MATMUL Optimisation

Name: Daksh Yadav
Roll No.: 23BT10010

KOSS SELECTION TASK

> git-init..

we'll talk about:

01 Introduction, and my understanding of the problem

02 Mathematical Optimizations

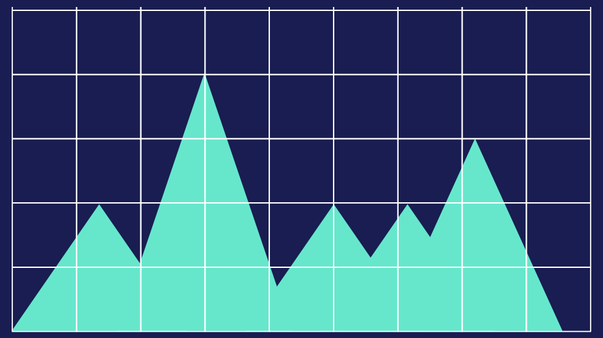
03 Language/Compiler Optimizations

04 Algorithmic Optimizations

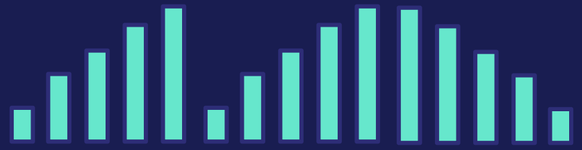
05 Memory Optimizations

06 CUDAAAA (and some BLAS)

07 Results and closing



random graphs to make this slide look *intense*



Let's get started

The task given was to optimize the process of multiplying two matrices. My main focus on this task was to make it compute as fast as possible, i.e. optimizing the **time complexity**. However, ignoring the other aspect of execution, **space complexity** often leads to results that highly motivated me ~~to rage quit.~~

Initially, I assumed this task to be primarily mathematically focused, hence my first intuition said to look for alternative ways of multiplying two matrices. One such approach was using the **Strassen's Algorithm**.

Mathematical Optimizations

Spoiler: the results didn't make sense

Naive Approach

The naive matrix multiplication algorithm computes each element of the resulting matrix by taking the sum of products of corresponding elements from the rows of the first matrix and the columns of the second matrix. This is achieved through nested loops iterating over the rows and columns of the matrices. While simple to implement, the naive algorithm has a time complexity of O(n^3), making it inefficient for large matrices due to the high number of arithmetic operations involved. Nonetheless, for small matrices, it remains a viable option.

Strassen's Algorithm

The Strassen's Algorithm reduces the number of multiplications that take place by using additional additions and subtractions. This follows a divide-and-conquer approach by breaking each matrix in 4 sub-matrices, and keeps breaking each sub-matrix down recursively until they are small enough to be computed efficiently. All the sub-matrix products are then combined using arithmetic operations to get the final result.

Benchmarking Results

These results show the average time taken to compute the product of two 100x100 matrices across 100 runs. Taken from my spicy benchmarking tool in the repo.

The Standard Algorithm

Optimized Standard Algorithm

The Strassen's Algorithm

```
Running python ./python/Naive.py
[##################################################] 100.00%
Average for python ./python/Naive.py : 92.98530340194702 ms

Running python ./python/Numpy.py
[##################################################] 100.00%
Average for python ./python/Numpy.py : 0.16771554946899414 ms

Running python ./python/StrassenInPy.py
[##################################################] 100.00%
Average for python ./python/StrassenInPy.py : 371.09222412109375 ms
```

Why is Strassen's Algorithm slower?

Strassen's algorithm, while efficient for large matrix sizes, can be slower for smaller sizes due to its **recursive nature and overhead**. For small matrices, the overhead of splitting the matrices into smaller submatrices and performing additional additions and subtractions **outweighs the benefits of reduced multiplications**.

The recursive nature of Strassen's algorithm incurs additional function call overhead and memory allocation overhead, which can become significant for smaller problem sizes. Additionally, the base case threshold, where the recursion stops and conventional matrix multiplication is used, can be reached quickly for smaller matrices, leading to unnecessary recursion overhead.

Language/Compiler Optimization

Interpreted vs Compiled

- In an interpreted programming language, the source code is executed line by line by an interpreter program.
- In a compiled programming language, the source code is translated into machine code or bytecode before execution.
- This difference makes compiled languages like C++ way faster than Python for computations where execution time needs to be optimized.
- You can run the benchmarking tool to see the exact

```
Running python ./python/Naive.py
[##################################################] 100.00%
Average for python ./python/Naive.py : 95.81132173538208 ms

Running ./executables/naive.exe
[##################################################] 100.00%
Average for ./executables/naive.exe : 2.62 ms

Running ./executables/vectorMultiplication
[##################################################] 100.00%
Average for ./executables/vectorMultiplication : 9.35 ms
```

Compiler Flags

For compiled languages like C/C++, the compiler has several built-in settings that can be changed by using appropriate flags during compilation. These flags can activate special optimization techniques like loop unrolling, fast math approximation, etc

-O3 OPTIMIZATION FLAG

This flag activates a high level of optimization, typically enabling aggressive optimizations such as loop unrolling, function inlining, etc, and aims to maximize the performance of the compiled code, potentially at the expense of increased compilation time and code size.

-FUNROLL-LOOPS

This flag directs the compiler to perform loop unrolling optimization, where loop iterations are expanded and executed sequentially rather than being executed within a loop structure. This optimization aims to reduce loop overhead and improve instruction-level parallelism

-FTREE-VECTORIZE

This flag enables automatic vectorization optimization, where the compiler transforms scalar code into vectorized code suitable for SIMD (Single Instruction, Multiple Data) instructions. It aims to exploit parallelism by performing multiple operations simultaneously.

Vectorization

Algorithmic Optimizations

Vectorization accelerates matrix multiplication by exploiting parallelism through SIMD (Single Instruction, Multiple Data) instructions. In a vectorized implementation, the CPU performs multiple arithmetic operations simultaneously on vector units, which can process multiple data elements in parallel.

Here's how I visualize the concept of SIMD:

Regular Loop Iteration (MIMD)

1 -> Multiply by 4 -> 4
2 -> Multiply by 3 -> 6
3 -> Multiply by 5 -> 15

This process usually takes place on one thread only.

Vectorized Loop (SIMD)

$$\begin{bmatrix} 5 \\ 6 \\ 7 \end{bmatrix} \text{ -> Multiply by 5 -> } \begin{bmatrix} 25 \\ 30 \\ 35 \end{bmatrix}$$

The same instruction is run on multiple threads with multiple data elements.

Memory Optimizations

Concept of Cache

When a program runs, it requires data in a very fast manner. Reading the data from SSD or RAM is often not the optimal solution. Hence, data is first copied to the much faster Cache memory which exists inside the processor itself.

The cache size is usually very small and there are three levels:

- **L1 Cache**: It is the smallest and fastest cache, located inside the core of the CPU. Each thread has its own L1 cache.
- **L2 Cache**: This cache level is slightly slower than L1 but is larger in size. Each core has its own L2 cache.
- **L3 Cache**: It is the largest cache and is located outside the cores. Its speed is slower than other caches but is still double the speed of DRAM.

L1

32 KB TO 128 KB

L2

256 KB TO 512 KB

L3

> 2 MB

Cache Misses

Cache misses occur when the CPU tries to access data or instructions that are not currently stored in the cache memory. This situation happens because the required data was either never in the cache or was evicted due to limited cache space. Cache misses result in the CPU needing to retrieve the data from slower main memory, leading to increased access latency and reduced performance. Minimizing cache misses is essential for improving overall system efficiency and optimizing performance.

To reduce the number of cache misses, one must make sure to organize the matrix data in memory to maximize **spatial locality**, ensuring that elements accessed together are stored close to each other. This reduces the likelihood of cache misses when accessing matrix elements.

This can be achieved by simply changing the order of the for-loops in the standard algorithm. This reordering ensures that elements of the matrices are accessed in a contiguous manner within the innermost loop, which can lead to fewer cache misses and improved performance.

Based on benchmarks..

Notice the order of counter variables in the loops.

```
for (int i = 0; i < SIZE; ++i)
{
    for (int j = 0; j < SIZE; ++j)
    {
        for (int k = 0; k < SIZE; ++k)
        {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
int r;
for (int k = 0; k < SIZE; ++k)
{
    for (int i = 0; i < SIZE; ++i)
    {
        r = A[i][k];
        for (int j = 0; j < SIZE; ++j)
        {
            C[i][j] += r * B[k][j];
        }
    }
}
```

```
Running ./executables/CacheMissesTestIJK
[##################################################] 100.0
Average for ./executables/CacheMissesTestIJK : 3.652548 ms
```

```
Running ./executables/CacheMissesTestKIJ
[##################################################] 100.00%
Average for ./executables/CacheMissesTestKIJ : 2.6830160000000
```

NVIDIA's CUDA

Compute Unified Device Architecture

- CUDA is a programming platform/model which allows programs to be run on Nvidia's supported GPUs. Graphic cards are specialized devices that have an insanely high number of cores that can all do operations in parallel, making them efficient for tasks like graphic processing, which requires a large number of algebraic calculations to be done for each frame.

- CUDA allows us to run any C++ code on the GPU unit. The large number of cores allows us to compute matrix multiplication very quickly by dividing the loop iterations between the separate threads of the GPU.

- Programming for CUDA is fairly similar to dynamic programming in C or C++. You start by creating a kernel function that contains the instructions to be operated on each core. Instead of using counter variables and for-loops, we use the intrinsic variables **blockIdx** and **blockDim** which returns the ID of the thread on which it is called. This is then used to define new counter variables and program the parallelized for-loop.

- Then, we simply allocate memory to the GPU and copy the data to it. The major difference in programming for CUDA is that each little operation like moving data to GPU, reading output to host, etc. needs to be done manually using the appropriate functions.

BLAS (or not?)

Basic linear algebra subfunctions are low-level fundamental operations in linear algebra, such as vector addition, scalar multiplication, dot product, cross product, matrix addition, matrix multiplication, and matrix transposition. These operations are optimized for performance and efficiency, making them faster and suitable for lower-level programming.

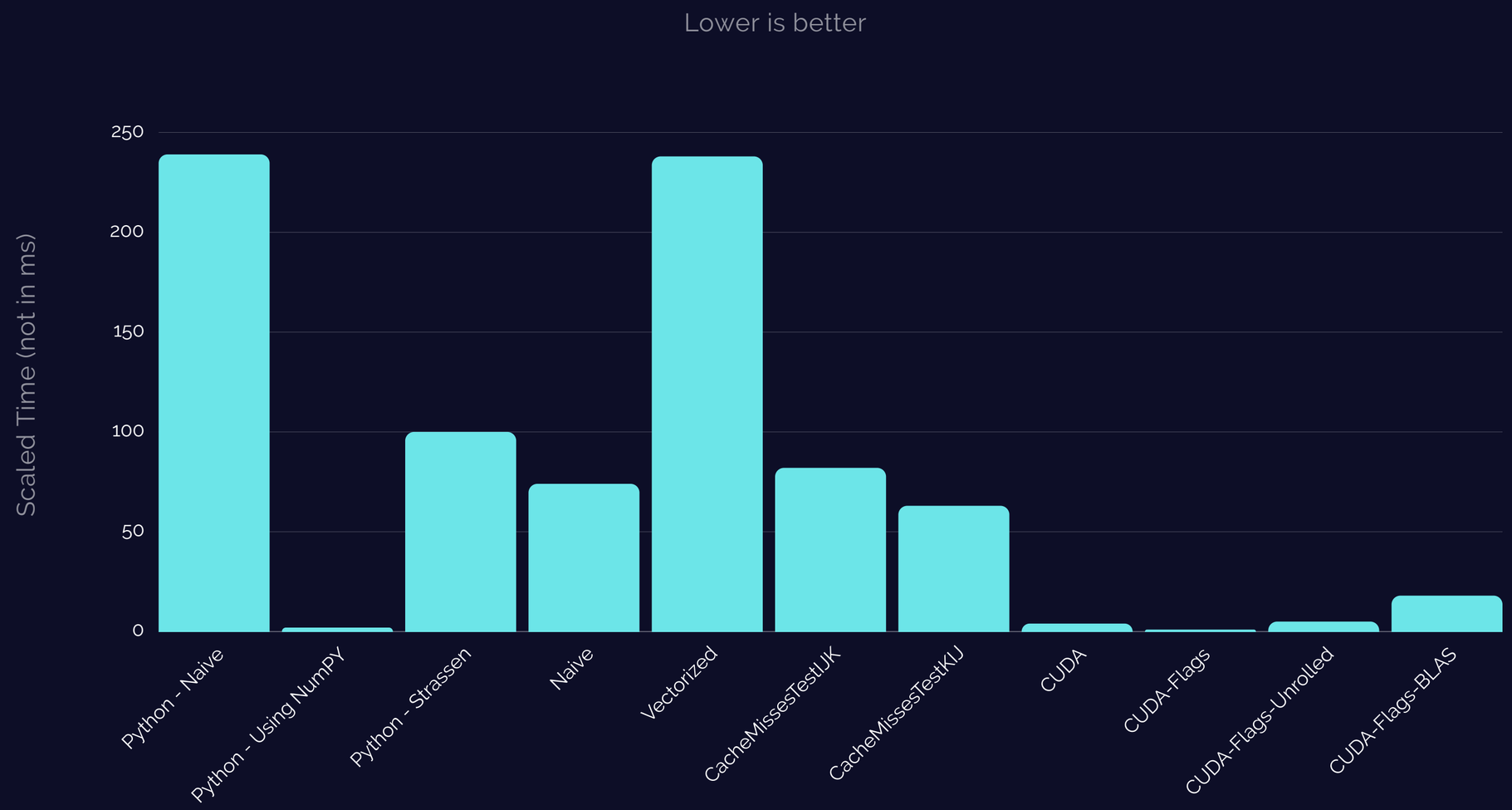
However, in my tests, I found that using BLAS functions instead of regular CUDA operators is slower. From my understanding, this happens due to the same reason that slowed down Strassen's algorithm. Such methods are only useful for very large matrices.

Using BLAS functions like **sgemm** (for single precision general matrix multiplication) allows lesser control over other factors that can be optimized like the order of loops without changing the low-level code itself.

```
Running ./executables/CUDA-Flags
[##################################################]
Average for ./executables/CUDA-Flags : 0.174525 ms
```

```
Running ./executables/CUDA-Flags-BLAS
[##################################################] 100.00%
Average for ./executables/CUDA-Flags-BLAS : 4.309177 ms
```

Finally, the entire benchmark!



And we're done for the day!

Thanks for reading. This task was genuinely enjoyable as more than teaching new things, it just gave me clarity on things I already knew but didn't understand. As I studied for this, a lot of things started clicking in, and that was an amazing feeling.