# Loop Unrolling Impact on CUDA Matrix Multiplication Operations

Vojdan Stefkovski, Dimitar Mileski, Marjan Gusev
*Faculty of Computer Science and Engineering*
Skopje, Republic of North Macedonia
vojdan.stefkovski@students.finki.ukim.mk, {dimitar.mileski, marjan.gushev}@finki.ukim.mk

***Abstract*—This paper investigates the impact of loop unrolling on CUDA matrix multiplication operations' performance across NVIDIA GPUs. We benchmarked both basic and unrolled kernels with varying unroll factors (2, 4, 8, and 16) and CUDA block sizes (8, 16, and 32) on matrices ranging from 128 × 128 to 4096 × 4096. Using two GPUs, the GeForce RTX 4060 and GTX TITAN X, we analyze how unrolling factors impact execution time. Our findings indicate that loop unrolling, particularly with factors of 8 and 16 and a block size of 32, yields significant performance gains on larger matrices. These results confirm loop unrolling as an effective optimization technique for CUDA matrix operations, providing insights for developers to enhance computational efficiency across different GPU architectures.**

***Index Terms*—CUDA, GPGPU, Matrix Multiplication, Parallel Computing**

## I. Introduction

Matrix operations are essential in scientific computing, engineering simulations, and machine learning (ML). Loop unrolling optimizes performance by executing multiple iterations per step, reducing overhead and enhancing parallelism. CUDA, NVIDIA's platform for parallel computing on GPUs leverages GPU parallelism for general-purpose computing, where loop unrolling improves matrix operations by decreasing control instructions and optimizing memory access, resulting in faster computation and better resource efficiency. These optimizations are vital for ML and AI applications that rely heavily on matrix operations, especially in real-time and large-scale deployments.

However, current GPU compilers perform limited automatic loop unrolling, highlighting the need for research into manual unrolling strategies and optimal unroll factors.

This paper examines the impact of loop unrolling on CUDA matrix multiplication, aiming to identify optimal unroll factors and their effect on execution time. By testing various matrix sizes and GPU architectures, the study seeks to demonstrate performance gains from loop unrolling, enhancing GPU programming practices and informing compiler optimizations for applications dependent on matrix multiplication in large-scale or real-time environments.

## II. Related Work

Recent advancements in General-Purpose Graphics Processing Units (GPGPUs) through programming models like CUDA and OpenCL have demonstrated significant performance improvements via architectural and compiler optimizations. Leveraging GPU parallelism, research has shown that loop unrolling—a classical optimization technique—can enhance GPGPU performance by up to 70%, with semi-automatic compile-time methods aiding the determination of optimal unroll factors [1]–[3].

Memory optimization is also pivotal for maximizing GPU performance. Siegel et al. [4] explored memory layout strategies within NVIDIA's CUDA, achieving up to 87× speedup over CPU implementations in applications like the Gravit gravity simulator. Effective memory management in GPU architectures requires innovative data handling and algorithm design to fully exploit their computational capabilities.

Furthermore, Michailidis and Margaritis [5] developed optimized CUDA algorithms for kernel density estimation using shared memory tiles and loop unrolling, resulting in significant performance gains over traditional CPU approaches. These improvements highlight CUDA's versatility in addressing diverse computational problems, emphasizing the importance of both algorithmic and compiler optimizations.

The evolution of CUDA toolkits has also been examined, with Yoshida et al. [6] finding that newer versions generally enhance performance and energy efficiency. However, certain applications may benefit from older versions due to specific factors like loop unrolling efficiency and instruction scheduling.

Additionally, the adoption of high-level programming languages for GPUs has necessitated advanced compilation techniques tailored to GPU parallelism, essential for maximizing GPU potential beyond traditional graphics tasks [7].

This research underscores the importance of hardware-aware and algorithm-aware optimizations in GPGPU computing, pushing the limits of current technologies and paving the way for future innovations in parallel computing.

## III. Methods

The proposed system architecture (Fig. 1) aims to provide comprehensive insights into the benefits of loop unrolling in GPU computing by systematically developing, measuring, and analyzing different kernel implementations.

### A. Solution Architecture

The proposed solution for evaluating the impact of loop unrolling on CUDA matrix multiplication operations involves a multi-phase approach (Fig. 1) designed to assess and optimize performance systematically. The system architecture consists

This is accepted version. Published version DOI: 10.1109/TELFOR63250.2024.10819077

of key components: Data Preparation, Kernel Development, Performance Measurement, and Analysis. Each phase plays a crucial role in the overall workflow.

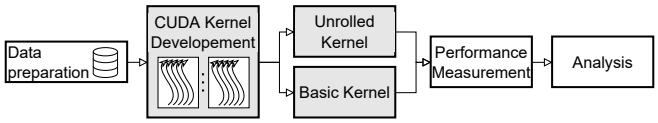


Fig. 1: Solution Architecture for Evaluating Loop Unrolling in CUDA Matrix Multiplication Operations

*1) Data Preparation:* The first step involves creating matrices of various sizes for performance testing to assess the impact of loop unrolling across different data scales. This includes generating random matrices to simulate workloads, initializing them on the CPU, and transferring them to GPU memory using CUDA's (`cudaMalloc` and `cudaMemcpy`) memory management functions.

*2) Kernel Development:* In this phase, we developed CUDA kernels for matrix multiplication with and without loop unrolling, intentionally avoiding shared memory or tiling techniques to isolate the performance impact of loop unrolling.

*a) Basic Kernel:* The basic kernel is a straightforward implementation of matrix multiplication. Each thread computes a single element of the output matrix $C$. The kernel performs the following steps:

1) **Compute Thread Indices**: Each thread calculates its corresponding row and column indices in the output matrix.

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

2) **Initialize Accumulator**: A local variable `sum` is initialized to accumulate the results of the dot product.

```
float sum = 0;
```

3) **Boundary Check**: The kernel includes checks to ensure that threads do not access memory beyond the matrix dimensions.

```
if (row < width && col < width) {
  // Computation
}
```

4) **Compute Dot Product**: A loop iterates over the shared dimension (width of the matrices), performing element-wise multiplication and accumulating the results.

```
  for (int k = 0; k < width; k++) {
    sum += A[row * width + k] * B[k * width + col];
  }
```

5) **Write Result**: The final accumulated value is written to the output matrix $C$.

```
C[row * width + col] = sum;
```

*b) Unrolled Kernels:* We implemented unrolled versions of the kernel with unroll factors of 2, 4, 8, and 16. The unrolled kernels modify the computation loop to reduce loop overhead and increase instruction-level parallelism.

- **Loop Unrolling Mechanism**: The loop increment matches the unroll factor (e.g., `k += 4` for an unroll factor of 4), allowing multiple multiply-add operations per iteration.
- **Conditional Checks**: Conditional statements prevent out-of-bounds access when the matrix size isn't divisible by the unroll factor.
- **Example**: For the unroll factor of 4, the kernel performs up to four multiply-add operations per loop iteration.

```
for (int k = 0; k < width; k += 4) {
  sum += A[row * width + k] * B[k * width + col];
  if (k + 1 < width)
    sum += A[row * width + k + 1] * B[(k + 1) *
        width + col];
  if (k + 2 < width)
    sum += A[row * width + k + 2] * B[(k + 2) *
        width + col];
  if (k + 3 < width)
    sum += A[row * width + k + 3] * B[(k + 3) *
        width + col];
}
```

*3) Performance Measurement:* We used CUDA events (`cudaEvent_t`) to measure each kernel's execution time by capturing start and stop timestamps around kernel launches. This method provides precise GPU timing without requiring external profiling tools.

*4) Analysis:* The final phase involves analyzing the collected performance data by comparing the execution times and metrics of basic and unrolled kernels to assess loop unrolling's effectiveness. Statistical methods and visualizations (e.g., bar charts and line graphs) highlight performance differences, aiming to quantify the impact of loop unrolling and identify the conditions where it provides the most benefit.

### B. Evaluation Methodology

Speedup measures the performance improvement of an optimized algorithm over a baseline. It is defined as:

$$Speedup = \frac{T_{\text{Basic Kernel}}}{T_{\text{Unrolled Kernel}}} \tag{1}$$

A speedup greater than 1 indicates the unrolled kernel is faster, while a value near 1 suggests minimal or no improvement.

### C. Experiments

The primary focus of our experiments is to evaluate the performance of matrix-to-matrix multiplication using CUDA, with and without loop unrolling. Each matrix element is initialized with randomly generated floating-point values of type `float`. Matrix sizes range from $128 \times 128$ to $4096 \times 4096$ to assess performance across varying workloads. The kernels tested include a basic version and four optimized versions with unrolling factors of 2, 4, 8, and 16.

*1) Experimental Setup:* The experiments were conducted on two NVIDIA GPUs: the GeForce RTX 4060 and GTX TITAN X. Each GPU was configured to run the same kernels across different block sizes (8, 16, and 32) to explore the impact of block size on performance. The CUDA Toolkit version 12.4 was used, and the code was compiled with the `nvcc` compiler. The CUDA events (`cudaEvent_t`) API was used to measure the execution time of each kernel with high precision, capturing the start and end times of GPU computations.

The code used for this study, including the full set of experiments, is available on GitHub at https://github.com/vstefkovski/cuda-matrix-benchmark.

*2) Procedure:* For each matrix size, both the basic and unrolled kernels were executed. The unrolled kernels leverage loop unrolling factors of 2, 4, 8, and 16 to minimize loop control overhead and enhance parallelism. CUDA's memory management functions (`cudaMalloc` and `cudaMemcpy`) were used to allocate and transfer matrices between host and device memory.

Each kernel was launched with grid and block configurations tailored to the matrix size and block size. Specifically, the grid dimensions were calculated as:

$$\text{Grid Size} = \left( \frac{N + \text{Block Size} - 1}{\text{Block Size}} \right)$$

where $N$ is the matrix size. This ensures that all matrix elements are processed while handling edge cases when the matrix size is not divisible by the block size.

The program checks for CUDA errors after each kernel launch and prints performance results for successfully executed kernels. This ensures that any misconfiguration or error is immediately identified, and the experiment continues with other configurations. Each kernel's execution time is recorded using CUDA events to ensure reliable measurement.

*3) Performance Metrics:* Performance is measured in terms of execution time (in milliseconds) for each kernel and configuration. The comparison focuses on the speedup achieved by the unrolled kernels over the basic kernel, calculated as:

$$\text{Speedup} = \frac{T_{\text{Basic Kernel}}}{T_{\text{Unrolled Kernel}}}$$

where $T_{\text{Basic Kernel}}$ and $T_{\text{Unrolled Kernel}}$ denote the execution times of the basic and unrolled kernels, respectively. A speedup greater than 1 indicates a performance improvement from loop unrolling.

## IV. Results

The x-axis in all graphs represents the matrix size, and the y-axis represents either execution time in milliseconds or the Speedup level. Each line in the legend denotes a specific configuration: either the basic kernel or an unrolled kernel, run on one of the two GPUs, with a specific unrolling factor.

Fig. 2 and Fig. 3 present the execution time for experiments running on the GeForce RTX 4060 GPU and GTX TITAN X GPU. We observe that as the matrix size increases, the execution time for the unrolled kernels decreases compared to the basic kernel, particularly for unroll factors 8 and 16 (Figure 2). For smaller matrix sizes, the execution times between the basic and unrolled kernels are close. Still, as the matrix size grows, loop unrolling shows significant benefits, with the unroll factor 16 achieving the most substantial reduction in execution time.

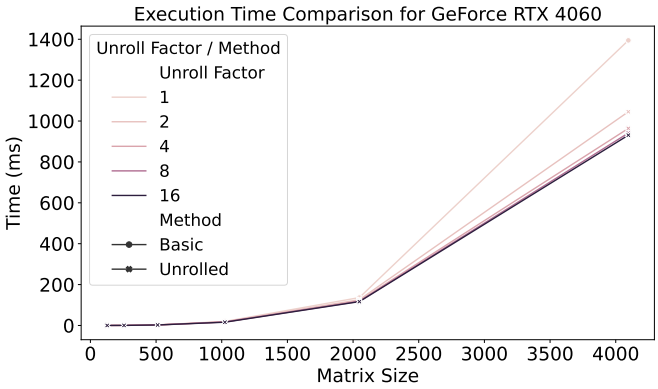


Fig. 2: Execution Time Comparison on GeForce RTX 4060

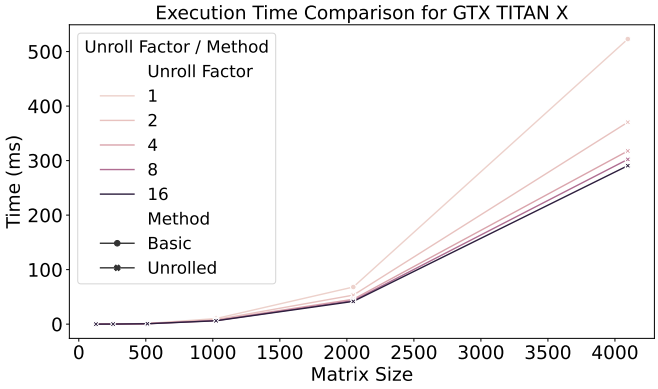


Fig. 3: Execution Time Comparison between Basic and Unrolled Kernels for Different Matrix Sizes on GTX TITAN X

The trends for GTX TITAN X GPU (Figure 3) are similar to those observed on the RTX 4060, where the execution time decreases significantly with larger matrix sizes applying the unrolling. Again, the unroll factors of 8 and 16 offer the most notable reductions in execution time.

## V. Discussion

### A. Speedup

Fig. 4 and Fig. 5 illustrate the Speedup of the unrolled kernels over the basic kernel on the RTX 4060 and GTX TITAN X GPU. The highest speedups are for smaller matrix sizes (128 and 256), with a rapid decrease in Speedup as matrix sizes increase. However, unroll factors 8 and 16 for larger matrices provide consistent performance gains, showing a speedup of approximately 1.5x to 2x over the basic kernel for matrix sizes 4096.

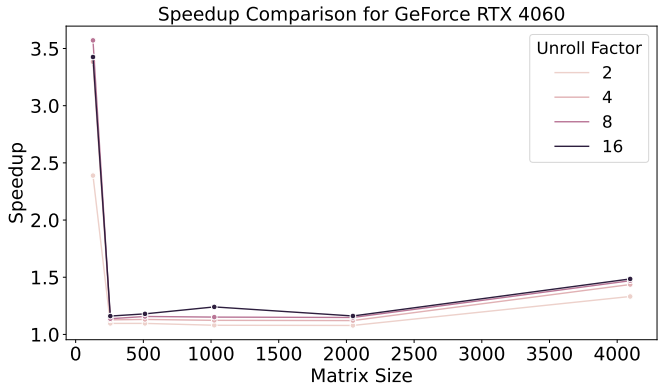


Fig. 4: Speedup Achieved by Unrolled Kernels Compared to the Basic Kernel on GeForce RTX 4060

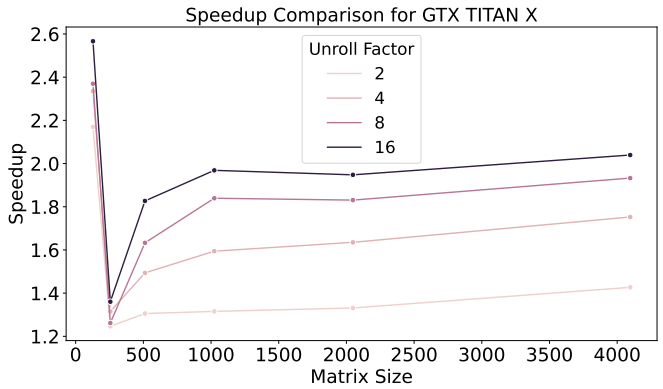


Fig. 5: Speedup Achieved by Unrolled Kernels Compared to the Basic Kernel on GTX TITAN X

The Speedup achieved on the TITAN X (Figure 5) follows a pattern similar to that of the RTX 4060. The most significant speedups occur for small matrix sizes, with factors 8 and 16, and up to 3.5x for larger matrices (2048 and 4096). The performance improvements plateau for larger matrices, with unroll factors 8 and 16 maintaining an advantage over the basic kernel.

These results confirm that loop unrolling can significantly improve the performance of CUDA matrix multiplication, particularly for larger matrices. The unrolling factors of 8 and 16 consistently provide the best balance between execution time reduction and overall efficiency across the RTX 4060 and GTX TITAN X GPUs.

### *B. Comparison to Other Research*

Our findings align with previous research showing that loop unrolling significantly enhances GPGPU performance. Murthy et al. [1] demonstrated that optimal unroll factors can greatly improve performance. Similarly, we found that higher unroll factors reduce execution time but may increase register usage, affecting GPU occupancy.

While Murthy et al. examined various GPGPU applications using a semi-automatic framework, our study focused specifically on matrix multiplication on modern GPUs, achieving up to a 3× speedup with unroll factors of 8 and 16. This highlights the importance of selecting appropriate unroll factors based on the specific application and hardware architecture.

Overall, our results confirm loop unrolling as a critical optimization technique in CUDA programming, while emphasizing the need to balance unrolling with resource constraints to maximize performance across different GPU architectures and problem sizes.

## VI. Conclusion

This study investigated the impact of loop unrolling on the performance of CUDA matrix multiplication kernels across various GPUs and configurations. Results show that loop unrolling significantly enhances GPU matrix operations by reducing execution time and achieving substantial speedups compared to basic implementations.

Applying optimal unroll factors of 8 and 16 with appropriate block sizes led to notable performance gains on NVIDIA GeForce RTX 4060 and GTX TITAN X GPUs, especially for larger matrices. The optimal configuration varies with GPU architecture and problem size, emphasizing the importance of selecting suitable unroll factors and block sizes to maximize performance by minimizing loop overhead and improving instruction-level parallelism.

Future work could further boost performance by combining loop unrolling with other optimizations like shared memory utilization, register tiling, and instruction scheduling. Additionally, exploring the effects of loop unrolling on different matrix operations and newer GPU architectures would provide deeper insights into GPU optimization strategies.

## References

[1] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for gpgpu programs," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–11.

[2] G. S. Murthy, "Optimal loop unrolling for gpgpu programs," Ph.D. dissertation, Ohio State University, 2009.

[3] G. Sreenivasa Murthy, "Optimal loop unrolling for gpgpu programs," Master's thesis, The Ohio State University, 2009.

[4] J. Siegel, J. Ributzka, and X. Li, "Cuda memory optimizations for large data-structures in the gravit simulator," *Journal of Algorithms & Computational Technology*, vol. 5, no. 2, pp. 341–362, 2011.

[5] P. D. Michailidis and K. G. Margaritis, "Accelerating kernel density estimation on the gpu using the cuda framework," *Applied Mathematical Sciences*, vol. 7, no. 30, pp. 1447–1476, 2013.

[6] K. Yoshida, S. Miwa, H. Yamaki, and H. Honda, "Analyzing the impact of cuda versions on gpu applications," *Parallel Computing*, vol. 120, p. 103081, 2024.

[7] G. Chakrabarti, V. Grover, B. Aarts, X. Kong, M. Kudlur, Y. Lin, J. Marathe, M. Murphy, and J.-Z. Wang, "Cuda: Compiling and optimizing for a gpu platform," *Procedia Computer Science*, vol. 9, pp. 1910–1919, 2012.