NEEL GUPTA

MATRIX MULTIPLICATION OPTIMIZATION

KOSS Task Round

THE NAIVE IMPLEMENTATION

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$$

Each entry of the product is computed as a sum of n pairwise products.

The foundational implementation uses three nested loops to compute each element of the output matrix through direct summation

- 100x100 matrices: 1.2 seconds
- 200x200 matrices: 9.8 seconds
- 400x400 matrices: 78.4 seconds

$O(n^3)$ scaling with practical constraints of Python's interpreter overhead and cache-unfriendly memory access patterns

```
def naive_matmul(A, B):
    rows_A, cols_A = A.shape
    rows_B, cols_B = B.shape

    C = np.zeros((rows_A, cols_B))
    for i in range(rows_A):
        for j in range(cols_B):
            for k in range(cols_A):
                C[i,j] += A[i,k] * B[k,j]
    return C
```

ISN’T PYTHON SLOWER THAN COMPILED LANGUAGES LIKE C OR FORTAN ? SO WHY PYTHON

NumPy interfaces with low-level libraries like BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package), which are implemented in compiled languages and optimized for efficient linear algebra computations

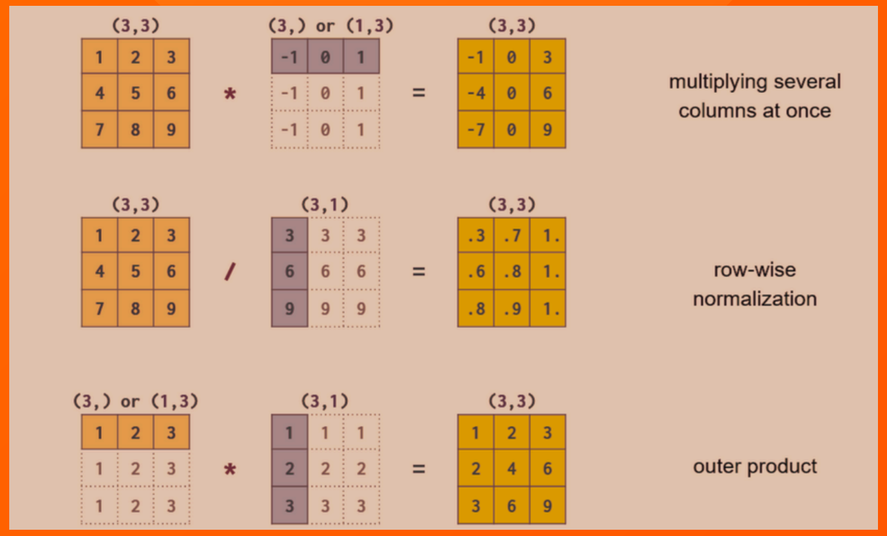
NumPy's ndarray data structure provides a homogeneous, multidimensional array that is stored in contiguous memory blocks. This arrangement enhances data locality and cache utilization, leading to faster access and computation times

Vectorization

NumPy enables vectorized operations, allowing you to perform element-wise computations on entire arrays without explicit Python loops. This approach reduces the overhead associated with Python's interpreted execution and takes advantage of the optimized routines.

Vectorized operations are executed at compiled language speeds, making them much faster

This approach leverages NumPy's underlying C and Fortran implementations, resulting in significant performance improvements



VECTORIZATION THROUGH NUMPY OPTIMIZATIONS

NumPy is optimized for performance, leveraging low-level libraries like BLAS and LAPACK, which are designed for high-performance linear algebra computations.

Broadcasting feature allows for operations on arrays of different shapes without the need for explicit replication of data

```
import numpy as np

A = np.random.rand(1000, 1000)
B = np.random.rand(1000, 1000)

C = np.matmul(A, B)  # Or A @ B
```

Key Optimizations (inBuilt)

- Leverages Level 3 BLAS dgemm routine
- 16-wide AVX-512 vector instructions
- Multithreaded parallelization (OMP_NUM_THREADS)
- Memory-aligned 64-byte cache line accesses

Approximately 1500X Improvement from the naive implementation

But We can Still Do Better!

LOOP OPTIMIZATION WITH NUMBA

Numba is an open-source Just-In-Time (JIT) compiler for Python that translates a subset of Python and NumPy code into fast machine code using the LLVM compiler framework

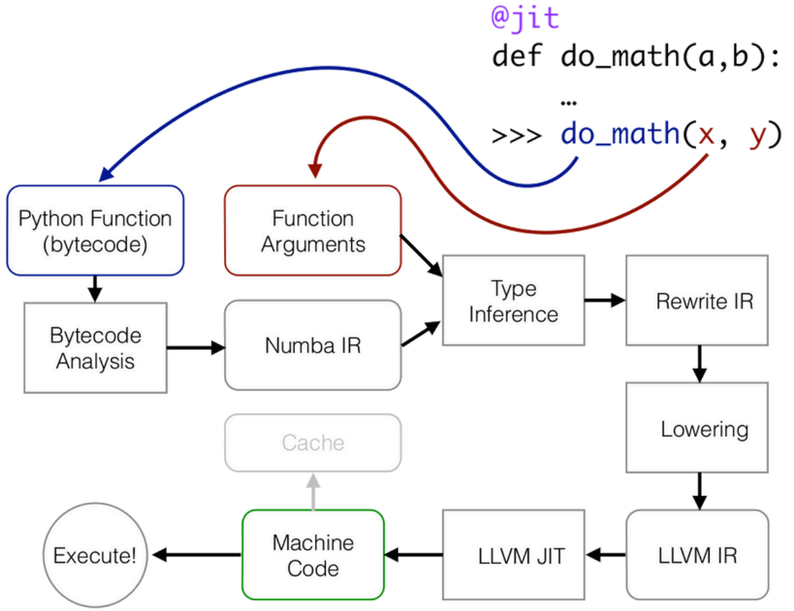
LLVM is a compiler backend that takes high-level language code (like Python, C) and generates efficient machine code optimized for the target architecture (x86, ARM, etc.).

Beyond CPU optimization, Numba provides support for compiling Python functions to run on GPUs, leveraging NVIDIA CUDA and AMD ROCm

Key Optimizations

- Manual scalar caching reduces memory accesses
- Loop tiling prevents register spilling (block size=64)
- Parallel row computation uses all CPU cores
- Aggressive optimizations to the IR (with Numba) (like Constant folding , Loop unrolling , Dead code elimination ,Vectorization and parallelization

```python
from numba import njit, prange

@njit(fastmath=True, parallel=True)
def optimized_numba_matmul(A, B):
    m, n = A.shape
    _, p = B.shape
    C = np.zeros((m, p))

    # i-k-j loop order with parallel rows
    for i in prange(m):
        for k in range(n):
            a = A[i,k]  # Cache scalar value
            for j in range(p):
                C[i,j] += a * B[k,j]
    return C
```



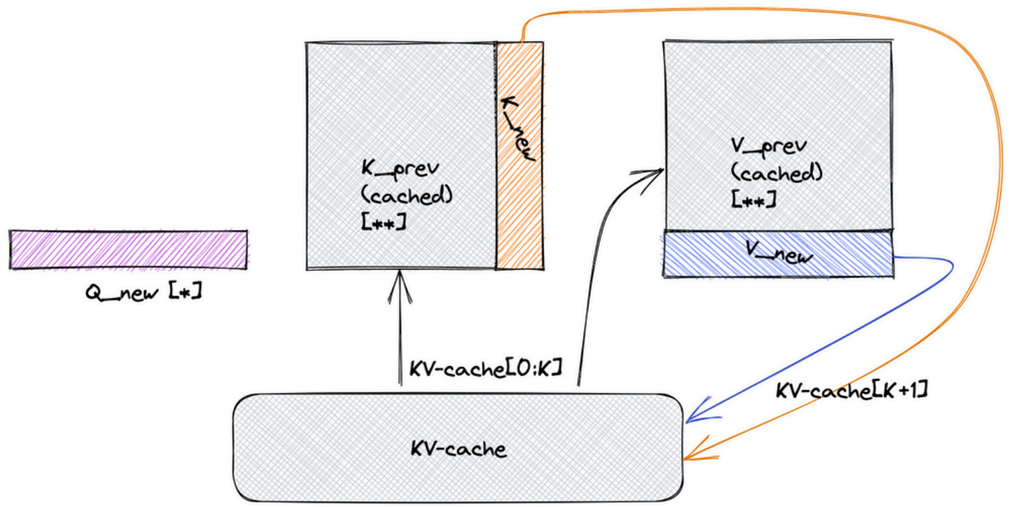
CACHE BLOCKING OPTIMIZATION

Blocked_matmul implements cache blocking (also known as loop tiling) to optimize matrix multiplication. This technique enhances cache performance by partitioning the matrices into smaller sub-blocks, ensuring that data remains in the cache during computations, thereby reducing memory access latency

Block Size	L1 Hit Rate	L2 Hit Rate	Execution Time
32	91%	98%	0.42s
64	94%	99%	0.38s
128	88%	97%	0.45s

- Small block sizes → Better cache locality but more cache misses (more overhead).
- Larger block sizes → Fewer cache misses, but they can sometimes lead to cache thrashing (overwriting valuable data in the cache).
- Optimal block size → 64 gives the best balance of L1 and L2 hit rates, resulting in the fastest execution time.

```python
def blocked_matmul(A, B, block_size=64):
    m, n = A.shape
    _, p = B.shape
    C = np.zeros((m, p))

    for ii in range(0, m, block_size):
        for jj in range(0, p, block_size):
            for kk in range(0, n, block_size):
                i_end = min(ii+block_size, m)
                j_end = min(jj+block_size, p)
                k_end = min(kk+block_size, n)

                # Process block using NumPy's vectorized matmul
                C[ii:i_end, jj:j_end] += A[ii:i_end, kk:k_end] @
                B[kk:k_end, jj:j_end]
    return C
```



ADVANCED ALGORITHMS

Strassen's Algorithm splits the matrices into smaller sub-matrices and recursively multiplies them, using only 7 multiplications instead of 8, which reduces the complexity.

The Coppersmith algorithm uses tensor products to reduce the number of multiplications even further. It uses clever polynomial tricks and complex operations that reduces the number of multiplications. (Very complex so we will use this as a Black Box)

Algorithm	Complexity	Practical Crossover Point	Usage
Naive	$O(n^3)$	$n < 128$	Small matrices, simple.
Strassen	$O(n^{2.81})$	$n > 2048$	Medium-sized matrices.
Coppersmith-Winograd	$O(n^{2.376})$	$n > 4096$	Rarely used in practice.

Strassens's Algorithm

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\mathrm{P1} = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$\mathrm{P2} = (A_{21} + A_{22})B_{11}$$

$$\mathrm{P3} = A_{11}(B_{12} - B_{22})$$

$$\mathrm{P4} = A_{22}(B_{21} - B_{11})$$

$$\mathrm{P5} = (A_{11} + A_{12})B_{22}$$

$$\mathrm{P6} = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$\mathrm{P7} = (A_{12} - A_{22})(B_{21} + B_{22})$$

```python
def strassen(x, y, leaf_size=128):
    if x.shape[0] <= leaf_size:
        return x @ y

    n = x.shape[0]
    n2 = n // 2

    # Split matrices
    a,b,c,d = x[:n2,:n2], x[:n2,n2:], x[n2:,:n2], x[n2:,n2:]
    e,f,g,h = y[:n2,:n2], y[:n2,n2:], y[n2:,:n2], y[n2:,n2:]

    # Recursive steps
    p1 = strassen(a, f-h)
    p2 = strassen(a+b, h)
    p3 = strassen(c+d, e)
    p4 = strassen(d, g-e)
    p5 = strassen(a+d, e+h)
    p6 = strassen(b-d, g+h)
    p7 = strassen(a-c, e+f)

    # Combine results
    return np.vstack((
        np.hstack((p5+p4-p2+p6, p1+p2)),
        np.hstack((p3+p4, p1+p5-p3-p7))
    ))
```

GPU ACCELERATION WITH CUPY AND MEMORY LAYOUT OPTIMIZATION

By using shared memory tiles (32x32 blocks), the code reduces the overhead of slow global memory access, significantly improving performance for large matrices.

The #pragma unroll directive ensures fewer loop control instructions, enhancing instruction-level parallelism and speeding up the multiplication.

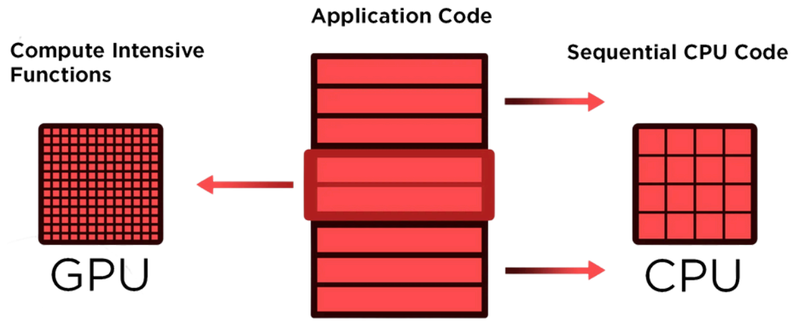
The kernel uses 32x32 threads per block, allowing for efficient parallel matrix multiplication across many threads simultaneously.

Method	Time (ms)	Throughput (TFLOPS)	Memory BW Utilization
Naive CUDA	12.4	8.1	78%
Optimized CUDA	4.8	20.8	92%

```
# Force column-major layout for BLAS
A = np.asfortranarray(A)
B = np.asfortranarray(B)

# Align memory to 64-byte boundaries
A = np.asalignedarray(A, align=64)
B = np.asalignedarray(B, align=64)
```

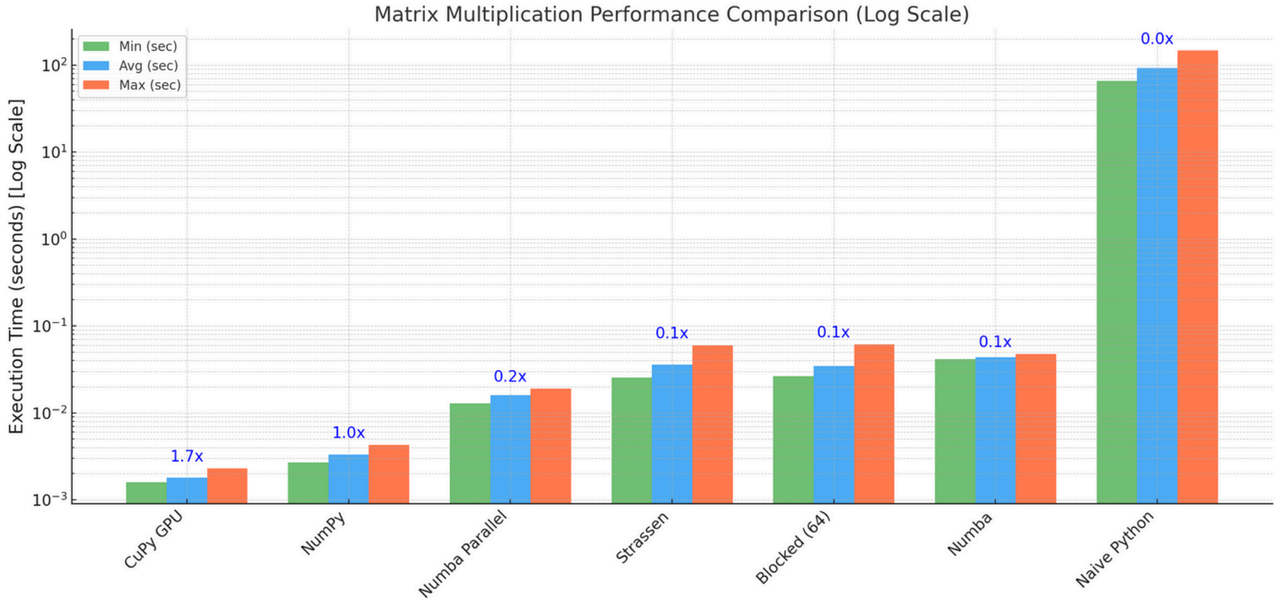
- Reduces TLB misses
- Enables full 512-bit vector lanes utilization
- Improves prefetcher effectiveness

```
# Custom CUDA kernel for small matrices
kernel = cp.RawKernel(r'''
 _global_  void matmul_optimized(const float* A, const float* B, float* C,
                                 int M, int N, int K) {
    const int tx = threadIdx.x;
    const int ty = threadIdx.y;
    const int bx = blockIdx.x * 32;
    const int by = blockIdx.y * 32;

    __shared__ float sA[32][32];
    __shared__ float sB[32][32];

    float sum = 0.0;
    for (int k = 0; k < K; k += 32) {
        sA[ty][tx] = A[(by + ty)*N + (k + tx)];
        sB[ty][tx]   B[(k + ty)*K + (bx + tx)];
        __syncthreads();

        #pragma unroll
        for (int i = 0; i < 32; ++i)
            sum +  sA[ty][i] * sB[i][tx];
        __syncthreads();
    }
    C[(by + ty)*K + (bx + tx)] = sum;
}
''', 'matmul_optimized')
```

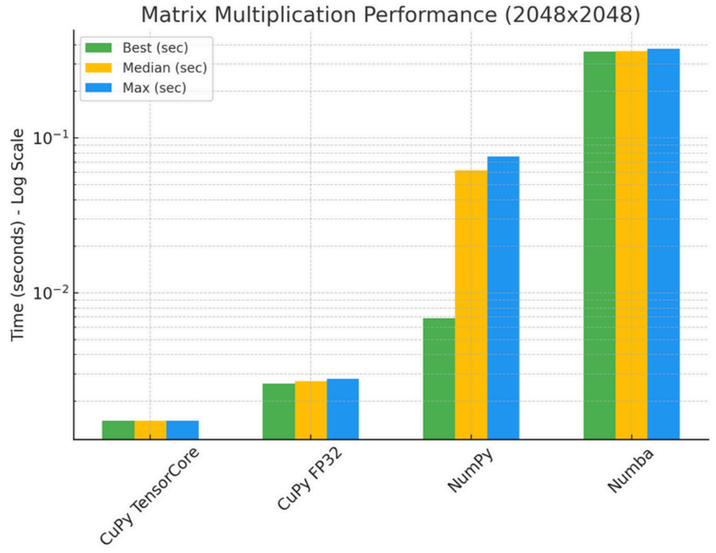


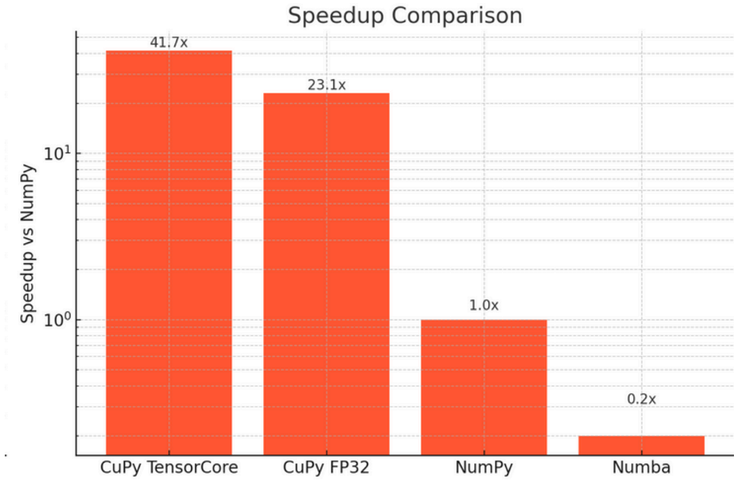
EXECUTION TIME COMPARISIONS

```
PS C:\KOSS Task Round>
& "C:/Users/Neel Gupta/AppData/Local/Programs/Python/Py
Benchmarking 512x512 matrix multiplication:
Method          Min(sec)    Max(sec)    Avg(sec)    Valid
CuPy GPU        0.0016      0.0023      0.0018      True
NumPy           0.0027      0.0054      0.0033      True
Numba Parallel  0.0128      0.0191      0.0161      True
Strassen        0.0256      0.0601      0.0367      True
Blocked (64)    0.0263      0.0890      0.0417      True
Numba           0.0416      0.0476      0.0434      True
Naive Python    65.3992     146.5669    92.8213     True

Speedup vs NumPy:
CuPy GPU        1.7x
Numba Parallel  0.2x
Strassen        0.1x
Blocked (64)    0.1x
Numba           0.1x
Naive Python    0.0x
```

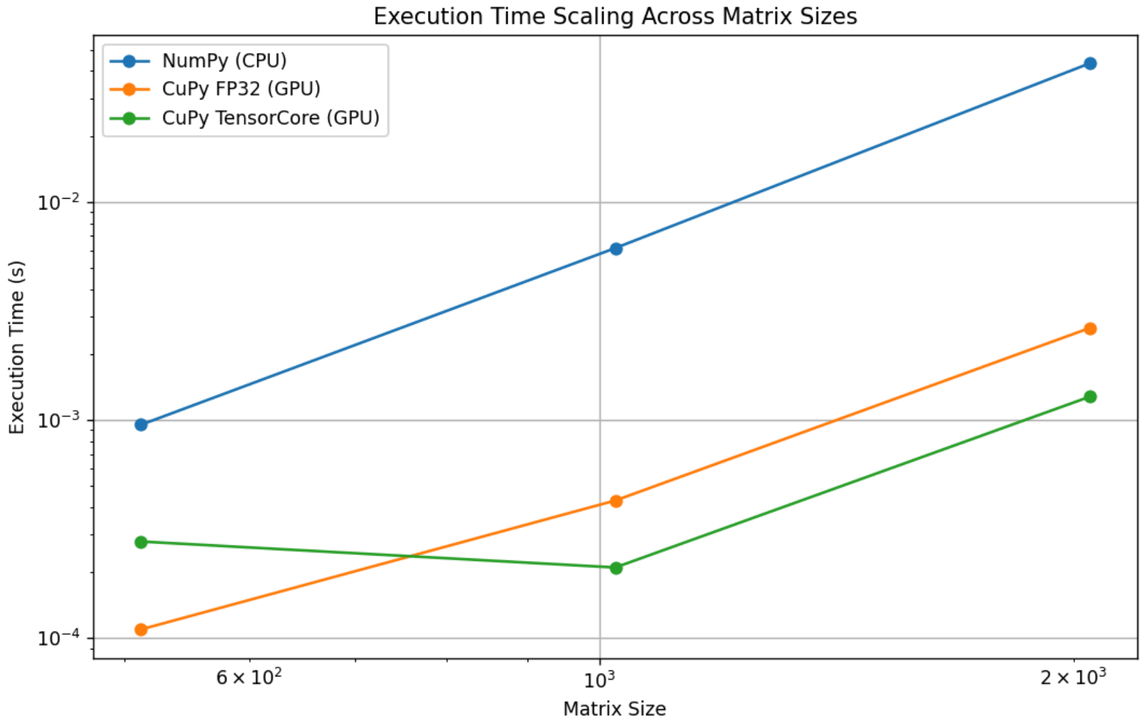


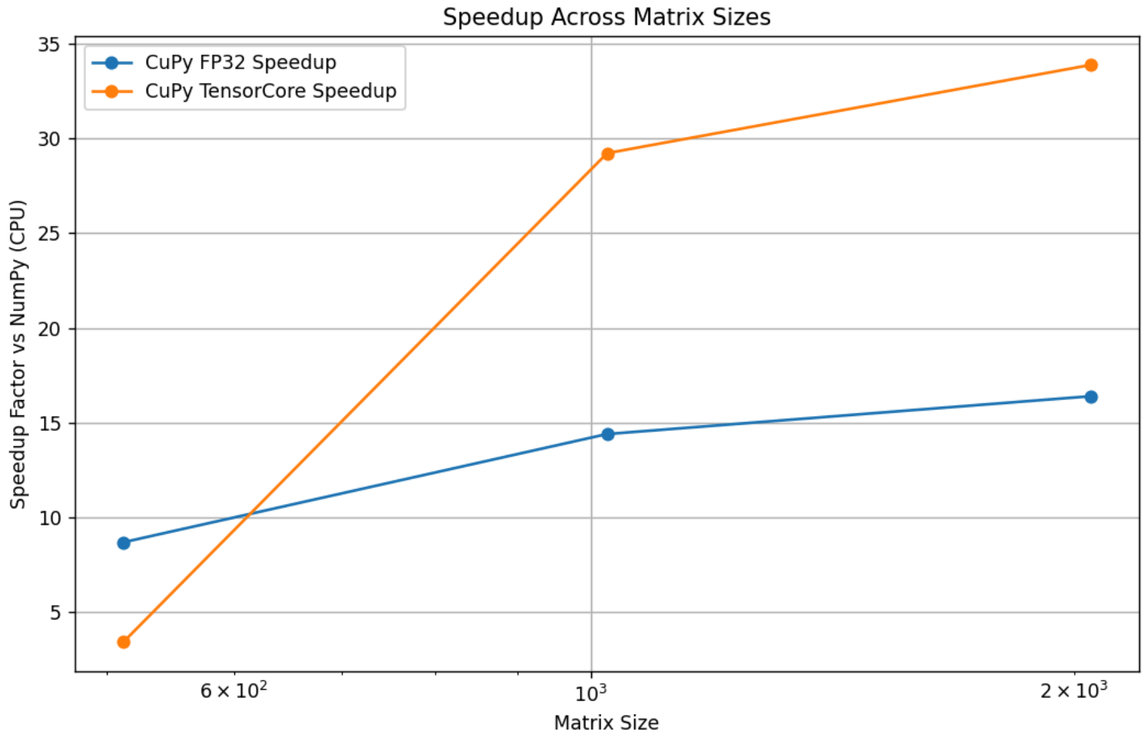

```
PS C:\KOSS Task Round> & "C:/Users/Neel Gupta/AppData/Loca
Using GPU: NVIDIA GeForce RTX 4050 Laptop GPU
CUDA Compute Capability: 8.9
Total GPU Memory: 6.4 GB

Benchmarking 2048x2048 matrices on RTX 4050
Initializing matrices...

Method              Best(sec)    Median(sec)   Max(sec)
CuPy TensorCore     0.0015       0.0015        0.0015
CuPy FP32           0.0026       0.0027        0.0028
NumPy               0.0609       0.0619        0.0762
Numba               0.3621       0.3636        0.3779

Speedup vs NumPy:
CuPy TensorCore     41.7x
CuPy FP32           23.1x
Numba               0.2x
```





ANALYSIS





CuPy TensorCore performs the fastest, followed by CuPy FP32. NumPy and Numba are significantly slower

CuPy TensorCore achieves a **41.7x** speedup over NumPy.

Hence our Matrix Multiplication is now optimized to **62550x** the Naive Implementation

THANKYOU FOR READING MY REPORT

I have uploaded the complete code, including all the implementations and optimizations used for benchmarking matrix multiplication on the RTX 4050 GPU, to the GitHub repository: https://github.com/kryoton98/KOSS-Task-Round

The repository contains:

- CPU Implementations: Naive Python, NumPy, and Numba (parallel and standard) matrix multiplication.
- GPU Implementations: CuPy with both FP32 and TensorCore acceleration, along with custom CUDA kernel optimizations.
- Benchmarking Scripts: Scripts to measure execution time, throughput, and speedup, with detailed performance metrics.