability compared to traditional CPU-based metrics that focus on core/thread numbers. Further, to gain insights into the relative efficiency of our GPU execution, we also collected sequential execution time. This, in turn, facilitated the calculation of the speedup factor and efficiency - key indicators of parallel performance. The speedup factor, a ratio of the parallel to the sequential execution times, gives us a clear measure of how much performance improvement we gained by parallelizing our code. The efficiency, calculated as the speedup factor divided by the total number of threads, provides a normalized measure of performance that allows us to assess the effectiveness of resource utilization in our GPU implementation.

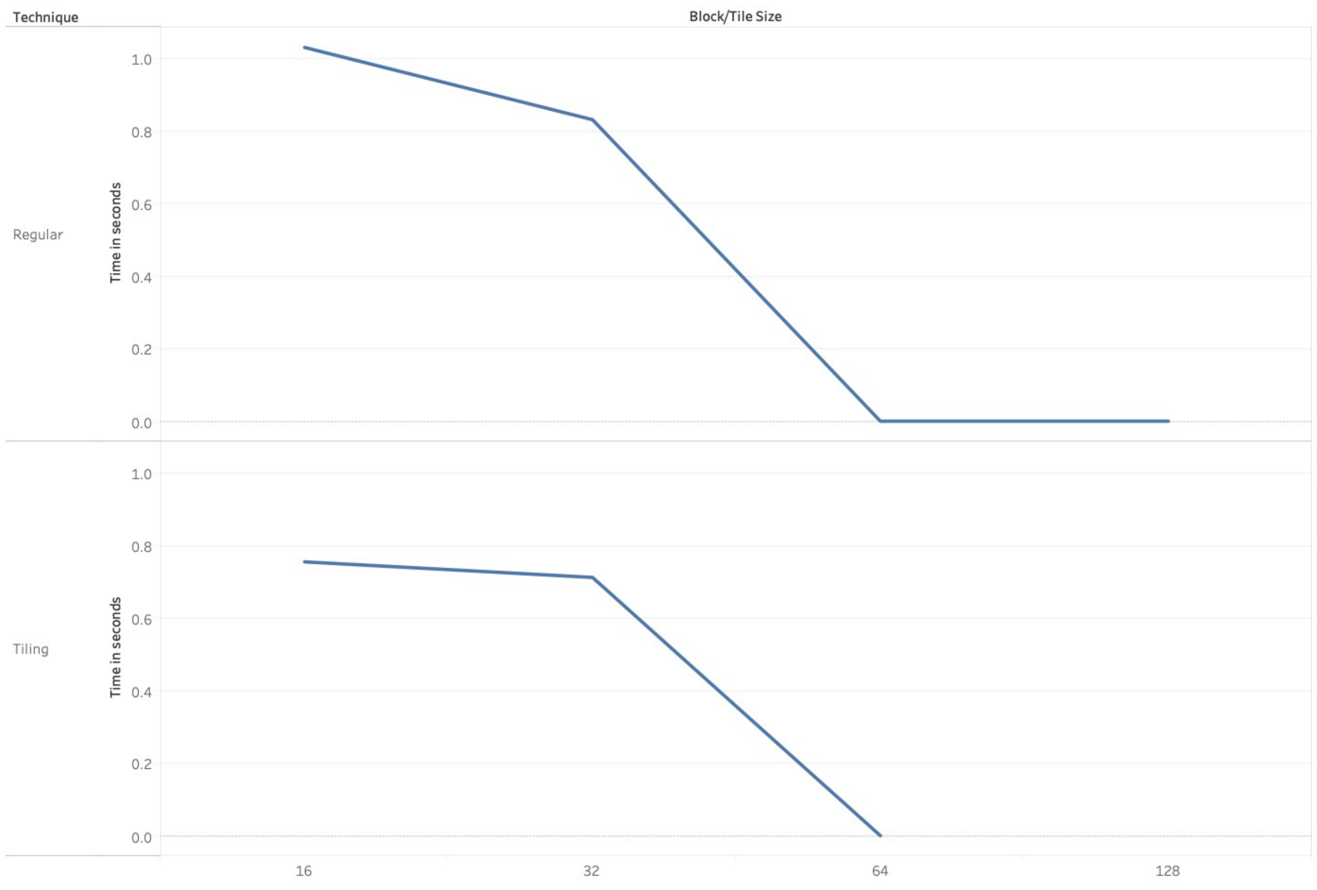
At first, we observed that the sequential time took a faster execution time than the Cuda C which was surprising however, upon optimizing the block and tile sizes this all differed. We observed that the tiled version is typically faster, especially for larger matrices. This is expected as in the tiled version, threads in a block collaborate to load tiles of the input matrices into the faster shared memory, where these tiles can be reused to compute multiple output elements. This reduces the need for slower global memory accesses and increases the memory bandwidth utilization. Also, it is imperative to note that we used multiples of 32 to accommodate for the Warp sizes and for the architecture’s hardware structure. The speed up obtained reached a factor of 200 in regular and tiling methods from the sequential method.

Note: the data has been cleaned when plotting and graphing. We discuss such “NaN” data points that were observed below. The graphs plotted compare the execution time, speed up factor, and the efficiency of the two techniques.

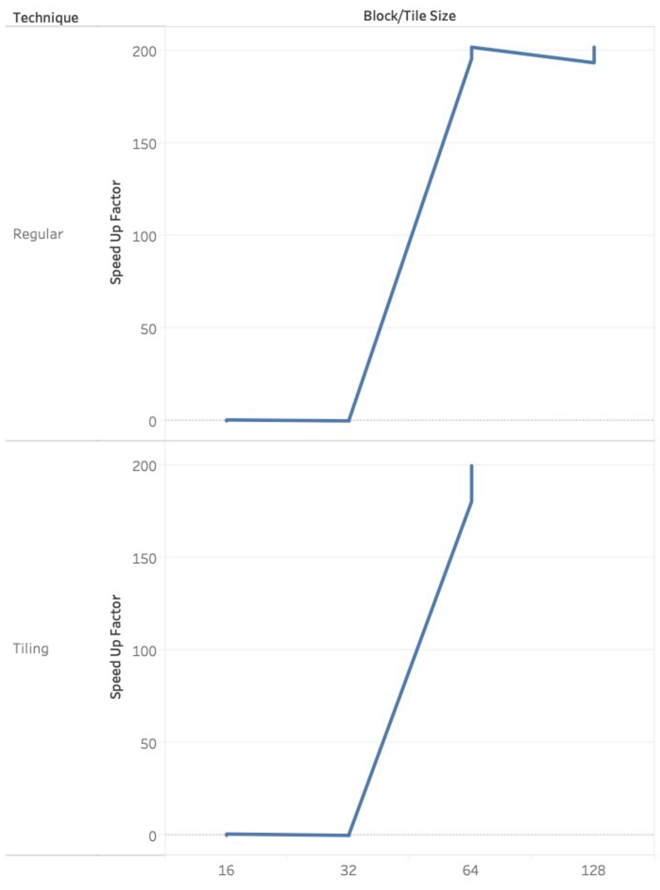
Osama Shamout
201906271
Assignment 3

Dr. Hamdan Abdellaef
CSC447 – Parallel Prog.

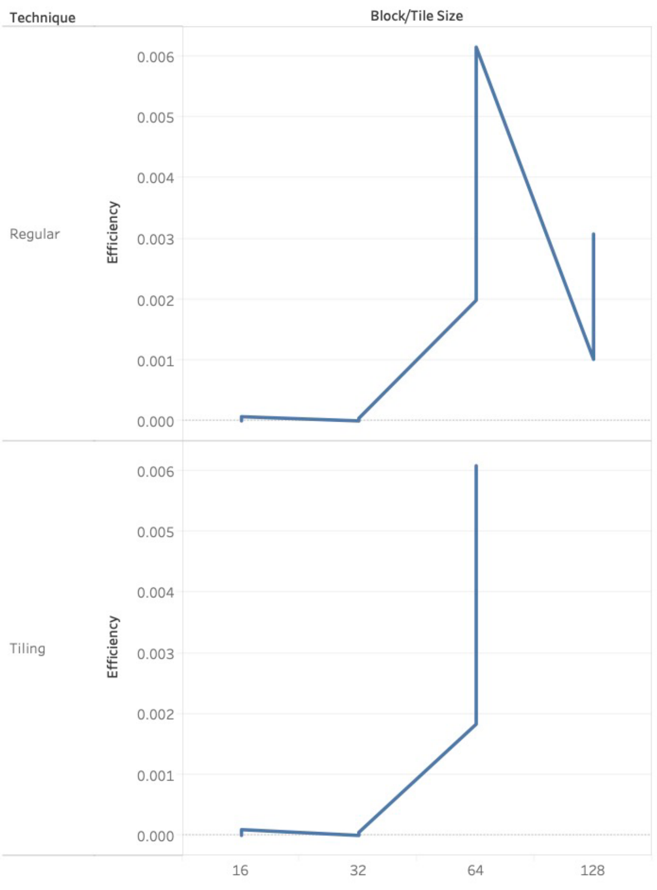
Execution Time over Block/Title Size



Speed Up Factor



Effeciency



Osama Shamout
201906271
Assignment 3

Dr. Hamdan Abdellaef
CSC447 – Parallel Prog.

Insights

Results

The execution time for the tiling is start at a shorter execution time than the regular way. Both the regular and the tiling techniques seem to achieve robust parallelization results. Perhaps if we had more data points and more complex matrices, we would start to see a differentiating cut between the two techniques. The speed up factors reached high results up to 200. Yet still, it is still not fully understood how to further gain efficiency in the system as it significantly low. Hence, it is important to note how execution is significantly faster yet the efficiency itself is not high.

Segmentation Fault

In sequential computation, Thus, for each element in the result matrix, we perform 1000 floating-point multiplications and 999 floating-point additions (totaling 1999 floating-point operations). Since the resulting matrix will have 1000x1000 = 1,000,000 elements, the total number of floating-point operations required for the multiplication of two 1000x1000 matrices would be 1,000,000 * 1999 = approximately 2 billion floating-point operation.

Specifically, for a matrix of size 1000x1000, we require storage for 1,000,000 elements. Given that we're dealing with floating-point numbers, each requiring 4 bytes of memory, the resulting matrix alone necessitates 4,000,000 bytes, or approximately 4 MB of memory.

This demand for memory does not even account for the memory needed for the input matrices or any other program-specific variables. In a standard program setup, this memory size exceeds the typical allocation reserved for the stack, leading to the absence of results for matrices of sizes 1000 and, by extension, 1500. This was also observed not just on the Google Colab, but also on my MacBook device, where attempting to compute for such sizes resulted in a segmentation fault. This points to an intrinsic limitation concerning memory management in handling large matrix computations.

Maximum Shared Memory in Tiling

The maximum shared memory size per block for the GPU is 49152 bytes. This was noted from the line observed in the python notebook: **0xc000 max**. Therefore when storing two tiles (one from matrix A and one from matrix B) into shared memory, we will need to divide the total shared memory size by 2 to accommodate both tiles. Since we're storing single-precision floats (which are 4 bytes each), then we can calculate the maximum dimension of our tile as follows:

49152 bytes/2/4 bytes/float = 6144 floats and sqrt(6144) ~= 78.

Hence, we can see the correct output showing in tile size 78 but not for 79, and by extension, anything larger. Therefore, when the tile size was set to 128, no value was produced. This outcome is due to the specific GPU used in this case, a Tesla 4, which due to its hardware architecture, cannot support such a large tile size as compared to the requirements of the basic implementation.