CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Nuclear Sciences and Physical Engineering

Parallel algorithms for operations with dense matrices

Paralelní algoritmy pro operace s hustými maticemi

Bachelor's Degree Project

Author:	**Alexandr Krastenov**
Supervisor:	**doc. Ing. Tomáš Oberhuber, Ph.D.**
Language advisor:	**Bc. Nathaniel Tobias Patton**
Academic year:	2023/2024

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Krastenov** Jméno: **Alexandr** Osobní číslo: **510804**

Fakulta/ústav: **Fakulta jaderná a fyzikálně inženýrská**

Zadávající katedra/ústav: **Katedra matematiky**

Studijní program: **Aplikovaná informatika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Paralelní algoritmy pro operace s hustými maticemi

Název bakalářské práce anglicky:

Parallel algorithms for operations with dense matrices

Pokyny pro vypracování:

1. Seznamte se se základy programování GPU pomocí CUDA a TNL.
2. Prostudujte současnou implementaci hustých matic v knihovně TNL.
3. Navrhněte vhodné optimalizace kódu zejména pro maticovou transpozici nebo násobení obdélníkových matic včetně využití tenzorových jader na GPU.
4. Implementujte unit testy a benchmarky pro porovnání efektivity s jinými knihovnami.
5. Věnujte pozornost i refaktorování kódu a sepsání dokumentace s příklady použití implementovaných funkcí.

Seznam doporučené literatury:

[1] GPU programming guide - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[2] TNL Users' guide - https://tnl-project.gitlab.io/tnl/UsersGuide.html.
[3] Cui X., Chen Y., Zhang C., Mei H., Auto-tuning Dense Matrix Multiplication for GPGPU with Cache, 2010 IEEE 16th International Conference on Parallel and Distributed Systems, Shanghai, China, 2010, pp. 237-242, 2010.
[4] Nath R., Tomov S., Dongarra J., An Improved Magma Gemm For Fermi Graphics Processing Units, The International Journal of High Performance Computing Applications, vol. 24, no.4, pp.511-515, 2010.
[5] Markidis S., Chien S. W. D. , Laure E., Peng I. B. , Vetter J. S., NVIDIA Tensor Core Programmability, Performance & Precision, 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, Canada, 2018, pp. 522-531.
[6] Feng B., Wang Y., Chen G., Zhang W., Xie Y., Ding Y., EGEMM-TC: Accelerating Scientific Computing on Tensor Cores with Extended Precision, PPoPP '21: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 278-291, 2021.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Tomáš Oberhuber, Ph.D. katedra matematiky FJFI

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **31.10.2023** Termín odevzdání bakalářské práce: **05.08.2024**

Platnost zadání bakalářské práce: **30.09.2025**

[illegible]	[illegible]	[illegible]
doc. Ing. Tomáš Oberhuber, Ph.D. podpis vedoucí(ho) práce	prof. Ing. Zuzana Masáková, Ph.D. podpis vedoucí(ho) ústavu/katedry	doc. Ing. Václav Čuba, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

28. 11. 2023	[illegible]
Datum převzetí zadání	Podpis studenta

Acknowledgment:
I would like to thank doc. Ing. Tomáš Oberhuber, Ph.D. for his expert guidance and express my gratitude to Bc. Nathaniel Tobias Patton for his language assistance. Additionally, I would like to extend my heartfelt thanks to my family, my partner, and my colleagues for their unwavering support and encouragement during these challenging times.

Author's declaration:
I declare that this Bachelor's Degree Project is entirely my own work and I have listed all the used sources in the bibliography.

Prague, August 5, 2024 Alexandr Krastenov

Název práce:

Paralelní algoritmy pro operace s hustými maticemi

Autor: Alexandr Krastenov

Obor: Aplikovaná informatika

Druh práce: Bakalářská práce

Vedoucí práce: doc. Ing. Tomáš Oberhuber, Ph.D., Katedra softwarového inženýrství, Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze

Abstrakt: Tato bakalářská práce je věnována paralelním algoritmům pro operace s hustými maticemi se zaměřením na násobení a transpozici. Obě operace jsou široce využívány ve vědeckých výpočtech a vyžadují efektivní paralelizační strategie ke zkrácení doby provádění. Tato práce poskytuje komplexní studii hlavních optimalizačních technik a metod, později implementovaných v široké řadě optimalizovaných kernelu pro násobení a transpozici na místě i mimo místo. S využitím CUDA C++ a knihovny TNL byly všechny vyvinuté kernely testovány, aby byla zajištěna správnost, a porovnány se zavedenými GPU NVIDIA knihovnami, aby se vyhodnotil jejich výkon.

Klíčová slova: CUDA C++, husté matice, doba provádění, transpozice na místě, násobení, NVIDIA GPU knihovny, optimalizační techniky, transpozice mimo místo, paralelní algoritmy, paralelizační strategie, vědecké výpočty, knihovna TNL

Title:

Parallel algorithms for operations with dense matrices

Author: Alexandr Krastenov

Abstract: This bachelor's degree project is dedicated to parallel algorithms for operations with dense matrices, focusing on multiplication and transposition. Both operations are widely utilized in scientific computations and require efficient parallelization strategies to reduce the execution time. This work provides a comprehensive study of main optimization techniques and methods, later implemented in a broad range of optimized kernels for multiplication and both in-place and out-of-place transposition. Utilizing CUDA C++ and the TNL library, all developed kernels were tested to ensure correctness and benchmarked against established NVIDIA GPU libraries to evaluate their performance.

Key words: CUDA C++, dense matrices, execution time, in-place transposition, multiplication, NVIDIA GPU libraries, optimization techniques, out-of-place transposition, parallel algorithms, parallelization strategies, scientific computations, TNL library

Contents

List of Tables

List of Figures

Introduction

Parallel algorithms for operations with dense matrices are at the forefront of scientific computing, allowing fast handling of massive datasets and complex numerical calculations. They are the cornerstone of various algorithms with applications, ranging from machine learning and data analytics to engineering simulations and theoretical physics. Dense matrices, characterized by their non-sparse nature, where most of the elements are non-zero, require optimized computational approaches and parallelization strategies to utilize the full potential of modern hardware architectures.

The main goal of this thesis is to apply GPU-suitable algorithm optimizations and produce a workable solution that may be added to the TNL library. The implementation is in CUDA C++, utilizing the TNL structures and optimizing dense matrix operations, focusing primarily on matrix multiplication and transposition.

This thesis is structured as follows: Chapter 1 delves into the theory behind GPU programming and the tools used in this project. Chapter 2 discusses comprehensive optimization techniques, leveraging a wide range of methods and tools for high-performance computing. Chapter 3 is dedicated to the implementation, detailing the logic behind the selection and application of optimization techniques for dense matrix operations. Finally, Chapter 4 concludes with UnitTests and benchmarks measured on a professional-grade GPUs, followed by a comparison of the implementations against alternative NVIDIA libraries.

Chapter 1

Theory

1.1 GPU

The Graphics Processing Unit (GPU) is a specialized processor originally designed to accelerate graphics rendering and to compute tasks together with the Central Processing Unit (CPU). GPUs and CPUs are very similar computational engines. However, they are built for different purposes and have different architectures [3].

The CPU, also known as the brain of the computer, is the main part that handles most of the internal processing. With just one or a few threads on a single processing unit, CPUs are highly optimized to operate at high frequencies, making them perfect for general-purpose computing tasks like running operating systems, programs, and user interfaces.

On the other hand, GPUs are extremely effective for tasks that can be parallelized because they can execute thousands of basic operations at once. GPUs are especially useful for complicated computations in domains such as scientific research, machine learning, and simulations because of their capacity for parallel processing.

Most applications consist of sequential and parallel components, so to maximize performance, systems are designed with a combination of CPU and GPU [4].

1.1.1 CUDA

To take advantage of the parallel computing capacity of NVIDIA GPUs, NVIDIA introduced Compute Unified Device Architecture (CUDA), a revolutionary computer platform and programming model. It enables programmers to utilize NVIDIA GPUs using C++ and other high-level programming languages [4].

CUDA's architecture is designed around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a CUDA program executes, it runs in parallel across a large number of threads - each thread computing one element of a matrix, for instance.

In addition to a software environment, the CUDA toolkit includes additional capabilities, such as debugging and profiling applications, to assist programmers in creating effective GPU applications.

1.1.2 Thread Hierarchy

Threads are the fundamental unit of execution in parallel computing with CUDA. To improve the efficiency and control of concurrent processing, CUDA organizes threads into bigger groups. CUDA introduces the concept of warps, which are collections of 32 threads executing concurrently. These threads begin their execution simultaneously at the same program address, offering a base for synchronized operation within the GPU's architecture.

Kernels are CUDA C++ extended functions that execute simultaneously through *N* different CUDA threads. They are defined using the *__global__* declaration specifier and the number of CUDA threads that execute that kernel for a given kernel launch is specified using a new <<< ... >>> execution configuration syntax [4].

Within a kernel, each executing thread is assigned a distinct ID, accessible through the `ThreadIdx` built-in variable. Threads are organized into blocks, which can be indexed in one, two, or three dimensions, offering flexible structuring options for parallel computation. This multi-dimensional indexing allows threads to be efficiently managed and coordinated.

Threads are further organized into grids, which are composed of blocks. These blocks can be uniquely identified by the `blockIdx` variable within a kernel. The CUDA framework requires thread blocks to operate independently for efficient parallel processing, as it allows blocks to be executed in any sequence.

This hierarchical organization enables precise control over the execution of parallel tasks, optimizing the performance and scalability of applications that leverage the power of GPU computing.

Threads within blocks can share data through shared memory and coordinate memory access by synchronizing execution. Synchronization points in the kernel can be specified by calling the *__syncthreads*() function, which acts as a barrier at which all threads in the block must wait before any is allowed to proceed [4, 5].

1.1.3 Synchronization

The synchronization method is provided by the CUDA API and is necessary for coordination thread execution within a block. When encountered in a kernel, this method blocks all threads at the calling point until each of them reaches and executes `__syncthreads()`, ensuring that all threads have completed the same amount of work at a given point before moving to the next step.

In matrix multiplication, it is important that all threads have loaded the necessary elements from the input matrices into shared memory and completed their computations before writing back to the result matrix. Similarly, in matrix transposition, synchronization is used to prevent data hazards when threads read from and write to shared memory. It is used to ensure that all threads in the block have reached the same execution point and that data integrity is maintained throughout the computation.

1.1.4 Memory

The CUDA runtime implements functions for memory allocation, deallocation, and data transmission between the host (CPU) and the device (GPU). Moreover, CUDA provides a number of memory spaces, each designed to serve a specific function and optimize certain elements of memory access and data storage.

Typically, `cudaMalloc()` is used to allocate memory and `cudaFree()` is used to free it. A method for transferring data between host and device memory is `cudaMemcpy()`. During execution, CUDA threads have access to data from several memory regions:

Global memory is the most extensive type of memory available to all threads and is accessible across all blocks. It is located off-chip, which means that it has high latency and provides substantial space for data storage. This type of memory is typically used to store large datasets that do not fit faster types of memory.

Shared memory is significantly faster than global memory as it is located on the chip. This type of memory offers much lower latency and higher bandwidth and is only visible to threads within the same block. It is commonly used as a user-managed cache, allowing threads within a block to quickly access and reuse data. Shared memory is declared in kernel code using the `__shared__` specifier.

Registers represent the fastest type of memory available in CUDA. They are also on the chip and private to each thread, making them ideal for frequently accessed data or variables requiring the fastest access speeds, such as loop counters or temporary variables in calculations.

Local memory serves as storage for automatic variables that do not fit into registers. Though physically situated in global memory, it behaves as a private space for individual threads. This type of memory is typically used to store array indices or complex data structures that cannot be efficiently stored in registers.

This strategic use of memory types helps optimize performance in CUDA applications by aligning memory usage with the needs of each specific computation task.

1.1.5 Tensor Cores

Tensor Cores are specialized programmable processing units within GPUs designed to accelerate deep learning and other intensive arithmetic computations. With only minimal space and power requirements, Tensor Cores and the data channels that go with them are specially designed to boost floating-point calculation throughput.

Each Tensor Core provides a $4 \times 4 \times 4$ matrix processing array that can execute the operation `D` = `A` × `B` + `C` , where `A`, `B`, `C`, and `D` are 4×4 matrices. This design allows multiple Tensor Cores to be used simultaneously within a single GPU warp, thereby significantly increasing the computation capacity.

A full warp of execution uses several Tensor Cores concurrently throughout program execution. Tensor cores can process larger $16 \times 16 \times 16$ matrix operations due to threads within a warp. These operations are made available by CUDA through the CUDA C++ Warp-level Matrix Multiply Accumulate (WMMA) API as warp-level matrix operations. In order to effectively use Tensor Cores in CUDA C++ programs, these C++ interfaces offer specialized matrix load, matrix multiply and accumulate, and matrix store operations [7].

1.1.6 HIP

Heterogeneous-compute Interface for Portability (HIP) is a programming language developed by AMD to provide a portable and efficient computing interface for developing applications that may operate on a variety of hardware. It provides a C++ runtime API and a kernel language, enabling developers to design high-performance applications that can operate on both AMD and NVIDIA GPUs.

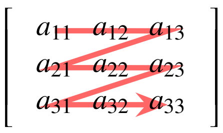
The primary purpose of HIP is to make it easier to construct applications on a variety of platforms by offering a standard programming model. This allows developers to easily convert CUDA code to portable C++ code that can operate on a variety of GPU architectures.

One of the important characteristics of HIP is its ability to automatically convert CUDA code to HIP code, allowing developers to retain a single codebase [6].

1.2 Dense Matrices

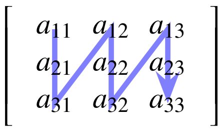
A dense matrix is a matrix with mostly non-zero elements. Such matrices are typically represented and stored in two-dimensional arrays, which allows direct access to any element a_{ij} given its row i and column j [1]. This characteristic makes them suitable for a wide range of mathematical and computational operations, including the algebraic operations and transformations discussed in this section.

Dense matrices are stored in row major or column major layouts. In row major order, the elements of the matrix are stored sequentially row by row. This means that the elements in a single row are located next to each other in memory. For example, in a 3×3 matrix, the storage order in the memory would be



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{21} & a_{22} & a_{23} & a_{31} & a_{32} & a_{33} \end{pmatrix}$$

In contrast, column major order stores matrix elements sequentially by columns. Thus, elements in a single column are contiguous in memory. Using the same 3×3 matrix example, the storage order would be



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{pmatrix} a_{11} & a_{21} & a_{31} & a_{12} & a_{22} & a_{32} & a_{13} & a_{23} & a_{33} \end{pmatrix}$$

Furthermore, the storage layout directly impacts the performance when interfacing with software libraries. For example, row-major order is the default storage used by programming languages such as C and C++, whereas column-major order is used by CUDA, Fortran, and libraries optimized for scientific computing. Dense matrix operations can often be parallelized to take advantage of multi-core and multi-threaded computing environments. The storage order should be considered to minimize memory contention and maximize the performance gains from parallel execution.

1.2.1 Dense Matrix Multiplication

Let A be an $m \times n$ matrix and B be an $n \times p$ matrix, where $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$. Matrix multiplication is an operation in which the matrix A is multiplied by the matrix B, resulting in a matrix $m \times p$ C, where $C \in \mathbb{R}^{m \times p}$. Each element c_{ij} of C is the sum of the products of the corresponding elements from the i-th row of matrix A and the j-th column of matrix B. Mathematically, the operation is defined as

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in} + b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}, \tag{1.1}$$

where $1 \le i \le m$ and $1 \le j \le p$. The element c_{ij} is computed by summing the products of the corresponding elements from the row of A and the column of B.

$$\underset{A}{\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}} \times \underset{B}{\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix}} = \underset{C}{\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}}$$

In the visualization above, matrices A, B, and C represent the input and output of the matrix multiplication operation. The highlighted cells illustrate the calculation of element c_{11} in matrix C, specifically showing how c_{11} is the sum of products of the elements from the first row of A and the first column of B. Similarly, each element of C is calculated using the corresponding row of A and column of B.

In the domain of linear algebra, dense matrix multiplication is a fundamental operation with wide-ranging applications in computer science, engineering, and physics. Performance on a broad spectrum of computer activities is directly affected by the efficiency of this process. The ability to split and execute matrix multiplication operations concurrently is a feature that parallel computing systems take advantage of. Computation time can be significantly decreased by dividing the work among several processors, either in a distributed computing environment or on a single computer with multi-threading. In the upcoming sections, we will explore various optimization strategies and their implementations in parallel computing environments.

1.2.2 Dense Matrix Transposition

Let A be an $m \times n$ matrix where $A \in \mathbb{R}^{m \times n}$. Matrix transposition is an operation that flips matrix A over its diagonal, resulting in a transpose matrix A^T which is an $n \times m$ matrix, where $A^T \in \mathbb{R}^{n \times m}$. The element $(A^T)_{ij}$ of the transposed matrix is equal to the element a_{ji} of the original matrix, i.e., the rows and columns are swapped. The operation is mathematically defined as

$$(A^T)_{ij} = a_{ji}, \tag{1.2}$$

where $1 \leq i \leq n$ and $1 \leq j \leq m$. This defines that each element c_{ij} in the transpose matrix A^T is derived from the element a_{ji} in the original matrix A.

Consider the transposition of a 4×2 matrix A:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \quad \rightarrow \quad A^T = \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \end{bmatrix}.$$

The visual diagram above illustrates the transposition in which each element a_{ij} of matrix A is repositioned to the corresponding location a_{ji} in A^T.

Although transposing a matrix is theoretically simple, the computational consequences can differ based on the matrix's size and the system architecture. It is possible to parallelize the transposition of a matrix so that several processing elements can carry out the computation concurrently. We will explore matrix transposition optimization strategies inside a parallel computing environment in the sections that follow.

1.2.2.1 Transposition Methods

The introduction of GPU computing has made the creation of effective transposing matrix algorithms on these parallel platforms necessary. There are two primary transposition methods: out-of-place and in-place.

Out-of-place transposition involves using additional memory to hold the transposed matrix. This method can achieve high performance by exploiting the substantial memory bandwidth of GPUs. However, it comes at the cost of doubling the memory requirement, which can be restrictive given the limited on-board memory of GPUs.

In contrast, in-place transposition operates directly on the original matrix without using additional memory. This method is preferable because GPUs have limited memory. In-place transposition methods are categorized into two primary types: those that handle square matrices and those that handle non-square matrices.

For square matrices, in-place transposition can be implemented relatively straightforwardly by swapping elements across the diagonal. This technique allows for efficient memory usage and can leverage the parallel processing capabilities of GPUs to achieve high performance.

However, for non-square matrices, in-place transposition is more challenging because swapping elements across the diagonal leads to out-of-bounds memory accesses. One approach is to reduce the non-square matrix into the largest possible square submatrix within the original. This allows for efficient transposition using GPU parallelism within the square submatrix. After the transposition, the matrix is padded back to its original non-square dimensions. This reduction can be achieved through blocking or tiling techniques that allow the non-square matrix to be divided into smaller square tiles that can be transposed individually.

1.3 TNL

Teplate Numerical Library (TNL) is an open-source library that offers a collection of building blocks that facilitate the development of efficient numerical solvers and High-performance computing (HPC) algorithms. It supports multicore CPUs, GPUs, and distributed systems with a unified interface for modern hardware architectures [11].

1.3.1 Lambda Functions

Programming for GPUs often involves writing kernels. TNL offers skeletons or patterns of such algorithms and combines them with user-defined lambda functions. This approach simplifies GPU programming and provides several advantages.

- Implementing lambda functions is easier compared to writing GPU kernels.
- The implemented code works on both CPU and GPU, allowing developers to write a single codebase for both hardware architectures.
- Debugging can be done on the CPU first, with minimal or no changes required to run it on the GPU.

Managing distinct memory architectures in high-performance computing requires special attention to address spaces. TNL simplifies this by offering a unified interface that abstracts the complexities of data transfer between CPU and GPU memory. These devices are defined as follows

```
TNL::Devices::Host
TNL::Devices::Cuda
TNL::Devices::Hip
```

When using `TNL::Devices::Host`, the computation occurs on the CPU. This process can be sequential or parallel, often utilizing multiple CPU threads through OpenMP for enhanced performance. For computations on NVIDIA GPUs, the `TNL::Devices::Cuda` is employed. This device uses the `__cuda_callable__` annotation to compile the lambda function such that it is executable on both the host and the CUDA device. This dual compatibility ensures that the same codebase can efficiently operate across different platforms. Similarly, the `TNL::Devices::Hip` targets AMD GPUs using HIP technology, which mirrors NVIDIA's CUDA in allowing cross-platform code functionality.

TNL automatically handles the differences in memory allocation and access between these address spaces, minimizing data transfer latency through efficient use of the PCI-Express bus. Data structures also provide views to encapsulate references, facilitating efficient memory access across devices. By leveraging these abstractions, developers can focus on writing high-performance code without worrying about device-specific memory intricacies [11].

1.3.2 Data Structures

1.3.2.1 Arrays and Vectors

An array in TNL is a templated class defined in the TNL::Containers namespace with template parameters:

- **Value**: The type of data stored in the array.
- **Device**: The device where the array is allocated.
- **Index**: The type used for indexing of the array elements.
- **Allocator**: The type of allocator used for the allocation and deallocation of memory used by the array.

TNL defines three types of arrays: common arrays with dynamic allocation, static arrays allocated on the stack, and distributed arrays with dynamic allocation. They are unable to share data with each other or data assigned elsewhere. This can be accomplished using the ArrayView structure, which has comparable semantics as Array but does not manage data allocation and deallocation. Array views are useful for encapsulating data allocated elsewhere and dividing huge arrays into sub-arrays. The technique of encapsulating external data in a view is known as binding.

Dynamic vectors in TNL are templated classes defined in the `TNL::Containers` namespace. They have three template parameters:

- **Real**: The type of data stored in the vector.
- **Device**: The device where the vector is allocated.
- **Index**: The type used for indexing of the vector elements.
- **Allocator**: The type of the allocator used for the allocation and deallocation of memory used by the vector.

They provide basic linear algebra operations with a focus on Blas level 1 procedures. TNL is intended to be simple to implement, and in some situations faster than Blas or CuBlas implementations. Vectors, unlike arrays, require the `Real` type to be numeric or a type with provided fundamental algebraic operations. The type of algebraic operations necessary depends on the vector operations that the user will invoke. Vectors are derived from arrays and hence inherit all of the array's methods.

1.3.2.2 Dense Matrices

Dense matrix (TNL::Matrices::DenseMatrix) is a templated class using five template parameters:

- **Real**: The type of matrix elements.
- **Device**: The device where the matrix is allocated.
- **Index**: The type to be used for the indexing of the matrix elements.

- **ElementsOrganization**: The organization of the elements of the matrix in memory.
- **RealAllocator**: The type of allocator used for the allocation and deallocation of memory used by the matrix.

This class exposes some public methods that enable the reading, editing, and inspection of matrix elements. This functionality is important in computational environments where matrices may reside across different memory spaces.

Accessing elements in `DenseMatrix` is facilitated through several methods and operator overloads, each tailored for specific use cases

- **`getElement` and `setElement`**: These methods provide a safe way to access and modify elements of a matrix, regardless of where the matrix data is stored. For example, if a matrix is stored on a GPU, these methods ensure that the element is correctly accessed or modified, even when called from a CPU. This is achieved by handling necessary data transfers between the CPU and GPU, ensuring consistency and correctness. The `getElement(row, column)` method returns a copy of the matrix element at the specified indices, preventing any modification of the actual data in the matrix. In contrast, the `setElement(row, column, value)` method modifies the element at the specified indices directly within the matrix.
- **Operators `()` and `[]`**: These operators are overloaded to provide more intuitive and direct access to matrix elements. The `()` operator allows users to access or modify elements directly using row and column indices, such as in `matrix(row, column)`. This direct access is efficient but must be used within the correct execution context (i.e., CPU code for CPU matrices and GPU kernels for GPU matrices) to avoid errors. Unlike `getElement` and `setElement`, which manage cross-device execution internally, these operators do not handle such transitions and are meant for use in environments where the execution context is known and consistent.

To facilitate the management of matrix structure, `DenseMatrix` offers methods like `getRows` and `getColumns`, which provide direct access to the matrix's dimensional properties. These methods are essential for algorithms that need to adapt to the matrix size dynamically or when initializing structures based on matrix dimensions.

The functionality of `DenseMatrix` is further enhanced by the provision of views, which allow for controlled manipulation and access:

- **`getView`**: This method returns a modifiable view of the matrix, enabling users to perform updates or manipulations on the matrix data while preserving the underlying data structure's integrity. This is particularly useful for operations that require repeated modifications to matrix elements.
- **`getConstView`**: In contrast, the `getConstView` method provides a read-only view of the matrix, ensuring data consistency and protecting the data against unintended modifications. This method is crucial for operations that require data integrity, such as during multi-threaded or distributed computations where data race conditions might otherwise lead to data corruption.

1.3.3 Norms

1.3.3.1 L2 Norm

The L2 norm, also known as the Euclidean norm, computes the Euclidean distance from the origin to a point in space, returning a positive scalar of a vector. This norm is defined as

$$\text{L2Norm} = \sqrt{a_1^2 + a_2^2 + a_3^2}, \tag{1.3}$$

where it represents the square root of the sum of squared vector components [12].

The implementation of the L2 norm within TNL is defined as follows:

```
auto l2Norm(const ET1& a) {
   using TNL::sqrt;
   return sqrt(sum(sqr(a)));
}
```

1.3.3.2 Max Norm

Contrastingly, the Max Norm, or also called the infinity norm, is determined by defining the maximum absolute value among the vector's components. This norm is mathematically expressed as

$$\text{MaxNorm} = \max(|a_1|, |a_2|, |a_3|), \tag{1.4}$$

where the maximal component's magnitude is represented without considering its direction.

The Max Norm's implementation within TNL is

```
auto maxNorm(const ET1& a) {
   return max(abs(a));
}
```

Chapter 2

Optimization Techniques

This chapter is dedicated to dissecting a range of optimization techniques that have been proposed in various articles to enhance the performance and efficiency of matrix multiplication and transposition on GPU architectures.

2.1 Global Memory Coalescing

Memory access patterns in CUDA kernels depend on the arrangement of threads. The positioning of threads within a block and blocks within a grid is a determining factor in how effectively a kernel accesses global memory.

Coalesced memory access occurs when threads of a single warp access consecutive memory locations. This efficient pattern allows the GPU to consolidate these requests into a fewer number of memory transactions, maximizing memory bandwidth utilization.

In contrast, non-coalesced memory access happens when threads in the same warp access non-sequential memory locations. This results in multiple, isolated memory transactions, leading to increased latency and decreased bandwidth efficiency, which can severely impact performance.

Typically, in matrix multiplication, threads are indexed in a CUDA block in a way that each thread is responsible for computing one element of the result matrix. The thread indices `threadIdx.x, threadIdx.y` within the block and the block indices `blockIdx.x, blockIdx.y` within the grid uniquely determine the position of an element in the output matrix.

In the naive implementation from the article [13], two threads in the same block, with Thread IDs (0, 0) and (0, 1), would load the same column of the second matrix but different rows of the first one. Specifically, Thread (0,0) accesses `A[0][0]` and `B[0][0]`, while Thread (0,1) accesses `A[0][1]` and `B[0][0]`. This access pattern leads to non-coalesced memory accesses, as threads of the same warp access memory that is not sequential, causing multiple memory transactions where fewer could suffice [13].

```
// Pseudocode of the Naive Kernel to demonstrate the memory access pattern
__global__ void naiveKernel(float *A, float *B, float *C, int N) {
    const int row = blockIdx.y * blockDim.y + threadIdx.y;
    const int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N) {
        float value = 0.0;
        for (int k = 0; k < N; ++k) {
            value += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = value;
    }
}
```

The thread configuration can be modified so that threads within the same warp access consecutive memory addresses, thus reducing the required memory transactions and increasing the bandwidth utilization [13].

```
// Pseudocode of the Optimized kernel with improved memory coalescing
__global__ void optimizedKernel(float *A, float *B, float *C, int N) {
    const int row = blockIdx.x * blockDim.x + (threadIdx.x / blockDim.y);
    const int col = blockIdx.y * blockDim.y + (threadIdx.x % blockDim.y);

    if (row < N && col < N) {
        float value = 0.0;
        for (int k = 0; k < N; k++) {
            value += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = value;
    }
}
```

In this optimized version, the thread indices are recalculated to ensure that Thread ID (0,0) accesses `A[0][0]` and `B[0][0]`, and Thread (0,1) accesses `A[0][1]` and `B[1][0]`.

By accessing consecutive elements of matrix `A` in terms of the row index and matrix `B` in terms of the column index, the memory transactions are significantly reduced. The optimized kernel achieves better memory coalescing, resulting in improved performance.

2.2 Shared Memory Cache-Blocking

When multiplying two matrices, parts of them are usually loaded from global memory into shared memory. This section highlights the rationale behind using `1D` and `2D` shared memory arrays in CUDA specifically for linear algebra operations.

Linear data structures like vector addition use 1D arrays where element-wise computations are done. They are simple to implement and most effective for linear data partitioning as well as sequential access by threads within a block.

The following code snippet demonstrates a basic vector addition (`C = A + B`) using shared memory `sData`.

```
__global__ void vectorAdd(const float* A, const float* B, float* C, int numElements) {
    extern __shared__ float sData[]; // Shared memory for vector A
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements) {
        sData[threadIdx.x] = A[i]; // Load A[i] into shared memory
        __syncthreads(); // Ensure all threads have loaded their elements
        C[i] = sData[threadIdx.x] + B[i]; // Perform addition using A from shared memory
    }
}
```

This example leverages shared memory for fast, repeated access to elements of vector A, reducing the number of global memory accesses and thereby enhancing the kernel's performance. The synchronization barrier ensures that all threads in the block have loaded their data into shared memory before proceeding to computation.

Each thread loads one element from vector A into shared memory. After synchronizing the threads within the block to ensure that all elements are loaded, each thread computes the sum of its corresponding elements in A and B, then writes the result to vector C. This method leverages fast access to data stored in shared memory, reducing global memory accesses.

2D arrays, on the other hand, are ideal for more complex data structures such as those involving matrices. This allows for more natural access patterns in two dimensions, which is important when loading elements in operations like matrix multiplication and transposition.

In the shared memory cache-blocking technique, data chunks are loaded from global memory into shared memory for processing. This reduces global memory accesses, leveraging shared memory's high speed to perform more operations on cached data before fetching new data.

Below is an example of a tiled matrix multiplication.

```
// Shared Memory Tiled Matrix Multiplication
__global__ void matrixMulShared(float *A, float *B, float *C, int N) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
    int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    float tmp = 0.0;
    for (int bkIdx = 0; bkIdx < N; bkIdx += BLOCK_SIZE) {
        As[threadIdx.y][threadIdx.x] = A[row * N + bkIdx + threadIdx.x];
        Bs[threadIdx.y][threadIdx.x] = B[(bkIdx + threadIdx.y) * N + col];
        __syncthreads();

        for (int k = 0; k < BLOCK_SIZE; k++) {
            tmp += As[threadIdx.y][k] * Bs[k][threadIdx.x];
        }
        __syncthreads();
    }
    if (row < N && col < N) {
        C[row * N + col] = tmp;
    }
}
```

In this approach, shared memory arrays As and Bs are strategically used to load tiles of matrices A and B to minimize global memory accesses. Each thread block collaboratively computes a portion of the output matrix C, maximizing data reuse in shared memory. This minimization of memory traffic between global and shared memory significantly boosts the performance of the matrix multiplication.

This effective utilization of shared memory for caching data demonstrates a significant reduction in latency and an increase in throughput for matrix computations in CUDA [13].

2.3 Padding in Shared Memory Arrays

Padding is a strategic modification applied to data structures in GPU programming to enhance memory access efficiency and prevent performance bottlenecks caused by bank conflicts. This section explores the use of padding in shared memory arrays, its rationale, and its benefits.

In CUDA programming, shared memory is organized into multiple memory banks. During parallel execution, multiple threads may attempt to access data from the same memory bank simultaneously. Since each memory bank can only service one access at a time, simultaneous requests lead to what is known as a bank conflict. This conflict forces accesses to be serialized, thereby degrading the performance by eliminating the benefits of parallel execution.

To mitigate bank conflicts, padding is introduced in the memory allocation of shared arrays. Padding involves adding an extra element to the dimensions of shared memory arrays, typically in scenarios where data structures are accessed by multiple threads. For example, in the context of matrix multiplication kernels, consider the shared memory allocation for matrix tiles:

```
// Two-dimensional shared memory arrays with added padding
__shared__ float tileA[tileDim][tileDim + 1];
__shared__ float tileB[tileDim][tileDim + 1];
```

The additional element in the second dimension $tileDim + 1$ ensures that each row of the tile is offset in memory. This offset prevents threads of a warp from accessing elements that fall into the same memory bank, thereby avoiding bank conflicts.

The main advantage of implementing padding in shared memory arrays is the significant reduction in bank conflicts, which leads to:

- Increased throughput of memory operations due to fewer delays caused by serialized accesses.
- Enhanced overall computational efficiency, particularly in kernels that heavily rely on shared memory, such as those used in matrix multiplication.
- Improved execution speed and smoother performance in applications that are intensive in memory bandwidth.

In summary, padding is a critical optimization technique in GPU programming, especially for applications that require high levels of parallelism and make extensive use of shared memory.

2.4 Loop Unrolling

Loop Unrolling is an optimization technique used to enhance performance by reducing the number of conditions. In CUDA, this can be beneficial due to the parallel nature of the computation. The syntax `#pragma unroll` explicitly instructs the compiler to unroll loops, which can lead to significant speed improvements in certain situations. It can be used in the following ways.

- `#pragma unroll`: Completely unrolls the loop, should be used when the loop count is fixed and small.
- `#pragma unroll N`: Unrolls the loop N times, where N is a positive integer.
- `#pragma unroll 1`: Prevents unrolling of the loop.

In the following example, the `#pragma unroll(4)` directive unrolls the loop body four times to reduce loop overhead and optimize performance.

```
// Using pragma unroll 4 for optimization
#pragma unroll 4
for (int i = 0; i < 4; ++i) {
    C[idx * 4 + i] = A[idx * 4 + i] + B[idx * 4 + i];
}
```

In this example, `#pragma unroll(4)` results in.

```
i = 0;
if (i > N - 4) goto remainder;
for (; i < N - 4; i += 4) {
    C[i] = A[i] + B[i];
    C[i+1] = A[i+1] + B[i+1];
    C[i+2] = A[i+2] + B[i+2];
    C[i+3] = A[i+3] + B[i+3];
}
if (i < N) {
    remainder:
    for (; i < N; i++) {
        C[i] = A[i] + B[i];
    }
}
```

By unrolling the loop, the compiler reduces the number of iterations and loop overhead. This increases computational efficiency for vector addition, as larger chunks of data are processed in each loop iteration.

2.5 Blocktiling and Warptiling

The concept of tiling in CUDA programming refers to the practice of partitioning data and computation into smaller chunks or tiles that can be processed more efficiently by the GPU. This technique is widely used to optimize memory usage and computational performance, especially for algorithms that operate on large datasets.

Blocktiling organizes computations into blocks, each executed independently on a stream multiprocessor (SM) within the GPU. The blocks are the fundamental units of computation, and by assigning each block to an SM, different blocks can process data concurrently. This model maximizes the parallel execution capabilities of the GPU.

In the matrix multiplication, this approach involves storing sub-matrix tiles in shared memory, which significantly reduces the demand for global memory accesses. Each computational block processes its assigned tile independently, which allows multiple blocks to execute in parallel across various SMs. This setup not only minimizes global memory access but also enhances computational efficiency by enabling the reuse of data loaded into shared memory by the threads within each block.

Warptiling optimizes computations at the warp level. By structuring the computations to align with warp size, warptiling reduces shared memory conflicts and enhances register usage. The optimized kernel below demonstrates this concept.

```
__global__ void MatrixMulWarp(float *A, float *B, float *C, int N) {
    __shared__ float As[WARP_SIZE][WARP_SIZE];
    __shared__ float Bs[WARP_SIZE][WARP_SIZE];

    int warpRow = threadIdx.y, warpCol = threadIdx.x;

    As[warpRow][warpCol] = A[warpRow * WARP_SIZE];
    Bs[warpRow][warpCol] = B[warpCol * WARP_SIZE];
    __syncthreads();

    float CValue = 0;
    for (int k = 0; k < WARP_SIZE; ++k) {
        CValue += As[warpRow][k] * Bs[k][warpCol];
    }
    C[warpRow] = CValue;
}
```

This kernel focuses on data reuse within the warp and minimizes synchronization overhead by structuring computations to align with warp execution. Each thread within a warp loads a single element of matrices `A` and `B` into the shared memory arrays `As` and `Bs`, respectively. The threads use their `threadIdx.x` and `threadIdx.y` to determine their column and row positions in the warp. After synchronizing the threads with `__syncthreads()`, each thread computes a partial sum of the dot product for its respective element in the resulting matrix `C`, iterating over the shared warp-size tiles.

Combining blocktiling and warptiling allows CUDA developers to harness GPU parallelism at multiple hierarchical levels. This ensures efficient allocation and usage of GPU resources, achieving high performance in computationally intensive tasks like matrix multiplication, approaching near-cuBLAS performance levels [13].

2.6 Register Blocking

Register Blocking is an optimization technique that utilizes the GPU's registers, the fastest accessible form of memory available to threads. By storing frequently accessed data directly in registers during computation, this method minimizes memory latency and maximizes computational throughput. Registers provide faster access than shared memory, allowing for faster computations and reduced reliance on slower memory hierarchies.

The article [14] describes an improved matrix multiplication algorithm for Fermi GPUs that employs Register Blocking. This algorithm intricately divides the computation across a grid of thread blocks (`TBs`), each dedicated to processing a 64×64 block of the output matrix C. This structured division is specifically designed to match the architecture of the Fermi GPU, allowing for an efficient distribution of computational tasks.

Within each thread block, the setup involves 16×16 threads, which collectively handle the entire 64×64 block. This configuration is critical as it ensures that each thread block is large enough to perform substantial computation but not so large as to cause excessive data management overhead or resource contention. Each thread computes elements within a 16×16 sub-block of the larger matrix, effectively utilizing the Fermi GPU's registers by limiting the number of elements each thread needs to handle simultaneously.

To facilitate this computation, the algorithm uses shared memory by first loading 64×16 elements of matrix `A` and 16×64 elements of matrix `B` into it. This use of shared memory is strategic—it serves as a staging area that reduces the frequency and volume of slower global memory accesses. Once in shared memory, subsets of this data—specifically, four elements from matrix `A` and four from matrix `B` are loaded into the registers of each thread.

By having each thread load a total of eight elements into registers, the algorithm minimizes the need for memory accesses during the core phase of matrix multiplication, which involves numerous operations. This efficient register utilization not only speeds up individual computations, but also significantly enhances overall kernel performance by reducing the overhead associated with memory accesses. Moreover, the organization of data loading and computation allows for coalesced writes back to global memory, which are optimized for memory bandwidth and reduce the overall execution time of the kernel [14].

2.7 Tensor Cores Utilization

Tensor Cores are programmed using the CUDA WMMA API, which allows for the efficient execution of mixed-precision GEMM (General Matrix to Matrix Multiplication) operations. Below is an example that demonstrates the use of WMMA for performing a simple matrix multiplication using Tensor Cores.

```
#include <mma.h>
using namespace nvcuda;

__global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K) {
    // Define the fragments
    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::col_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

    // Initialize the output to zero
    wmma::fill_fragment(c_frag, 0.0f);

    // Load the inputs
    wmma::load_matrix_sync(a_frag, a, M);
    wmma::load_matrix_sync(b_frag, b, K);

    // Perform the matrix multiplication
    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

    // Store the output
    wmma::store_matrix_sync(c, c_frag, M, wmma::mem_col_major);
}
```

This kernel performs the matrix multiplication of two input matrices `A` and `B`, both of size 16×16, using Tensor Cores and stores the result in matrix `C`. Each element of matrices `A` and `B` is a half-precision floating-point number (half), while the elements of matrix `C` are single-precision floating-point numbers (float).

The WMMA API is designed to work with fragments of the matrices. A fragment is a data structure that holds a portion of a matrix tile that Tensor Cores can compute on. For the matrices in this case, we utilize 16×16 segments. The functions `wmma::load_matrix_sync`, `wmma::mma_sync`, and `wmma::store_matrix_sync` allow you to load data into fragments, execute the matrix multiply-accumulate operation, and store the result back into memory.

Tensor Cores significantly speed up matrix computations, but because they utilise half-precision inputs, they present certain issues, especially with precision. Significant rounding mistakes may result from this, particularly in situations where high numerical accuracy is required. However, methods such as mixed precision programming, which combines half- and full precisions to balance accuracy and performance, and precision refinement can help to reduce these inaccuracies [8].

2.8 Autotuning

Autotuning is a technique that optimizes block sizes and grid dimensions to maximize computational efficiency and resource utilization. The size and dimension of blocks per grid and threads per block are important factors in kernel's performance.

While selecting the parameters for the execution configuration should be done simultaneously, there are some general guidelines that can be applied to each parameter separately. The main goal is to keep the entire GPU occupied while selecting the first execution configuration parameter, which is the number of blocks per grid, or grid size. For each multiprocessor to have at least one block to execute, the number of blocks in a grid must exceed the number of multiprocessors.

Additionally, each multiprocessor should have numerous active blocks so that the hardware can be kept occupied by blocks that aren't waiting for a synchronization.

To help with coalescing and prevent wasting computation on under-populated warps, the number of threads per block should be a multiple of the warp size. Only when there are several concurrent blocks per multiprocessor should a minimum of 64 threads per block be employed. A decent place to start when experimenting with varied block sizes is between 128 and 256 threads per block. If latency is an issue for a multiprocessor, use multiple smaller thread blocks instead of one larger one. Kernels that call `__syncthreads()` regularly will especially benefit from this.[5].

Chapter 3

Implementation

This chapter is dedicated to the implementation of kernels that utilize all the techniques and strategies listed in the previous chapter. The goal is to implement optimizations for both matrix multiplication and transposition in the TNL project. The following kernels have been developed and optimized.

Dense Matrix Multiplication

- **Kernel 1.1:** Implemented in TNL the authors's involvement in this project.
- **Kernel 1.2:** Optimizes linear thread index calculations for efficient shared memory access.
- **Kernel 1.3:** Utilizes 2D shared memory arrays to improve memory access patterns during matrix multiplication.
- **Kernel 1.4:** Implements a tile-based approach with two-dimensional shared memory arrays for matrix multiplication.
- **Kernel 1.5:** Identical with Kernel 1.4 but includes padding within shared memory to minimize bank conflicts and further enhance performance.
- **Kernel 1.6:** Employs Register Blocking technique.

Dense Matrix Transpostion

- **Kernel 2.1:** Implemented in TNL before my involvement in this project.
- **Kernel 2.2:** Combines aligned and non-aligned strategies for matrix transposition into a single kernel, using a dynamic selection mechanism based on matrix dimension alignment.
- **Kernel 2.3:** Consolidates matrix transposition into one kernel with an extra column in shared memory configuration to avoid bank conflicts, applicable for both row-major and column-major formats.
- **Kernel 2.4:** Enhances Kernel 2.3 by enabling in-place transposition, specifically optimized for square matrices to save memory space.

We will explore the implementation specifics of each kernel using a profiling tool.

3.1 Profiling

NVIDIA's `nvprof` command-line profiler was used to collect comprehensive performance data of the kernels implemented below. With *nvprof* is possible to gather a wide range of performance measures, such as hardware utilization efficiencies and memory operations.

The test matrix configurations were chosen with care to assess the GPU kernels in a variety of circumstances:

- **Square Matrices:** Sizes like 128×128 to 1024×1024 test the kernels under conditions where memory access patterns are optimal and computation is uniform across dimensions.
- **Rectangular Matrices with More Rows than Columns:** These matrices (e.g., 1024×512) assess the kernels' efficiency in handling scenarios where the aspect ratio favors more rows, posing challenges in parallelization and memory access.
- **Rectangular Matrices with More Columns than Rows:** Configurations such as 512×1024 examine performance under conditions where the aspect ratio favors more columns, which can stress different aspects of memory bandwidth and computational distribution.

These configurations provide thorough understanding of the behaviour of the kernels across common use cases, offering a solid foundation for application deployment and optimization.

The environment for all profiling measurements was provided by the Faculty of Nuclear Sciences and Physical Engineering at CTU in Prague. Detailed specifications of the system used are listed in the table below.

System Component	**Specification**
Hostname	gp1.fjfi.cvut.cz
CPU	2x Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
Cores	8 cores
CPU Cache	20480 KB
RAM	125 Gi
GPU	Quadro P6000, 24576 MiB

Table 3.1: System specifications for GP1

3.1.1 Performance Metrics

The following table details the GPU performance metrics collected using NVIDIA's `nvprof` tool. The metrics provide insights into various aspects of GPU performance and efficiency. Each metric is explained briefly to facilitate understanding of its impact on the GPU's performance.

Metric	**Description**
Instructions per Warp	Average number of instructions executed per warp.
Global Load Throughput (GB/s)	Rate at which data is read from global memory.
Global Store Throughput (GB/s)	Rate at which data is written to global memory.
SM Load Trans. Per Req.	Average number of transactions per request from a Streaming Multiprocessor to load data.
SM Store Trans. Per Req.	Average number of transactions per request from a Streaming Multiprocessor to store data.
Warp Execution Efficiency (%)	Percentage of warps that are eligible for execution vs. those stalled.
Issued IPC	Instructions per cycle issued to the GPU. Indicates how well the SMs are being utilized.
Achieved Occupancy	Ratio of active warps to the maximum number of warps supported per SM.
SM Utilization	General indication of the activity level of the Streaming Multiprocessors.
DRAM Read Throughput (GB/s)	Rate at which data is read from the Dynamic Random Access Memory (DRAM).
DRAM Write Throughput (GB/s)	Rate at which data is written to the Dynamic Random Access Memory (DRAM).

Table 3.2: GPU Performance Metrics and Their Descriptions

3.2 Dense Matrix Multiplication

3.2.1 Kernel 1.1

The first kernel, which was already implemented in TNL, employs key optimization features to achieve efficient matrix multiplication on GPUs. It starts by allocating 1D shared memory tiles designated for `matrixA` and `matrixB`, as well as for the resultant `matrixC` tile. Each tile is defined by a dimension `tileDim`, which specifies the size of the square submatrix each tile represents.

```
// Shared memory tiles for matrix A, B, and C
shared RealType tileA[ tileDim * tileDim ];
shared RealType tileB[ tileDim * tileDim ];
shared RealType tileC[ tileDim * tileDim ];
```

Tile `tileC` is initialized to zero, ensuring a clean state for accumulation across all its elements. The initialization is performed efficiently using a loop that increments by a block size specific to the rows, `tileRowBlockSize`, allowing each thread to handle a specific element in the tile.

```
// Initialize the result tile to zero
for( IndexType row = 0; row < tileDim; row += tileRowBlockSize )
    tileC[ ( row + threadIdx.y ) * tileDim + threadIdx.x ] = 0.0;
```

Next, the coordinates for the result tile are computed based on the grid and block indices.

```
// Compute the result tile coordinates
const IndexType resultTileRow = ( gridIdx_y * gridDim.y + blockIdx.y ) * tileDim;
const IndexType resultTileColumn = ( gridIdx_x * gridDim.x + blockIdx.x ) * tileDim;
```

The kernel then iteratively loads `matrixA` and `matrixB` tiles into shared memory and computes the dot product for each element in the result tile.

```
// Sum over the matrix tiles
for( IndexType i = 0; i < matrixAColumns; i += tileDim ) {
    for( IndexType row = 0; row < tileDim; row += tileRowBlockSize ) {
        ...
        if( matrixARow < matrixARows && matrixAColumn < matrixAColumns )
            tileA[ ( threadIdx.y + row ) * tileDim + threadIdx.x ] = matrixA( matrixARow, matrixAColumn );
            ...
        if( matrixBRow < matrixBRows && matrixBColumn < matrixBColumns )
            tileB[ ( threadIdx.y + row ) * tileDim + threadIdx.x ] = matrixB( matrixBRow, matrixBColumn );
    }
    __syncthreads();
}
```

In the provided code snippet above, the loop variable `i` indexes across the columns of `matrixA` and the rows of `matrixB`, advancing by `tileDim` with each iteration to cover all necessary segments of the matrices for the multiplication.

Finally, the computed tile is written back to the global memory result matrix.

```
// Write the result tile to the result matrix
for( IndexType row = 0; row < tileDim; row += tileRowBlockSize ) {
    const IndexType matrixCRow = resultTileRow + row + threadIdx.y;
    const IndexType matrixCColumn = resultTileColumn + threadIdx.x;
    if( matrixCRow < matrixCRows && matrixCColumn < matrixCColumns )
        resultMatrix( matrixCRow, matrixCColumn ) = tileC[ ( row + threadIdx.y ) * tileDim + threadIdx.x ];
}
```

The kernel efficiently leverages 1D shared memory and optimized indexing to accelerate matrix multiplication on GPUs, achieving high performance.

3.2.2 Kernel 1.2

In the optimization process for `Kernel 1.2`, the focus is on linear thread index calculation for more efficient access to elements in shared memory. Each thread calculates its linear index and then determines its row and column in the tile, simplifying indexing and increasing performance during data loading.

```
// Optimized index calculation
IndexType linearThreadIdx = threadIdx.y * blockDim.x + threadIdx.x;
row = linearThreadIdx / tileDim;
col = linearThreadIdx % tileDim;
```

This optimized calculation allows for direct mapping of threads to their respective elements in the tile, enhancing the data loading process from global to shared memory and thus speeding up the overall multiplication operation.

The following table summarizes the performance metrics, providing a comparative analysis with the percentage change from Kernel 1.1 to Kernel 1.2.

Metric	**Kernel 1.1 (Avg)**	**Kernel 1.2 (Avg)**	**Change (%)**
Instr. per Warp	4626.0	4311.1	- 6.80718
Global Load Thr. (GB/s)	127.80	132.90	+ 3.99061
Global Store Thr. (GB/s)	4.4548	4.6326	+ 3.9912
SM Load Trans. Per Req.	1.940085	2.000000	+ 3.08827
SM Store Trans. Per Req.	1.496247	1.984820	+ 32.6532
Warp Exec. Eff. (%)	100.00	100.00	-
Issued IPC	0.407371	0.393491	-3.40721
Achieved Occupancy	0.846299	0.846157	- 0.01678
SM Utilization	Low (2)	Low (2)	-
DRAM Read Thr. (GB/s)	8.9935	10.071	+ 11.9809
DRAM Write Thr. (GB/s)	1.1111	1.1635	+ 4.71605

Table 3.3: Comparative Performance Metrics and Percentage Increase (Kernel 1.1 vs Kernel 1.2)

The comparative analysis reveals significant improvements in memory handling efficiencies with Kernel 1.2, particularly in Global Load Throughput and DRAM Throughput. These enhancements indicate that Kernel 1.2 is more effective in handling memory-intensive operations.

3.2.3 Kernel 1.3

Kernel 1.3 introduces an optimization by 2D shared memory arrays for improved memory access patterns keeping the linear thread index calculation from Kernel 1.2. Instead of 1D arrays, this approach uses 2D for storing tiles of matrices in shared memory, allowing more efficient indexing within the matrix multiplication process. Shared memory tiles `tileA`, `tileB`, and `tileC` are initialized for matrices `A`, `B`, and the result respectively, with dimensions `tileDim` × `tileDim`.

```
// 2D shared memory usage
__shared__ RealType tileA[ tileDim ][ tileDim ];
__shared__ RealType tileB[ tileDim ][ tileDim ];
__shared__ RealType tileC[ tileDim ][ tileDim ]
```

By introducing 2D shared memory arrays, Kernel 1.3 enhances memory access efficiency, further optimizing matrix multiplication performance on GPUs. The table below compares the performance metrics.

Metric	**Kernel 1.1 (Avg)**	**Kernel 1.3 (Avg)**	**Change (%)**
Instr. per Warp	4626.0	4235.0	- 8.45223
Global Load Thr. (GB/s)	127.80	132.17	+ 3.41941
Global Store Thr. (GB/s)	4.4548	4.6071	+ 3.41878
SM Load Trans. Per Req.	1.940085	2.000000	+ 3.08827
SM Store Trans. Per Req.	1.496247	1.984820	+ 32.6532
Warp Exec. Eff. (%)	100.00	100.00	-
Issued IPC	0.407371	0.383543	- 5.84921
Achieved Occupancy	0.846299	0.846527	+ 0.02694
SM Utilization	Low (2)	Low (2)	-
DRAM Read Thr. (GB/s)	8.9935	8.9339	- 0.662701
DRAM Write Thr. (GB/s)	1.1111	1.1513	+ 3.61804

Table 3.4: Comparative Performance Metrics and Percentage Increase (Kernel 1.1 vs Kernel 1.3)

The adjustment in shared memory arrays along with the linear thread indexing from Kernel 1.2 increased the efficiency in Kernel 1.3 as evidenced by the enhanced Global Load and Store Throughputs. The structured use of 2D shared memory contributes to the noted increases in SM Load and Store Transactions per Request, reflecting a refined handling of data within GPU memory spaces.

3.2.4 Kernel 1.4

Kernel 1.4 implements a tile-based approach for matrix multiplication on GPUs, optimizing memory access and computational efficiency.

Firstly, 2D shared memory arrays, `tileA` and `tileB`, are declared to store portions of the input matrices `A` and `B`.

```
// Two-dimensional shared memory arrays
__shared__ RealType tileA[ tileDim ][ tileDim ];
__shared__ RealType tileB[ tileDim ][ tileDim ];
```

Secondly, instead of creating a separate shared memory tile, a local accumulator (`CValue`) is initialized to zero for each thread. This accumulator will hold the multiplication product results for the elements of the result matrix `C` that the thread is responsible for computing.

```
// Initialize the accumulator for C
RealType CValue = 0;
```

In the indexing calculation each thread computes its unique row and column indices relative to the overall matrix dimensions.

```
// Calculate thread and block indices
IndexType bx = blockIdx.x, by = blockIdx.y;
IndexType tx = threadIdx.x, ty = threadIdx.y;

// Calculate the row and column index
IndexType row = by * tileDim + ty;
IndexType col = bx * tileDim + tx;
```

Then, the tile-based computation loop follows. In the loading phase, threads collaboratively load elements from matrices `A` and `B` into `tileA` and `tileB`, including boundary checks. In the multiplication phase, each thread calculates the dot product of the corresponding elements in `tileA` and `tileB`, accumulating the results in `CValue`. Synchronization points are included as a barrier that ensures all threads have completed loading their respective elements into shared memory before proceeding.

```
// Loop over the tiles of the input matrices
for (IndexType m = 0; m < (tileDim + matrixA.getColumns() - 1)/tileDim; ++m) {
   // Load A and B tiles into shared memory
   if (m * tileDim + tx < matrixA.getColumns() && row < matrixA.getRows())
         tileA[ty][tx] = matrixA(row, m * tileDim + tx);
   else
         tileA[ty][tx] = 0.0;

   if (m * tileDim + ty < matrixB.getRows() && col < matrixB.getColumns())
         tileB[ty][tx] = matrixB(m * tileDim + ty, col);
   else
         tileB[ty][tx] = 0.0;

   __syncthreads();

   // Compute product for this tile
   for (IndexType k = 0; k < tileDim; ++k)
         CValue += tileA[ty][k] * tileB[k][tx];

   __syncthreads();
}
```

And finally, after completing all iterations, each thread writes its accumulated value, scaled by `matrixMultiplicator`, to the appropriate element in the result matrix `C`, provided the indices fall within the bounds of `C`.

```
// Write the result to the global memory
if (row < resultMatrix.getRows() && col < resultMatrix.getColumns())
   resultMatrix(row, col) = CValue * matrixMultiplicator;
```

The fourth kernel efficiently employs a tile-based approach, leveraging 2D shared memory tiles and a local accumulator for optimized matrix multiplication on GPUs. The following table is for comparison.

Metric	**Kernel 1.1 (Avg)**	**Kernel 1.4 (Avg)**	**Change (%)**
Instr. per Warp	4626.0	2591.5	- 43.9797
Global Load Thr. (GB/s)	127.80	210.23	+ 64.4992
Global Store Thr. (GB/s)	4.4548	7.3284	+ 64.5057
SM Load Trans. Per Req.	1.940085	2.000000	+ 3.08827
SM Store Trans. Per Req.	1.496247	2.000000	+ 33.6678
Warp Exec. Eff. (%)	100.00	100.00	-
Issued IPC	0.407371	0.375813	- 7.74675
Achieved Occupancy	0.846299	0.840715	- 0.659814
SM Utilization	Low (2)	Low (3)	-
DRAM Read Thr. (GB/s)	8.9935	20.751	+ 130.733
DRAM Write Thr. (GB/s)	1.1111	1.8462	+ 66.1597

Table 3.5: Comparative Performance Metrics and Percentage Increase (Kernel 1.1 vs Kernel 1.4)

By utilizing tile-based computation and maintaining synchronization between threads, Kernel 1.4 significantly reduces Instructions per Warp indicating a more efficient execution strategy. Substantial increases in Global Load and Store Throughputs suggest improved memory bandwidth utilization. The most notable enhancement is in DRAM Read Throughput, emphasizing the kernel's capacity to manage large data volumes effectively.

3.2.5 Kernel 1.5

Kernel 1.5 builds on the tile-based approach of Kernel 1.4 by integrating padding in shared memory to reduce bank conflicts.

```
// Two-dimensional shared memory arrays with added padding
__shared__ RealType tileA[ tileDim ][ tileDim + 1];
__shared__ RealType tileB[ tileDim ][ tileDim + 1];
```

This version strategically employs an additional element in the second dimension of the tile arrays, improving memory access patterns among threads. This modification prevents simultaneous access to the same memory bank, thereby avoiding performance degradation due to serialization of memory operations. The table below compares the performance metrics.

Metric	Kernel 1.1 (Avg)	Kernel 1.5 (Avg)	Change (%)
Instr. per Warp	4626.0	2592.5	- 43.9581
Global Load Thr. (GB/s)	127.80	227.18	+ 77.7621
Global Store Thr. (GB/s)	4.4548	7.9193	+ 77.77
SM Load Trans. Per Req.	1.940085	1.500000	- 22.6838
SM Store Trans. Per Req.	1.496247	2.000000	+ 33.6678
Warp Exec. Eff. (%)	100.00	100.00	-
Issued IPC	0.407371	0.406589	- 0.191963
Achieved Occupancy	0.846299	0.839794	- 0.768641
SM Utilization	Low (2)	Low (2)	-
DRAM Read Thr. (GB/s)	8.9935	22.791	+ 153.416
DRAM Write Thr. (GB/s)	1.1111	1.9788	+ 78.0938

Table 3.6: Comparative Performance Metrics and Percentage Increase (Kernel 1.1 vs Kernel 1.5)

The performance metrics demonstrate substantial improvements from the previous Kernel 1.4, with notable reductions in shared memory load transactions per request and significant increases in global memory load throughput. The overall performance improvement is evident, indicating the effectiveness of the added padding strategy in optimizing memory access patterns and computational efficiency.

3.2.6 Kernel 1.6

Kernel 1.6 employs register blocking technique with inspiration from the previously discussed article [14]. The kernel begins by setting up the thread block and grid dimensions, and initializing 2D shared memory arrays with padding to prevent bank conflicts.

```
// Shared memory for submatrices of A and B
__shared__ RealType sharedA[ 64 ][ 16 + 1 ];
__shared__ RealType sharedB[ 16 ][ 64 + 1 ];

// Each thread computes a 4x4 block of the result matrix
IndexType row = blockIdx.y * 64 + threadIdx.y * 4;
IndexType col = blockIdx.x * 64 + threadIdx.x * 4;

// Local storage for the result of a 4x4 block
RealType CValue[ 4 ][ 4 ] = { 0 };
```

Then, each thread loads a 4×4 block of the resulting matrix, utilizing shared memory effectively to enhance data access speed. One of the key optimizations in Kernel 1.6 is the precomputation of tile numbers and boundary conditions. By calculating `numTiles`, the kernel determines how many iterations are required to cover all data in matrix A and B, minimizing out-of-bounds checks during execution. Similarly, `maxARow` and `maxBCol` ensure that threads do not attempt to load or compute beyond the actual dimensions of the matrices involved, which is critical for handling matrices that are not perfectly divisible by the block size.

```
// Precompute the number of tiles
const IndexType numTiles = ( matrixAColumns + 15 ) / 16;

// Precompute the maximum valid iterations for aRow and bCol
```

```
IndexType maxARow = min( 4, matrixARows - row );
IndexType maxBCol = min( 4, matrixBColumns - col );

// Iterate through each tile
for( IndexType m = 0; m < numTiles; ++m ) {
    IndexType aCol = m * 16 + threadIdx.x;
    IndexType bRow = m * 16 + threadIdx.y;

    // Load valid data from global memory into shared memory for matrix A
    if( aCol < matrixAColumns ) {
        for( IndexType i = 0; i < maxARow; ++i ) {
            IndexType aRow = row + i;
            sharedA[ threadIdx.x ][ threadIdx.y * 4 + i ] = AValues[ aCol * matrixARows + aRow ];
        }

         for( IndexType i = maxARow; i < 4; ++i ) {
            sharedA[ threadIdx.x ][ threadIdx.y * 4 + i ] = 0.0;
         }
    }
    else {
#pragma unroll
        for( IndexType i = 0; i < 4; ++i ) {
            sharedA[ threadIdx.x ][ threadIdx.y * 4 + i ] = 0.0;
        }
    }

    // Load valid data from global memory into shared memory for matrix B
    if( bRow < matrixBRows ) {
        for( IndexType i = 0; i < maxBCol; ++i ) {
            IndexType bCol = col + i;
            sharedB[ threadIdx.x * 4 + i ][ threadIdx.y ] = BValues[ bCol * matrixBRows + bRow ];
        }

        for( IndexType i = maxBCol; i < 4; ++i ) {
            sharedB[ threadIdx.x * 4 + i ][ threadIdx.y ] = 0.0;
        }
    }
    else {
#pragma unroll
        for( IndexType i = 0; i < 4; ++i ) {
            sharedB[ threadIdx.x * 4 + i ][ threadIdx.y ] = 0.0;
        }
    }
    __syncthreads();

```

For matrix A, each thread loads a column of 4 elements into the sharedA array. If the column index exceeds the matrix bounds, the thread sets those elements to zero. For matrix B, each thread loads a row of 4 elements into the sharedB array, with similar boundary checks and zero-padding.

Once the necessary data is in shared memory, each thread first loads the data into registers for increased speed. Each thread uses 8 registers to temporarily store 4 elements from sharedA and 4 elements from sharedB. After loading the data into registers, the threads perform the multiplication operation. The results of these multiplications are accumulated in CValue.

```
// Compute the matrix multiplication for this tile
for( IndexType k = 0; k < 16; ++k ) {
    RealType regA[ 4 ], regB[ 4 ];
    for( IndexType i = 0; i < 4; ++i ) {
        regA[ i ] = sharedA[ threadIdx.y * 4 + i ][ k ];
        regB[ i ] = sharedB[ k ][ threadIdx.x * 4 + i ];
        }
    for( IndexType i = 0; i < 4; ++i ) {
        for( IndexType j = 0; j < 4; ++j ) {
            CValue[ i ][ j ] += regA[ i ] * regB[ j ] * matrixMultiplicator;
        }
     }
}
```

Finally, the computed results are stored back to the result matrix using coalesced writes to optimize memory bandwidth usage.

```
// Store the result into the result matrix
for( IndexType i = 0; i < maxARow; ++i ) {
    for( IndexType j = 0; j < maxBCol; ++j ) {
        IndexType cRow = row + i;
        IndexType cCol = col + j;

        IndexType index = cCol * matrixARows + cRow;
        resultValues[ index ] = CValue[ i ][ j ];
    }
}
```

The performance of Kernel 1.6 is profiled in the table below.

Metric	**Kernel 1.1 (Avg)**	**Kernel 1.6 (Avg)**	**Change (%)**
Instr. per Warp	4626.0	24903	+ 438.327
Global Load Thr. (GB/s)	127.80	47.254	- 63.025
Global Store Thr. (GB/s)	4.4548	4.3926	- 1.39625
SM Load Trans. Per Req.	1.940085	4.500000	+ 131.949
SM Store Trans. Per Req.	1.496247	5.000000	+ 234.169
Warp Exec. Eff. (%)	100.00	100.00	-
Issued IPC	0.407371	0.158391	- 61.1187
Achieved Occupancy	0.846299	0.253719	- 70.0202
SM Utilization	Low (2)	Low (1)	-
DRAM Read Thr. (GB/s)	8.9935	2.7453	- 69.4746
DRAM Write Thr. (GB/s)	1.1111	1.5252	+ 37.2694

Table 3.7: Comparative Performance Metrics and Percentage Increase (Kernel 1.1 vs Kernel 1.6)

The performance profiling between Kernel 1.1 and Kernel 1.6 clearly demonstrates the impact of the enhanced register use. The dramatic increase in Instructions per Warp indicates a higher computational complexity per warp. Despite this, there are significant declines in Global Load Throughput and DRAM Read Throughput, suggesting that the kernel's intense computational focus may compromise memory access efficiency.

3.3 Dense Matrix Transposition

3.3.1 OutOfPlace Transposition

3.3.1.1 Kernel 2.1

The Kernel 2.1, which was already implemented in TNL, is segmented into aligned and non-aligned versions. The choice between the use of these implementations is determined by the matrix alignment needs, with each version tailored to optimize specific scenarios in matrix transposition operations. Both kernels leverage the same fundamental structure but differ in how they handle memory accesses.

Each kernel begins by allocating a 1D shared memory tile to store a sub-matrix of the input. This enables efficient reuse of data, reducing global memory accesses.

```
// Shared memory tile for the input matrix
__shared__ Real tile[ tileDim * tileDim ];
```

To ensure coalesced memory access, the kernels map blocks diagonally, adjusting the block indices (`blockIdx_x` and `blockIdx_y`) based on whether the matrix is square or rectangular.

```
// Diagonal mapping of the CUDA blocks
Index blockIdx_x, blockIdx_y;
if( columns == rows ) {
    blockIdx_y = blockIdx.x;
    blockIdx_x = ( blockIdx.x + blockIdx_y ) % gridDim.x;
} else {
    Index bID = blockIdx.x + gridDim.x * blockIdx.y;
    blockIdx_y = bID % gridDim.y;
    blockIdx_x = ( ( bID / gridDim.y ) + blockIdx_y ) % gridDim.x;
}
```

The tiles of the input matrix are loaded into shared memory in blocks of `tileRowBlockSize` rows, enabling efficient reading from the input matrix.

```
// Read the tile to the shared memory
const Index readRowPosition = ( gridIdx_y * gridDim.y + blockIdx_y ) * tileDim + threadIdx.y;
const Index readColumnPosition = ( gridIdx_x * gridDim.x + blockIdx_x ) * tileDim + threadIdx.x;
for( Index rowBlock = 0; rowBlock < tileDim; rowBlock += tileRowBlockSize ) {
    tile[ Backend::getInterleaving( threadIdx.x * tileDim + threadIdx.y + rowBlock ) ] =
       inputMatrix( readRowPosition + rowBlock, readColumnPosition );
}
```

The differences between the two kernels appear in how each reads and writes tiles based on alignment.

In the aligned kernel, the tile is read directly into shared memory assuming aligned access.

```
for (Index rowBlock = 0; rowBlock < tileDim; rowBlock += tileRowBlockSize) {
    tile[Backend::getInterleaving(threadIdx.x * tileDim + threadIdx.y + rowBlock)] =
       inputMatrix(readRowPosition + rowBlock, readColumnPosition);
}
```

However, in the non-aligned kernel, additional checks are performed for non-aligned memory access.

```
if (readColumnPosition < columns) {
    for (Index rowBlock = 0; rowBlock < tileDim; rowBlock += tileRowBlockSize) {
        if (readRowPosition + rowBlock < rows)
            tile[Backend::getInterleaving(threadIdx.x * tileDim + threadIdx.y + rowBlock)] =
               inputMatrix(readRowPosition + rowBlock, readColumnPosition);
    }
}
```

In writing back to global memory, the non-aligned kernel ensures non-aligned writes are valid.

```
if (writeColumnPosition < rows) {
    for (Index rowBlock = 0; rowBlock < tileDim; rowBlock += tileRowBlockSize) {
        if (writeRowPosition + rowBlock < columns)
            resultMatrix(writeRowPosition + rowBlock, writeColumnPosition) =
            matrixMultiplicator * tile[Backend::getInterleaving(
                                                    (threadIdx.y + rowBlock) * tileDim + threadIdx.x)];
    }
}
```

Despite these differences in handling memory access patterns, both implementations aim to optimize the transposition of matrices efficiently on GPUs by utilizing shared memory and structured data access patterns.

3.3.1.2 Kernel 2.2

The Kernel 2.2 combines the approaches of the aligned and the non-aligned transposition kernels listed above. The optimized approach uses a boolean parameter - `isAligned` to dynamically select the appropriate transposition strategy based on the alignment of the matrix dimensions with the tile size. This flexibility allows the kernel to handle both cases without the need for separate kernels. This is achieved by conditional checks within the memory access patterns, ensuring that only valid memory locations are read from or written to, thus avoiding out-of-bounds memory access and ensuring the integrity of the transposition process.

```
// Read the tile to the shared memory
const Index readRowPosition = ( gridIdx_y * gridDim.y + blockIdx_y ) * tileDim + threadIdx.y;
const Index readColumnPosition = ( gridIdx_x * gridDim.x + blockIdx_x ) * tileDim + threadIdx.x;
if( isAligned || readColumnPosition < columns ) {
    for( Index rowBlock = 0; rowBlock < tileDim; rowBlock += tileRowBlockSize ) {
        if( isAligned || ( readRowPosition + rowBlock < rows ) ) {
        tile[ Backend::getInterleaving( threadIdx.x * tileDim + threadIdx.y + rowBlock ) ] =
            inputMatrix( readRowPosition + rowBlock, readColumnPosition );
        }
    }
}
__syncthreads();

// Write the tile to the global memory
const Index writeRowPosition = ( gridIdx_x * gridDim.x + blockIdx_x ) * tileDim + threadIdx.y;
const Index writeColumnPosition = ( gridIdx_y * gridDim.y + blockIdx_y ) * tileDim + threadIdx.x;
if( isAligned || writeColumnPosition < rows ) {
    for( Index rowBlock = 0; rowBlock < tileDim; rowBlock += tileRowBlockSize ) {
        if( isAligned || ( writeRowPosition + rowBlock < columns ) ) {
        resultMatrix( writeRowPosition + rowBlock, writeColumnPosition ) =
            matrixMultiplicator * tile[ Backend::getInterleaving(
                ( threadIdx.y + rowBlock ) * tileDim + threadIdx.x ) ];
        }
    }
}
```

The following table provides a detailed comparison of the performance metrics between Kernel 2.1 and Kernel 2.2. The values presented in the table for Kernel 2.1 are computed as averages between the aligned and non-aligned transposition strategies from earlier kernels.

Metric	**Kernel 2.1 (Avg)**	**Kernel 2.2 (Avg)**	**Change (%)**
Instr. per Warp	161.259375	168.161458	+ 4.28011
Global Load Thr. (GB/s)	226.185	237.51	+ 5.00696
Global Store Thr. (GB/s)	226.185	237.51	+ 5.00696
SM Load Trans. per Req.	1.996032	2.000000	+ 0.198794
SM Store Trans. per Req.	1.988281	1.998677	+ 0.522864
Warp Exec. Eff. (%)	99.71	99.89	+ 0.180524
Issued IPC	0.255455	0.285753	+ 11.8604
Achieved Occupancy	0.716286	0.645097	- 9.93863
SM Utilization	Low (1)	Low (1)	-
DRAM Read Thr. (GB/s)	88.771	87.755	- 1.14452
DRAM Write Thr. (GB/s)	86.960	84.177	- 3.20032

Table 3.8: Comparative Performance Metrics and Percentage Increase (Kernel 2.1 vs Kernel 2.2)

The analysis shows that Kernel 2.2 improves on several key performance metrics, such as Global Load and Store Throughputs and Issued IPC, indicating better computational efficiency and faster memory operations. However, the decline in Achieved Occupancy and DRAM Throughputs could reflect the kernel's increased complexity, which potentially impacts its overall memory efficiency and resource utilization.

3.3.1.3 Kernel 2.3

Kernel 2.3 introduces a highly optimized approach to matrix transposition.

A tile of shared memory is declared with dimensions `tileDim x (tileDim + 1)`. The extra column in the declaration (`tileDim + 1`) is used to avoid shared memory bank conflicts, which can significantly degrade performance due to serialization of memory accesses.

```
__shared__ Real tile[ tileDim ][ tileDim + 1 ];
```

By using shared memory effectively, the kernel minimizes the slow global memory accesses, thus speeding up the transposition process.

The kernel retrieves the number of columns and rows from the input matrix to ensure that threads do not attempt to access out-of-bounds elements. The direct mapping of thread indices to shared memory locations for loading and writing eliminated the need for complex indexing calculations. Each thread calculates its corresponding row and column in the input matrix, scales the matrix element by `matrixMultiplicator`, and stores it in the shared memory tile.

```
const Index matrixColumns = inputMatrix.getColumns();
const Index matrixRows = inputMatrix.getRows();

Index row = blockIdx.y * tileDim + threadIdx.y;
Index col = blockIdx.x * tileDim + threadIdx.x;

if( row < matrixRows && col < matrixColumns ) {
   tile[ threadIdx.y ][ threadIdx.x ] = inputMatrix( row, col ) * matrixMultiplicator;
}

   __syncthreads();
```

The `__syncthreads()` function is called to ensure that all threads have completed their reads and writes to shared memory before proceeding to write back to the result matrix. After that, the threads use transposed indices (swapping `blockIdx.x` with `blockIdx.y`) to write the transposed and scaled elements back to the result matrix.

```
// Adjust writing based on result matrix organization
row = blockIdx.x * tileDim + threadIdx.y;
col = blockIdx.y * tileDim + threadIdx.x;

if( row < matrixColumns && col < matrixRows ) {
   resultMatrix( row, col ) = tile[ threadIdx.x ][ threadIdx.y];
}
```

Conditional checks for `row` and `col` ensure that the kernel does not perform out-of-bounds memory accesses, which could lead to incorrect results and possible crashes.

This implementation reduces the size of the kernel, improving readability and maintainability while maintaining robust performance and flexibility. The performace can be observed in the table below.

Metric	**Kernel 2.1 (Avg)**	**Kernel 2.3 (Avg)**	**Change (%)**
Instr. per Warp	161.259375	39.299805	- 75.6294
Global Load Thr. (GB/s)	226.185	329.33	+ 45.6021
Global Store Thr. (GB/s)	226.185	164.66	- 27.2012
SM Load Trans. per Req.	1.996032	2.000000	+ 0.19880
SM Store Trans. per Req.	1.988281	1.123047	- 43.5167
Warp Exec. Eff. (%)	99.71	99.97	+ 0.260756
Issued IPC	0.255455	0.799932	+ 213.14
Achieved Occupancy	0.716286	0.761484	+ 6.31005
SM Utilization	Low (1)	Low (1)	-
DRAM Read Thr. (GB/s)	88.771	79.054	- 10.9461
DRAM Write Thr. (GB/s)	86.960	77.767	- 10.5715

Table 3.9: Comparative Performance Metrics and Percentage Increase (Kernel 2.1 vs Kernel 2.3)

The significant reduction in Instructions per Warp showcases the simplicity and efficiency of Kernel 2.3. The increase in Global Load Throughput reflects the kernel's enhanced capability to access memory faster. Additionally, a notable surge in Issued IPC indicates that kernel 2.3 can execute significantly more instructions per cycle.

3.3.2 InPlace Transposition

3.3.2.1 Kernel 2.4

Kernel 2.4 builds upon the approach utilized in Kernel 2.3, employing a shared memory tile to facilitate matrix transposition. The primary enhancement in Kernel 2.4 is its capability to transpose matrices in-place, specifically designed for square matrices, thus eliminating the need for additional storage space required by out-of-place methods.

The significant difference between Kernel 2.4 and Kernel 2.3 lies in how the transposed data is written back. Instead of writing to a separate output matrix, Kernel 2.4 writes the data back into the original matrix, enabling in-place transposition.

```
__shared__ Real tile[ tileDim ][ tileDim + 1 ];

Index xIndex = blockIdx.x * tileDim + threadIdx.x;
Index yIndex = blockIdx.y * tileDim + threadIdx.y;

if( xIndex < matrixColumns && yIndex < matrixRows ) {
    tile[ threadIdx.y ][ threadIdx.x ] = matrix( yIndex, xIndex ) * matrixMultiplicator;
}

__syncthreads();

// The indices for writing back are transposed
xIndex = blockIdx.y * tileDim + threadIdx.x;
yIndex = blockIdx.x * tileDim + threadIdx.y;

if( xIndex < matrixRows && yIndex < matrixColumns ) {
    matrix( yIndex, xIndex ) = tile[ threadIdx.x ][ threadIdx.y ];
}
```

This adaptation to write back transposed data directly into the input matrix is what allows Kernel 2.4 to operate in-place, which improves the memory efficiency when dealing with large matrices. However, it is restricted to square matrices to maintain the integrity of the row and column dimensions post-transposition.

Metric	**Kernel 2.1 (Avg)**	**Kernel 2.4 (Avg)**	**Change (%)**
Instr. per Warp	161.259375	45.000000	- 72.0946
Global Load Thr. (GB/s)	226.185	314.06	+ 38.8509
Global Store Thr. (GB/s)	226.185	157.03	- 30.5745
SM Load Trans. per Req.	1.996032	2.000000	+ 0.19880
SM Store Trans. per Req.	1.988281	2.000000	+ 0.58940
Warp Exec. Eff. (%)	99.71	100.0	+ 0.290843
Issued IPC	0.255455	0.398982	+ 56.1848
Achieved Occupancy	0.716286	0.632813	- 11.6536
SM Utilization	Low (1)	Low (1)	-
DRAM Read Thr. (GB/s)	88.771	49.225	- 44.5483
DRAM Write Thr. (GB/s)	86.960	72.069	- 17.1240

Table 3.10: Comparative Performance Metrics and Percentage Increase (Kernel 2.1 vs Kernel 2.4)

Kernel 2.4 retains the simplicity of Kernel 2.3, evidenced by a 72% reduction in Instructions per Warp. This kernel also sees a notable improvement in Global Load Throughput, enhancing its ability to access memory more quickly and efficiently.

Chapter 4

Testing and Benchmarks

4.1 Testing

4.1.1 Roofline model

The Roofline model is a powerful tool for performance analysis, providing a visual representation of the performance limits of a given hardware architecture. It helps in identifying whether a kernel's performance is constrained by the computational capabilities (compute-bound) or by the memory transfer speeds (memory-bound) of the system. The model plots operational intensity (operations per byte of memory traffic) against performance (operations per second), allowing identification of bottlenecks and areas for optimization [19].

For this analysis, we used the Quadro P6000 GPU described in Table 3.1. The GPU has a peak performance of 8.9 TFLOPs and a memory bandwidth of 432 GB/s. The Roofline model was constructed by plotting the performance and computational intensity of the multiplication and transposition kernels.

To understand the Roofline model, it is essential to comprehend the key metrics used in the analysis. Table 4.1 summarizes these metrics and their descriptions.

Metric	**Description**
FLOPs	Number of floating-point operations performed, including all types of floating-point operations such as add, multiply, and FMA (fused multiply-add).
Total Bytes Moved (Bytes)	Total number of bytes read and written during the execution.
Operational Intensity (FLOPs/Bytes)	Ratio of floating-point operations to the total number of bytes moved. It indicates the computational intensity of the kernel.

Table 4.1: GPU Performance Metrics and Their Descriptions

4.1.1.1 Performance Metrics for Multiplication Kernels

Table 4.2 summarizes the performance metrics for the multiplication kernels. These metrics provide insights into the efficiency and computational intensity of each kernel.

Metric	Kernel 1.1	Kernel 1.2	Kernel 1.3	Kernel 1.4	Kernel 1.5	Kernel 1.6
FLOPs	631428437	618397696	618397696	53712486	417432258	625475584
Bytes	32556419	34807468	31419641	1068275	44893557	13954115
FLOPs/Bytes	19.39490	17.76624	19.68188	50.27964	9.29827	44.82373

Table 4.2: Performance Metrics for Roofline Model (Multiplication Kernels)

The following graph 4.1 illustrates the Roofline model for the dense matrix multiplication kernels, where each kernel is represented by a distinct point. The memory bound line represents the maximum performance limited by the memory bandwidth of the GPU. A kernel is considered optimized when it approaches this line, showing that it is efficiently utilizing the available memory bandwidth.

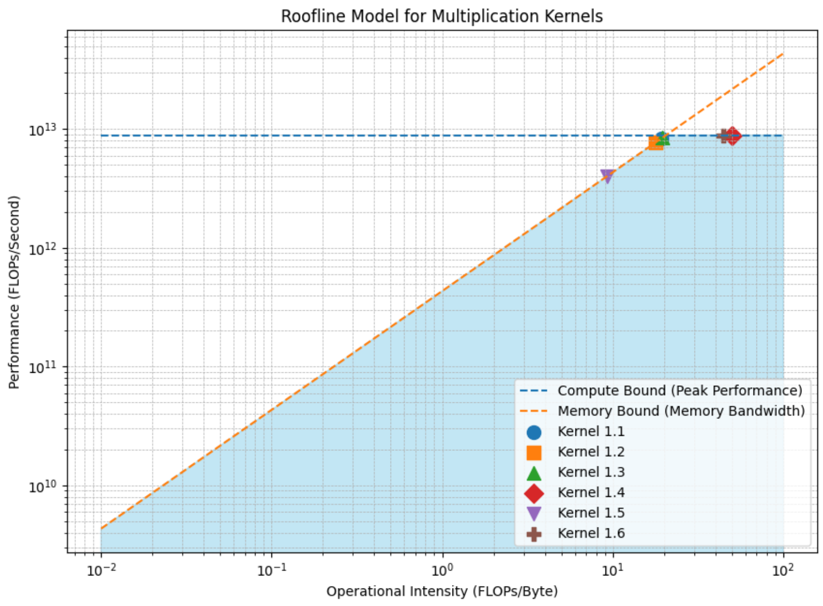


Figure 4.1: Roofline Model for Multiplication Kernels

The roofline model for multiplication kernels shows they have high operational intensities and are mostly compute-bound, effectively utilizing the GPU's peak performance. Kernels 1.1, 1.2, 1.3, and 1.4 reach performance levels close to the GPU's computational limits, indicating efficient use of the hardware's processing power.

4.1.1.2 Performance Metrics for Transposition Kernels

Similarly, Table 4.3 summarizes the performance metrics for the transposition kernels. These metrics help in understanding the efficiency and computational intensity of each kernel in the context of matrix transposition operations.

Metric	**Kernel 2.1**	**Kernel 2.2**	**Kernel 2.3**	**Kernel 2.4**
FLOPs	417382	354304	216064	314456
Bytes	10503660	8207226	3292400	3846542
FLOPs/Bytes	0.03974	0.04317	0.06563	0.01561

Table 4.3: Performance Metrics for Roofline Model (Transposition Kernels)

Figure 4.2 presents the Roofline model for the dense matrix transposition kernels.

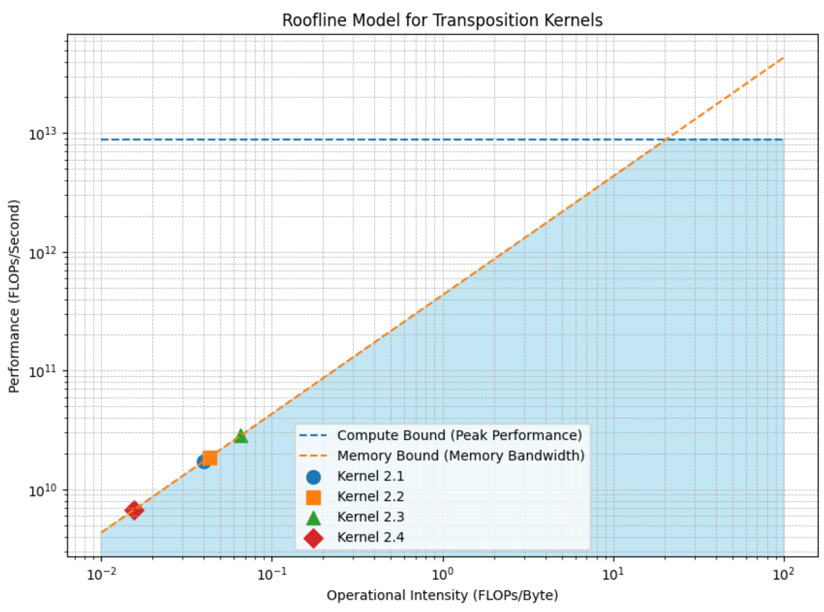


Figure 4.2: Roofline Model for Transposition Kernels

The transposition kernels, although lower in operational intensity, are well-optimized for their purpose. Kernels 2.1, 2.2, 2.3, and 2.4 fall within acceptable performance ranges, indicating that their memory-bound nature is expected and well-handled.

4.1.2 UnitTests

With the aid of the GoogleTest library, which is extensively utilized in the TNL extension, unit tests were created to confirm the accuracy of the implementation. These tests covered matrix multiplication and transposition across different matrix types and memory organizations. The following scenarios were tested:

- **Multiplication with Identity Matrix** - to ensure that the original matrix is returned.
- **Empty Matrices** - to ensure that the operations return an empty matrix.
- **Multiplication with Zero Matrix** - to confirm that any matrix multiplied by a zero matrix returns a zero matrix.

- **Single-Element Matrices** - to ensure that the operations return the same matrix.
- **Single-Row or Single-Column Matrices** - to ensure the correct handling of one-dimensional matrices.
- **Transposition States in Matrix Multiplication** - to ensure computational correctness in all transposition states (none, transposing the first matrix, the second matrix, or both).
- **Small and Large Matrices** - to test computational correctness and efficiency.

Each matrix type was tested for both row-major and column-major layout to verify that all cases are handled correctly under different conditions. The tests can now be run as part of the library compilation because they were incorporated into TNL.

4.1.3 Utilizing Norms

A structure `denseMatricesResult` was defined to hold the benchmark outcomes of operations on dense matrices of our benchmark libraries listed in the following section. The structure is templated on the real number type, device type (e.g., CPU, GPU), and index type to ensure flexibility across different computational environments.

```
// Compute and append differences for each benchmark result
for( const auto& benchmarkMatrix : benchmarkResults ) {
    // Check if dimensions match
    auto diff = referenceResult.getValues() - benchmarkMatrix.getValues();
    elements << TNL::maxNorm( abs( diff ) ) << TNL::l2Norm( diff );
}
```

The results include detailed comparisons between the reference implementation and the benchmark libraries. Differences are calculated using `TNL::maxNorm` and `TNL::L2Norm` for accuracy assessments. All the implemented kernels have been tested through this method and all of them performed well.

4.2 Benchmarks

4.2.1 Benchmark Setup

The benchmark is designed to assess the performance of dense matrix multiplication and transposition, dynamically adapting to different computational environments.

The implementation utilizes template programming to handle various data types and index sizes, which enhances the flexibility and reusability of the code. The primary template parameters are defined for the data type (`Real`) and matrix indices (`Index`).

```
template<typename Real = double, typename Index = int>
...
```

A wide range of settings can be adjusted to fine-tune the benchmark, including the choice of computational device, the number of iterations, and the logging specifics. These configurations are managed using the `TNL::Config::ConfigDescription` object.

The benchmark's execution is initiated by the `runBenchmark()` function. This function configures logging, determines the computation device, and performs numerous tests across various configurations. The results are systematically recorded for analysis.

The framework is structured to manage metadata and execute optimized computational kernels. The metadata setup, kernel launch, and benchmark execution phases are provided below.

```
// Metadata configuration for recording benchmark specifics
benchmark.setMetadataColumns({
    { "index type", getType<Index>() },
    { "device", device },
    { "algorithm", "SpecificAlgorithm" },
    { "matrix1 size", to_string(matrix1Rows) + "x" + to_string(matrix1Columns) },
    { "matrix2 size", to_string(matrix1Columns) + "x" + to_string(matrix2Columns) }
});

// Resetting matrix values before execution
resultMatrix.getValues() = 0;

// Lambda function for optimized kernel execution
auto matrixOperationBenchmarkOptimized = [ & ]() mutable {
    launchMatrixOperationKernel(denseMatrix1, denseMatrix2, resultMatrix);
};


// Differences, calculated using TNL::maxNorm and TNL::L2Norm
std::vector< MatrixType > benchmarkMatricesOptimized = {
    Library1ResultMatrix, Library2ResultMatrix, Library3ResultMatrix
    };

// Timing the execution of the matrix operation benchmark
benchmark.time< DeviceType >(device, matrixOperationBenchmarkOptimized, ResultOfBenchmark);
```

This setup illustrates a systematic approach for configuring the metadata that characterizes each test scenario, including matrix sizes and computational algorithms. The lambda function encapsulates the optimized kernel launch for either multiplication or transposition, ensuring maximum performance across various computational libraries. The execution time is measured, and the metadata and results are recorded for each test, allowing comprehensive performance analysis and comparison between different computational devices and libraries.

4.2.2 Environment

The benchmarks were conducted on two different systems equipped with NVIDIA CUDA environments. These systems will be referred to as HELIOS and GP1. The specifications for GP1 are listed in Table 3.1, and the specifications for HELIOS are listed below.

System Component	Specification
Hostname	helios.fjfi.cvut.cz
CPU	Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
Cores	16 cores per CPU, 32 threads total
CPU Cache	22528 KB
RAM	376 Gi
GPU	NVIDIA Tesla V100, 32 GiB

Table 4.4: System specifications for HELIOS

We decided that it is beneficial to test the kernel on two different architectures. GP1 has a Pascal architecture GPU, while HELIOS is equipped with a Tesla architecture GPU. The benchmarks will be performed using the exact same matrix sizes on both systems to ensure a consistent and fair comparison of performance.

4.2.3 Benchmark Libraries

This section introduces the libraries used for benchmarking and testing the performance of the developed kernels. Benchmarking against widely recognized libraries is essential for understanding the efficiency and effectiveness of this work in comparison to established standards. We focused on selecting libraries that offer functions performing the same operations as our kernels.

4.2.3.1 CuBLAS

The NVIDIA CUDA Basic Linear Algebra Subprograms (CuBLAS) library is a GPU-accelerated library providing efficient linear algebra subroutines. It offers APIs for matrix and vector operations in GPU memory, including a flexible API for General Matrix Multiply (GEMM) operations. This flexibility allows for various matrix data layouts and computational optimizations [4]. We utilized the CuBLAS GEMM function and used TNL templates for the benchmarking setup, ensuring compatibility.

```
// Perform the matrix multiplication using cuBLAS
if constexpr(std::is_same_v<RealType, float>) {
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, &alpha,
                matrix1.getValues().getData(), lda,
                matrix2.getValues().getData(), ldb, &beta,
                resultMatrix.getValues().getData(), ldc);
} else if constexpr(std::is_same_v<RealType, double>) {
    cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, &alpha,
                matrix1.getValues().getData(), lda,
                matrix2.getValues().getData(), ldb, &beta,
                resultMatrix.getValues().getData(), ldc);
}
```

Key parameters for the `cublasSgemm` or `cublasDgemm` function calls are as follows.

- **handle**: Manages CuBLAS operation resources.
- **TRANSA and TRANSB**: Specify matrix operations; ‘CUBLAS_OP_N‘ means no transpose.
- **m, n, k**: Dimensions of the matrices involved in the multiplication.

- **alpha** and **beta**: Scalars for scaling the matrix product and the input matrix C, respectively.
- **A** and **B**: Input matrices, with **lda** and **ldb** indicating their leading dimensions.
- **C**: The result matrix, with **ldc** as its leading dimension.

This setup ensures efficient matrix multiplication leveraging GPU capabilities for enhanced performance.

4.2.3.2 MAGMA

The Matrix Algebra on GPU and Multicore Architectures (MAGMA) library provides high-performance linear algebra operations on heterogeneous computing systems, combining the power of multi-core CPUs and GPUs. It offers an interface compatible with existing linear algebra frameworks, facilitating the efficient utilization of hybrid computing resources [16]. In the benchmarks, we used the MAGMA GEMM function, adapting it to specific configurations to align with the requirements of our benchmark.

```
// Perform the matrix multiplication using MAGMA
if constexpr( std::is_same_v<RealType, float> ) {
    magma_sgemm(MagmaNoTrans, MagmaNoTrans, m, n, k, alpha,
                matrix1.getValues().getData(), lda,
                matrix2.getValues().getData(), ldb,
                beta, resultMatrix.getValues().getData(), ldc, queue);
} else if constexpr( std::is_same_v<RealType, double> ) {
    magma_dgemm(MagmaNoTrans, MagmaNoTrans, m, n, k, alpha,
                matrix1.getValues().getData(), lda,
                matrix2.getValues().getData(), ldb,
                sbeta, resultMatrix.getValues().getData(), ldc, queue);
}
```

Key parameters for the `magma_sgemm` or `magma_dgemm` function calls are as follows.

- **MagmaNoTrans**: Specifies no transpose operation for the input matrices.
- **m, n, k**: Dimensions of the matrices involved in the operation, representing rows and columns in a manner similar to the CuBLAS parameters.
- **alpha** and **beta**: Scaling factors for the matrix multiplication and addition operations, respectively.
- **A** and **B**: The input matrices for the multiplication, with their dimensions specified by **lda** and **ldb**.
- **C**: The output matrix, with **ldc** indicating its leading dimension.
- **queue**: A queue to manage execution of the operation, facilitating asynchronous execution and resource management.

MAGMA is the only library among those used that offers a direct function for matrix transposition. The code snippet below illustrates how matrix transposition is performed using MAGMA.

```
// Perform the matrix transposition using MAGMA
if constexpr(std::is_same_v<RealType, float>) {
   magmablas_stranspose(m, n, d_input, m, d_transposed, n, queue);
} else if constexpr(std::is_same_v<RealType, double>) {
   magmablas_dtranspose(m, n, d_input, m, d_transposed, n, queue);
}
```

Key parameters for the `magmablas_stranspose` or `magmablas_dtranspose` function calls, as used in the matrix transposition operations, are outlined below.

- **m** and **n**: Specify the dimensions of the matrix to be transposed. **m** is the number of rows, and **n** is the number of columns in the input matrix.
- **d_input**: Pointer to the input matrix in the device (GPU) memory that is to be transposed.
- **d_transposed**: Pointer to the output matrix in the device memory where the transposed matrix will be stored.
- **queue**: A handle to a queue that manages the execution of the operation. This parameter allows for asynchronous execution and effective resource management, similar to other MAGMA operations.

4.2.3.3 CUTLASS

CUTLASS is a collection of CUDA C++ templates for dense linear algebra operations, emphasizing high performance through customization for various NVIDIA GPUs. It provides a flexible API for configuring matrix multiplication operations (GEMM) according to different data types, layouts, and computing capabilities [17]. CUTLASS's GEMM functionality was integrated, utilizing its template mechanism to ensure optimal performance and compatibility with the benchmarking setup.

```
// Perform matrix multiplication using CUTLASS
using CutlassGemm = cutlass::gemm::device::Gemm< ElementA,
                                                LayoutA,
                                                ElementB,
                                                LayoutB,
                                                ElementC,
                                                LayoutC >;
CutlassGemm gemm_operator;

typename CutlassGemm::Arguments args(
    {m, n, k},
    {matrix1.getValues().getData(), lda},
    {matrix2.getValues().getData(), ldb},
    {resultMatrix.getValues().getData(), ldc},
    {resultMatrix.getValues().getData(), ldc},
    {1.0, 0.0}
);
```

Key parameters for the `cutlass::gemm::device::Gemm` function call are as follows.

- **ElementA, ElementB, ElementC**: Data types for matrices A, B, and C, respectively, allowing for mixed precision operations.
- **LayoutA, LayoutB, LayoutC**: Memory layouts of matrices A, B, and C, which can significantly affect performance.
- **m, n, k**: Define the problem size, similar to other libraries, indicating the matrix dimensions.
- **alpha** and **beta**: Scalars used to scale the operands in the GEMM operation, akin to their roles in CuBLAS and MAGMA.

- **Data pointers**: Pointers to the matrices in GPU memory, specifying the inputs and outputs for the operation.
- **Leading dimensions**: Specified for each matrix, critical for memory access patterns and performance.

4.2.4 Graphical Representation

The graphs below use a logarithmic scale for the y-axis which represents the execution time in seconds. This axis is inversely oriented, meaning higher points indicate shorter execution times and enhanced performance. The x-axis enumerates various matrix size combinations used in the multiplications, providing a spectrum from small to large matrices.

All graphs were generated using a custom `Python` script that employs the `matplotlib` library for plotting. This script processes benchmark log data, which includes execution times for different matrix sizes and algorithms. It reads the log data, organizes them by algorithm, and plots each algorithm's performance across the matrix sizes. The script is designed to highlight the performance differences and trends that are critical for optimizing CUDA-based matrix multiplication. Below is a simplified overview of the script's functionality.

```python
import matplotlib.pyplot as plt
import json

def read_log_files(file_path):
    # Function to read and parse log data
    ...

def process_data_for_plotting(log_entries, algorithms_to_plot=None):
    # Function to organize data for plotting
    ...

def plot_combined_graph(data_for_plotting):
    plt.figure(figsize=(14, 10))
    # Plotting function including labels, legend, and log scale settings
    ...

if __name__ == "__main__":
    log_file_path = 'path_to_log_file.log'
    log_entries = read_log_files(log_file_path)
    data_for_plotting = process_data_for_plotting(log_entries)
    plot_combined_graph(data_for_plotting)
```

This approach ensures that the visualizations are both accurate and informative, providing a clear representation.

4.2.4.1 Speedup Factor

The speedup factor, ranging from 0 to infinity, measures the relative performance improvement of one algorithm over another, calculated by

$$\text{Speedup Factor} = \frac{\text{Execution Time of Base Algorithm}}{\text{Execution Time of Compared Algorithm}}.$$

This metric is used in the tables below to quantify and compare the efficiency of different algorithms against the base algorithm. A speedup factor greater than one indicates that the compared algorithm is faster than the base, whereas a factor less than one shows it is slower.

4.2.5 Dense Matrix Multiplication

This subsection evaluates the performance of our dense matrix multiplication kernels on both HELIOS and GP1. We compare them with cuBLAS, Magma, and Cutlass, demonstrating their computational efficiency in different scenarios.

Each of the graphs presented in this subsection visually illustrates the performance outcomes for different matrix sizes. Directly beneath each graph, corresponding tables detail the execution times and speedup factor versus cuBLAS for selected matrix sizes.

4.2.5.1 HELIOS

The graph below presents the performance of the multiplication kernels of square matrices. The analysis primarily focuses on comparing our kernels against the benchmarking libraries.

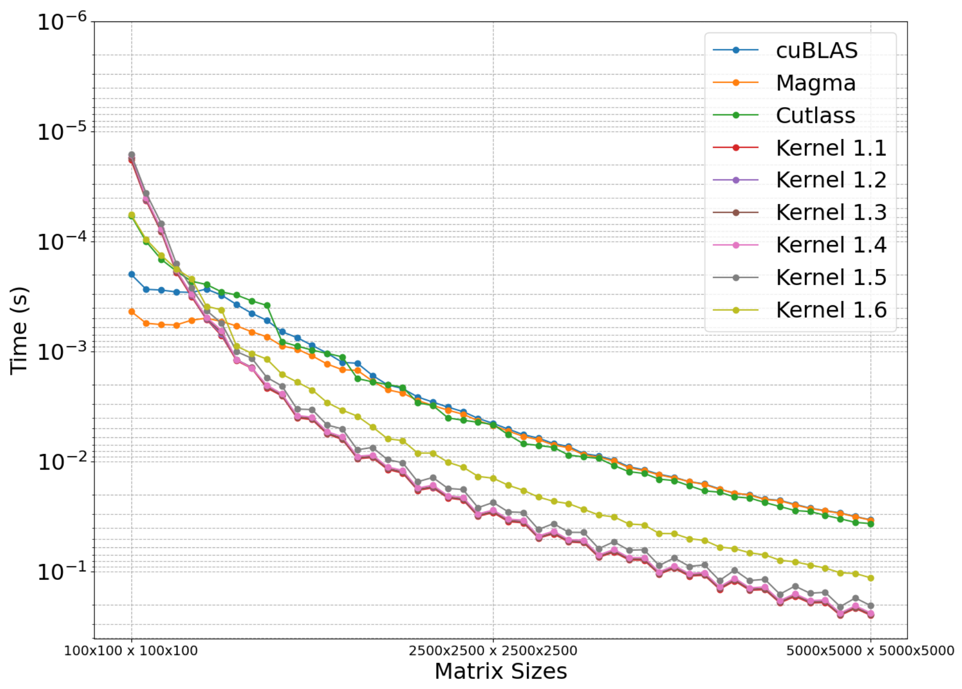


Figure 4.3: Square Matrix Multiplication (HELIOS)

$100x100 \times 100x100$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	1.984529e-04	Base
Magma	4.309488e-04	-
Cutlass	5.808045e-05	-
Kernel 1.1	1.819620e-05	10.91
Kernel 1.2	1.726845e-05	11.49
Kernel 1.3	1.729075e-05	11.48
Kernel 1.4	1.640175e-05	12.10
Kernel 1.5	1.606900e-05	12.35
Kernel 1.6	5.692355e-05	3.49

$1300x1300 \times 1300x1300$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	8.776071e-04	Base
Magma	1.092818e-03	-
Cutlass	9.687090e-04	-
Kernel 1.1	4.144436e-03	0.21
Kernel 1.2	4.051583e-03	0.22
Kernel 1.3	4.050027e-03	0.22
Kernel 1.4	3.933084e-03	0.22
Kernel 1.5	3.358497e-03	0.26
Kernel 1.6	2.226371e-03	0.39

$2500x2500 \times 2500x2500$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	4.491389e-03	Base
Magma	4.693396e-03	-
Cutlass	4.593031e-03	-
Kernel 1.1	2.898556e-02	0.15
Kernel 1.2	2.833413e-02	0.16
Kernel 1.3	2.831249e-02	0.16
Kernel 1.4	2.747724e-02	0.16
Kernel 1.5	2.340984e-02	0.19
Kernel 1.6	1.411427e-02	0.32

$3700x3700 \times 3700x3700$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	1.383658e-02	Base
Magma	1.402894e-02	-
Cutlass	1.481735e-02	-
Kernel 1.1	9.284642e-02	0.15
Kernel 1.2	9.079028e-02	0.15
Kernel 1.3	9.071090e-02	0.15
Kernel 1.4	8.798600e-02	0.16
Kernel 1.5	7.476492e-02	0.19
Kernel 1.6	4.499732e-02	0.31

$5000x5000 \times 5000x5000$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	3.369595e-02	Base
Magma	3.388870e-02	-
Cutlass	3.646766e-02	-
Kernel 1.1	2.475856e-01	0.14
Kernel 1.2	2.427144e-01	0.14
Kernel 1.3	2.423779e-01	0.14
Kernel 1.4	2.354888e-01	0.14
Kernel 1.5	2.021816e-01	0.17
Kernel 1.6	1.130343e-01	0.30

Table 4.5: Execution Times and Speedups versus cuBLAS for Selected Matrix Sizes in Figure 4.3

It is evident from Figure 4.3 that Kernels 1.1 to 1.5 demonstrate a competitive edge over the benchmarking libraries for smaller matrix sizes, particularly up to approximately 1000×1000. However, as the matrix size increases beyond this point, their execution times begin to lengthen. Notably, Kernel 1.5 shows a substantial improvement in performance for larger matrices compared to Kernels 1.1 to 1.4, suggesting effective optimizations through the tile-based approach and the shared memory padding. Among our implementations, Kernel 1.6 emerges as the fastest, especially for the largest matrix sizes tested. This superior performance is attributed to its innovative use of the register blocking technique.

The subsequent figures, 4.4, 4.5 and 4.6, show the performance of CUDA kernels during the multiplication of rectangular matrices, exploring variations with incremental width and height expansions respectively.

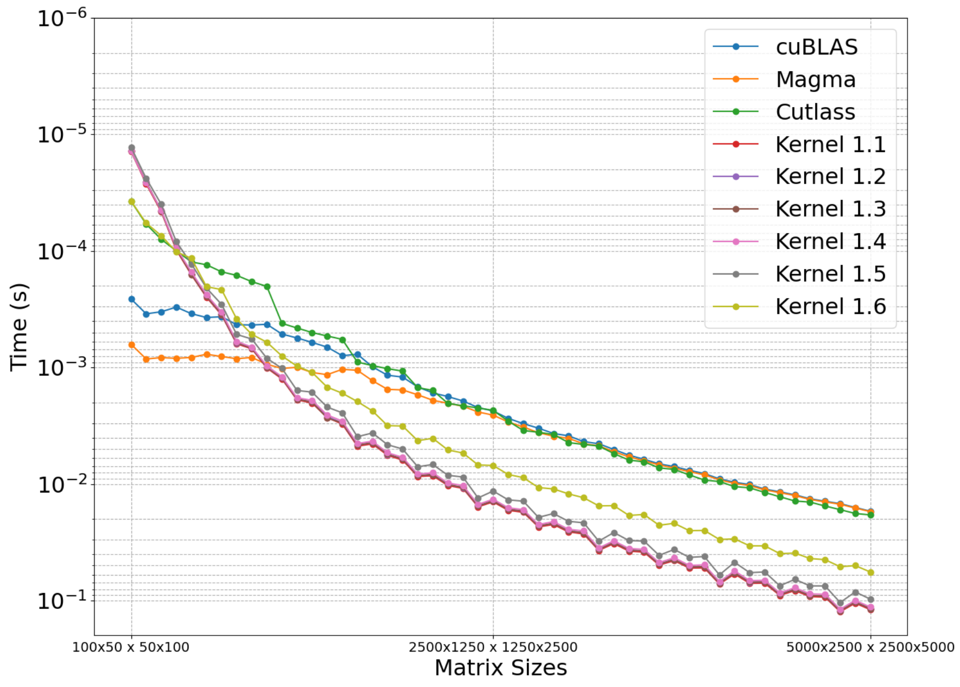


Figure 4.4: Wide Matrix Multiplication with Incremental Width Expansion (HELIOS)

100x50 × 50x100		
Kernel	**Time (s)**	**Speedup**
cuBLAS	2.580858e-04	Base
Magma	6.313587e-04	-
Cutlass	3.743810e-05	-
Kernel 1.1	1.397670e-05	18.47
Kernel 1.2	1.390565e-05	18.56
Kernel 1.3	1.390685e-05	18.56
Kernel 1.4	1.390960e-05	18.56
Kernel 1.5	1.287530e-05	20.05
Kernel 1.6	3.747340e-05	6.89

1300x650 × 650x1300		
Kernel	**Time (s)**	**Speedup**
cuBLAS	6.093547e-04	Base
Magma	1.104328e-03	-
Cutlass	5.013439e-04	-
Kernel 1.1	2.017711e-03	0.30
Kernel 1.2	1.974541e-03	0.31
Kernel 1.3	1.977299e-03	0.31
Kernel 1.4	1.911685e-03	0.32
Kernel 1.5	1.627211e-03	0.37
Kernel 1.6	1.102072e-03	0.55

2500x1250 × 1250x2500		
Kernel	**Time (s)**	**Speedup**
cuBLAS	2.360265e-03	Base
Magma	2.561863e-03	-
Cutlass	2.332852e-03	-
Kernel 1.1	1.421160e-02	0.17
Kernel 1.2	1.389100e-02	0.17
Kernel 1.3	1.389008e-02	0.17
Kernel 1.4	1.349161e-02	0.17
Kernel 1.5	1.147134e-02	0.21
Kernel 1.6	6.928284e-03	0.34

3700x1850 × 1850x3700		
Kernel	**Time (s)**	**Speedup**
cuBLAS	7.081138e-03	Base
Magma	7.290077e-03	-
Cutlass	7.486675e-03	-
Kernel 1.1	4.538069e-02	0.16
Kernel 1.2	4.432301e-02	0.16
Kernel 1.3	4.430013e-02	0.16
Kernel 1.4	4.286869e-02	0.17
Kernel 1.5	3.637342e-02	0.19
Kernel 1.6	2.170774e-02	0.33

5000x2500 × 2500x5000		
Kernel	**Time (s)**	**Speedup**
cuBLAS	1.708849e-02	Base
Magma	1.723424e-02	-
Cutlass	1.839945e-02	-
Kernel 1.1	1.194838e-01	0.14
Kernel 1.2	1.170084e-01	0.15
Kernel 1.3	1.168582e-01	0.15
Kernel 1.4	1.135885e-01	0.15
Kernel 1.5	9.705805e-02	0.18
Kernel 1.6	5.678837e-02	0.30

Table 4.6: Execution Times and Speedups versus cuBLAS for Selected Matrix Sizes in Figure 4.4

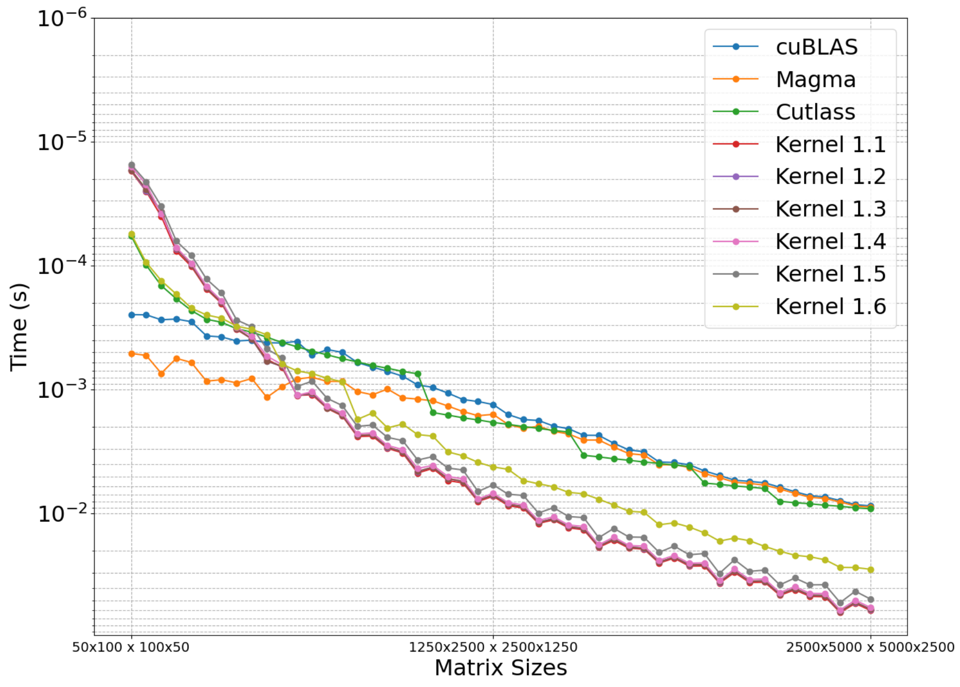


Figure 4.5: Tall Matrix Multiplication with Incremental Height Expansion (HELIOS)

$50x100 \times 100x50$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	2.481409e-04	Base
Magma	5.100682e-04	-
Cutlass	5.709680e-05	-
Kernel 1.1	1.720255e-05	14.42
Kernel 1.2	1.686000e-05	14.72
Kernel 1.3	1.686780e-05	14.71
Kernel 1.4	1.587955e-05	15.63
Kernel 1.5	1.522170e-05	16.30
Kernel 1.6	5.524410e-05	4.49

$650x1300 \times 1300x650$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	5.270255e-04	Base
Magma	7.940489e-04	-
Cutlass	4.885556e-04	-
Kernel 1.1	1.097866e-03	0.48
Kernel 1.2	1.077016e-03	0.49
Kernel 1.3	1.065280e-03	0.49
Kernel 1.4	1.038286e-03	0.51
Kernel 1.5	8.531341e-04	0.62
Kernel 1.6	7.406615e-04	0.71

$1250x2500 \times 2500x1250$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	1.316508e-03	Base
Magma	1.580323e-03	-
Cutlass	1.835531e-03	-
Kernel 1.1	7.236771e-03	0.18
Kernel 1.2	7.073846e-03	0.19
Kernel 1.3	7.066941e-03	0.19
Kernel 1.4	6.846133e-03	0.19
Kernel 1.5	5.851904e-03	0.22
Kernel 1.6	4.200364e-03	0.31

$1850x3700 \times 3700x1850$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	3.863503e-03	Base
Magma	4.021961e-03	-
Cutlass	4.053717e-03	-
Kernel 1.1	2.284243e-02	0.17
Kernel 1.2	2.245615e-02	0.17
Kernel 1.3	2.241748e-02	0.17
Kernel 1.4	2.172074e-02	0.18
Kernel 1.5	1.825795e-02	0.21
Kernel 1.6	1.186037e-02	0.33

$2500x5000 \times 5000x2500$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	8.646138e-03	Base
Magma	8.845528e-03	-
Cutlass	9.122770e-03	-
Kernel 1.1	6.019382e-02	0.14
Kernel 1.2	5.897396e-02	0.15
Kernel 1.3	5.886967e-02	0.15
Kernel 1.4	5.712306e-02	0.15
Kernel 1.5	4.878342e-02	0.18
Kernel 1.6	2.807671e-02	0.31

Table 4.7: Execution Times and Speedups versus cuBLAS for Selected Matrix Sizes in Figure 4.5

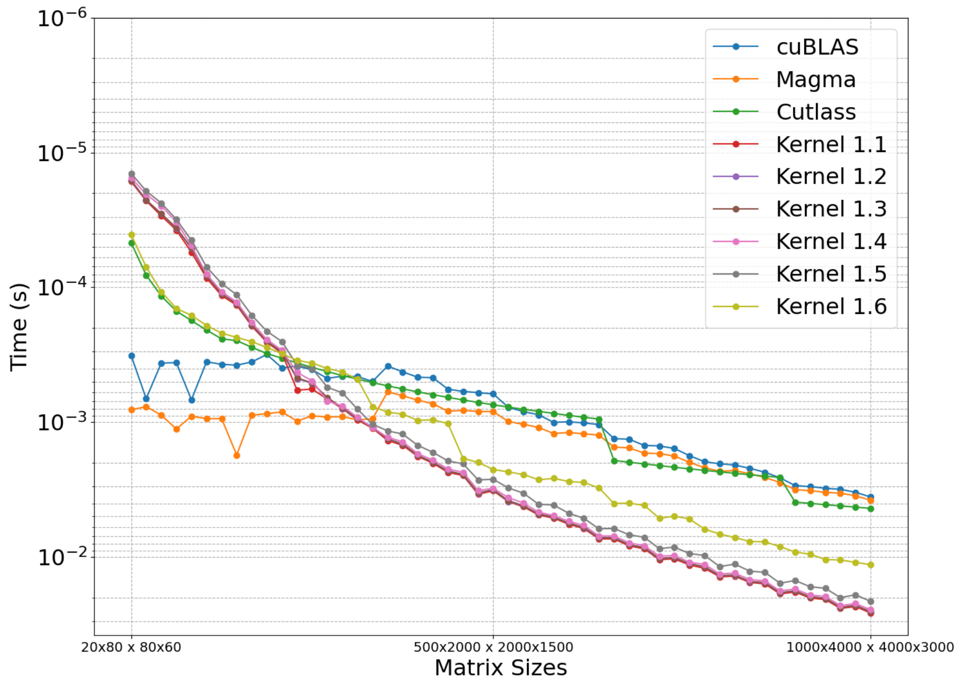


Figure 4.6: Random Growth Matrix Multiplication (HELIOS)

$20x80 \times 80x60$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	3.187681e-04	Base
Magma	8.094483e-04	-
Cutlass	4.654310e-05	-
Kernel 1.1	1.639695e-05	19.44
Kernel 1.2	1.620260e-05	19.67
Kernel 1.3	1.607265e-05	19.83
Kernel 1.4	1.539845e-05	20.70
Kernel 1.5	1.423195e-05	22.40
Kernel 1.6	4.030835e-05	7.91

$260x1040 \times 1040x780$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	4.081724e-04	Base
Magma	8.965029e-04	-
Cutlass	3.920705e-04	-
Kernel 1.1	5.672858e-04	0.72
Kernel 1.2	5.127146e-04	0.80
Kernel 1.3	5.117773e-04	0.80
Kernel 1.4	4.935519e-04	0.83
Kernel 1.5	3.990912e-04	1.02
Kernel 1.6	3.641580e-04	1.12

$500x2000 \times 2000x1500$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	6.173699e-04	Base
Magma	8.339090e-04	-
Cutlass	7.430282e-04	-
Kernel 1.1	3.226742e-03	0.19
Kernel 1.2	3.163629e-03	0.20
Kernel 1.3	3.167241e-03	0.20
Kernel 1.4	3.094795e-03	0.20
Kernel 1.5	2.651307e-03	0.23
Kernel 1.6	2.240721e-03	0.28

$740x2960 \times 2960x2220$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	1.566608e-03	Base
Magma	1.783317e-03	-
Cutlass	2.159620e-03	-
Kernel 1.1	1.031852e-02	0.15
Kernel 1.2	1.008389e-02	0.16
Kernel 1.3	1.007599e-02	0.16
Kernel 1.4	9.740718e-03	0.16
Kernel 1.5	8.437294e-03	0.19
Kernel 1.6	4.979483e-03	0.31

$1000x4000 \times 4000x3000$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	3.587794e-03	Base
Magma	3.788730e-03	-
Cutlass	4.358139e-03	-
Kernel 1.1	2.600388e-02	0.14
Kernel 1.2	2.540506e-02	0.14
Kernel 1.3	2.538266e-02	0.14
Kernel 1.4	2.458283e-02	0.15
Kernel 1.5	2.132744e-02	0.17
Kernel 1.6	1.140153e-02	0.31

Table 4.8: Execution Times and Speedups versus cuBLAS for Selected Matrix Sizes in Figure 4.6

As observed in Figures 4.4, 4.5 and 4.6, the performance trends are similar to those seen in Figure 4.3. Kernel 1.6 consistently outperforms the other kernels for larger matrices.

The following figures illustrate the performance of the kernels when handling extremely thin matrix multiplication.

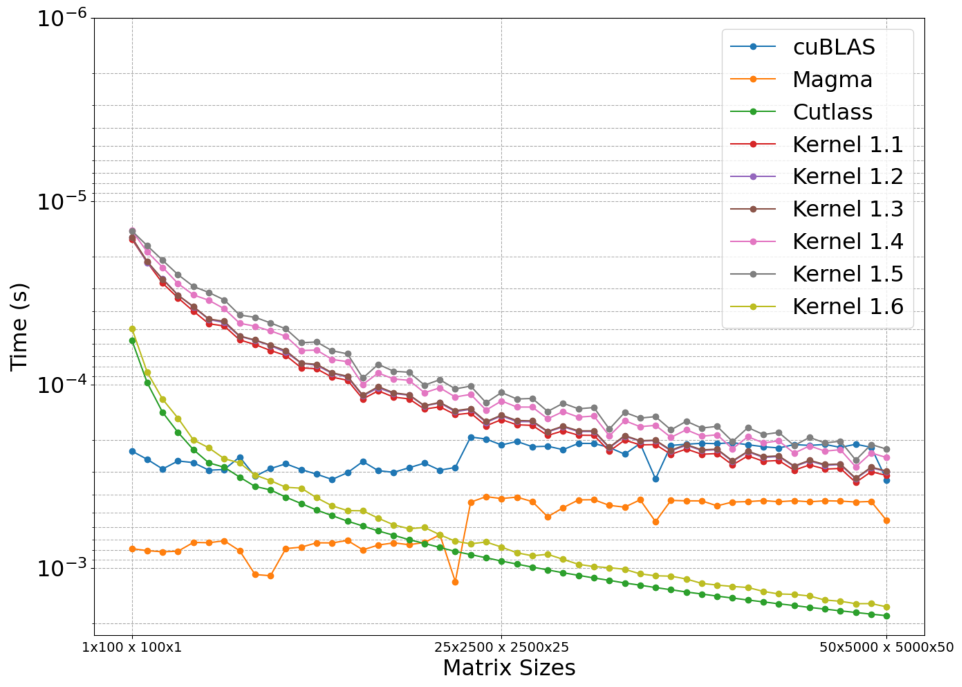


Figure 4.7: Extremely Thin Matrix Multiplication 1 (HELIOS)

$1x100 \times 100x1$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	2.306094e-04	Base
Magma	7.846315e-04	-
Cutlass	5.720200e-05	-
Kernel 1.1	1.609795e-05	14.33
Kernel 1.2	1.577330e-05	14.62
Kernel 1.3	1.577265e-05	14.62
Kernel 1.4	1.446040e-05	15.95
Kernel 1.5	1.450180e-05	15.90
Kernel 1.6	4.935695e-05	4.67

$12x1200 \times 1200x12$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	2.909985e-04	Base
Magma	7.705508e-04	-
Cutlass	4.459676e-04	-
Kernel 1.1	8.109390e-05	3.56
Kernel 1.2	7.647685e-05	3.80
Kernel 1.3	7.650700e-05	3.80
Kernel 1.4	6.523880e-05	4.46
Kernel 1.5	5.914245e-05	4.92
Kernel 1.6	3.674766e-04	0.79

$25x2500 \times 2500x25$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	2.136234e-04	Base
Magma	4.188335e-04	-
Cutlass	9.179343e-04	-
Kernel 1.1	1.553242e-04	1.38
Kernel 1.2	1.482717e-04	1.44
Kernel 1.3	1.469067e-04	1.45
Kernel 1.4	1.228667e-04	1.74
Kernel 1.5	1.105505e-04	1.93
Kernel 1.6	7.698886e-04	0.28

$36x3600 \times 3600x36$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	2.148083e-04	Base
Magma	4.284218e-04	-
Cutlass	1.315318e-03	-
Kernel 1.1	2.402200e-04	0.89
Kernel 1.2	2.299845e-04	0.93
Kernel 1.3	2.281640e-04	0.94
Kernel 1.4	1.941776e-04	1.11
Kernel 1.5	1.766797e-04	1.22
Kernel 1.6	1.109534e-03	0.19

$50x5000 \times 5000x50$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	3.310832e-04	Base
Magma	5.512668e-04	-
Cutlass	1.822532e-03	-
Kernel 1.1	3.131439e-04	1.05
Kernel 1.2	2.985052e-04	1.11
Kernel 1.3	2.964971e-04	1.12
Kernel 1.4	2.501058e-04	1.32
Kernel 1.5	2.250437e-04	1.47
Kernel 1.6	1.630874e-03	0.20

Table 4.9: Execution Times and Speedups versus cuBLAS for Selected Matrix Sizes in Figure 4.7

In Figure 4.7, Kernels 1.1 to 1.5 outperform the benchmarking libraries for the outer product of two extremely thin matrices, with Kernel 1.5 leading in execution speed and being even faster then the benchmarking libraries. Notably, Kernel 1.6, which generally excels in larger matrix configurations, shows decreased performance in this scenario.

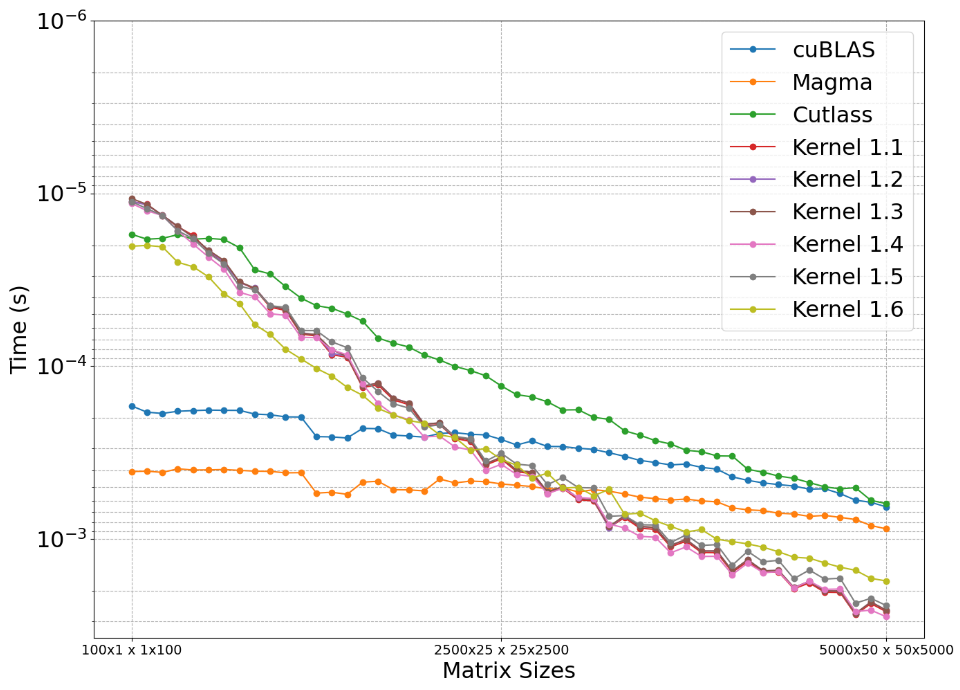


Figure 4.8: Extremely Thin Matrix Multiplication 2 (HELIOS)

$100x1 \times 1x100$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	1.697587e-04	Base
Magma	4.073414e-04	-
Cutlass	1.726265e-05	-
Kernel 1.1	1.117585e-05	15.19
Kernel 1.2	1.112365e-05	15.26
Kernel 1.3	1.071265e-05	15.85
Kernel 1.4	1.143750e-05	14.84
Kernel 1.5	1.116295e-05	15.21
Kernel 1.6	2.022535e-05	8.39

$1200x12 \times 12x1200$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	1.968918e-04	Base
Magma	4.132536e-04	-
Cutlass	4.037955e-05	-
Kernel 1.1	6.517600e-05	3.02
Kernel 1.2	6.411310e-05	3.07
Kernel 1.3	6.462995e-05	3.05
Kernel 1.4	6.803665e-05	2.89
Kernel 1.5	6.217610e-05	3.17
Kernel 1.6	9.079455e-05	2.17

$2500x25 \times 25x2500$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	2.656855e-04	Base
Magma	4.802173e-04	-
Cutlass	1.300928e-04	-
Kernel 1.1	3.438627e-04	0.77
Kernel 1.2	3.366878e-04	0.79
Kernel 1.3	3.368511e-04	0.79
Kernel 1.4	3.697045e-04	0.72
Kernel 1.5	3.190552e-04	0.83
Kernel 1.6	3.470997e-04	0.77

$3600x36 \times 36x3600$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	3.728953e-04	Base
Magma	5.948661e-04	-
Cutlass	2.821611e-04	-
Kernel 1.1	1.110312e-03	0.34
Kernel 1.2	1.088539e-03	0.34
Kernel 1.3	1.088801e-03	0.34
Kernel 1.4	1.202460e-03	0.31
Kernel 1.5	1.049321e-03	0.36
Kernel 1.6	8.453369e-04	0.44

$5000x50 \times 50x5000$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	6.520912e-04	Base
Magma	8.741802e-04	-
Cutlass	6.242091e-04	-
Kernel 1.1	2.636753e-03	0.25
Kernel 1.2	2.583512e-03	0.25
Kernel 1.3	2.585248e-03	0.25
Kernel 1.4	2.819973e-03	0.23
Kernel 1.5	2.429158e-03	0.27
Kernel 1.6	1.749444e-03	0.37

Table 4.10: Execution Times and Speedups versus cuBLAS for Selected Matrix Sizes in Figure 4.8

Figure 4.8 shows that Kernels 1.1 to 1.5 maintain superior performance over Kernel 1.6 in multiplying matrices up to $2500x25 \times 25x2500$. As matrix sizes increase, Kernel 1.6 slightly improves its relative speed. Notably, Cutlass exhibits the best performance overall, while our kernels remain competitive with Magma and cuBLAS for a significant portion of the tested matrix sizes.

Overall, on the device specified in 4.4, our multiplication kernels showcased competitive and adaptable performance across diverse matrix configurations. Kernels 1.1 to 1.5 excelled for smaller matrices up to 1000x1000, while Kernel 1.6 dominated in larger matrix scenarios. Notably, Kernel 1.5 outperformed benchmarking libraries in extremely thin matrix tests, highlighting the effectiveness of our optimization strategies across varying computational demands.

4.2.5.2 GP1

The graphs below illustrate the performance of our multiplication kernels on the GP1 device, providing a comparative analysis against the HELIOS device.

Figures 4.9 to 4.12 present the results for square and various rectangular matrix configurations (wide, tall, and random growth), respectively.

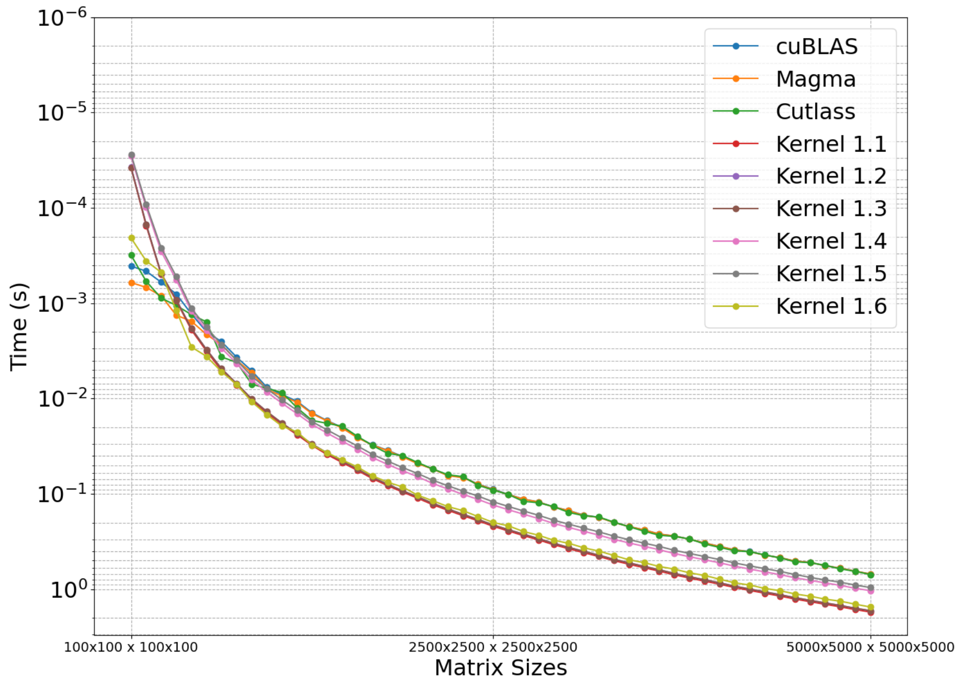


Figure 4.9: Square Matrix Multiplication (GP1)

$100x100 \times 100x100$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	4.081895e-04	Base
Magma	6.065724e-04	-
Cutlass	3.129970e-04	-
Kernel 1.1	3.759330e-05	10.86
Kernel 1.2	3.698110e-05	11.04
Kernel 1.3	3.773420e-05	10.82
Kernel 1.4	2.852165e-05	14.31
Kernel 1.5	2.729625e-05	14.95
Kernel 1.6	2.065424e-04	1.98

$1300x1300 \times 1300x1300$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	1.413625e-02	Base
Magma	1.443936e-02	-
Cutlass	1.696084e-02	-
Kernel 1.1	3.106995e-02	0.45
Kernel 1.2	3.009823e-02	0.47
Kernel 1.3	3.010516e-02	0.47
Kernel 1.4	1.868727e-02	0.76
Kernel 1.5	1.750033e-02	0.81
Kernel 1.6	3.075797e-02	0.46

$2500x2500 \times 2500x2500$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	8.954067e-02	Base
Magma	9.055234e-02	-
Cutlass	9.124815e-02	-
Kernel 1.1	2.186193e-01	0.41
Kernel 1.2	2.117630e-01	0.42
Kernel 1.3	2.116808e-01	0.42
Kernel 1.4	1.310077e-01	0.68
Kernel 1.5	1.210994e-01	0.74
Kernel 1.6	1.996307e-01	0.45

$3700x3700 \times 3700x3700$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	2.762528e-01	Base
Magma	2.766087e-01	-
Cutlass	2.793153e-01	-
Kernel 1.1	7.056310e-01	0.39
Kernel 1.2	6.834065e-01	0.40
Kernel 1.3	6.836959e-01	0.40
Kernel 1.4	4.220297e-01	0.65
Kernel 1.5	3.903197e-01	0.71
Kernel 1.6	6.158302e-01	0.45

$5000x5000 \times 5000x5000$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	6.929490e-01	Base
Magma	6.933283e-01	-
Cutlass	7.022150e-01	-
Kernel 1.1	1.735417e+00	0.40
Kernel 1.2	1.681039e+00	0.41
Kernel 1.3	1.680823e+00	0.41
Kernel 1.4	1.036679e+00	0.67
Kernel 1.5	9.616447e-01	0.72
Kernel 1.6	1.539299e+00	0.45

Table 4.11: Execution Times and Speedups versus cuBLAS for Selected Matrix Sizes in Figure 4.9

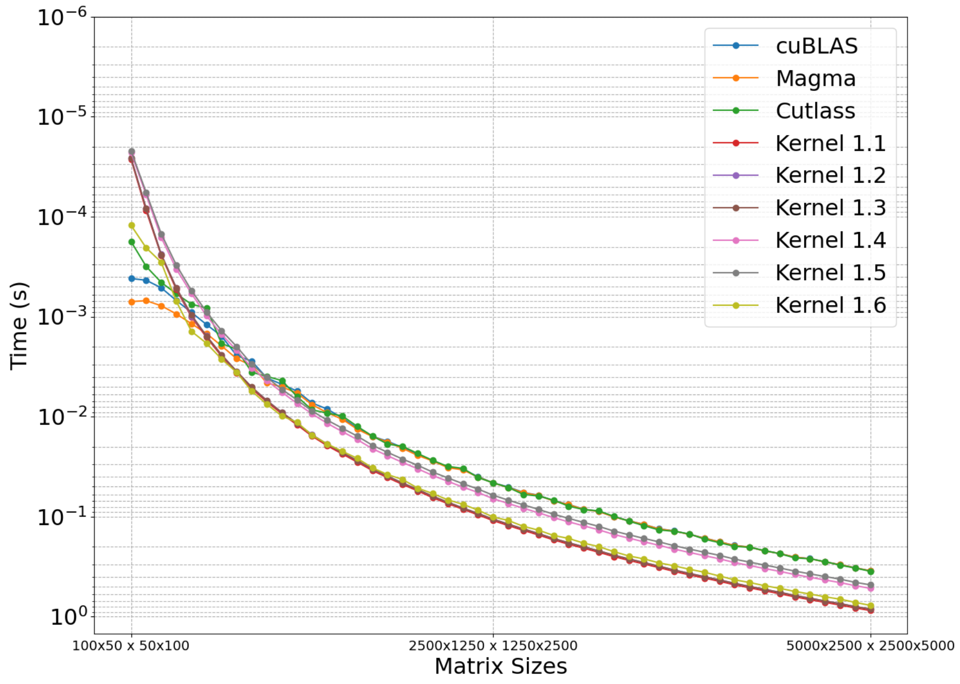


Figure 4.10: Wide Matrix Multiplication with Incremental Width Expansion (GP1)

100x50 × 50x100		
Kernel	**Time (s)**	**Speedup**
cuBLAS	4.149670e-04	Base
Magma	7.060486e-04	-
Cutlass	1.770556e-04	-
Kernel 1.1	2.655980e-05	15.62
Kernel 1.2	2.580260e-05	16.08
Kernel 1.3	2.593020e-05	16.00
Kernel 1.4	2.275040e-05	18.24
Kernel 1.5	2.191160e-05	18.94
Kernel 1.6	1.218854e-04	3.40

1300x650 × 650x1300		
Kernel	**Time (s)**	**Speedup**
cuBLAS	7.301026e-03	Base
Magma	7.586739e-03	-
Cutlass	8.555414e-03	-
Kernel 1.1	1.565428e-02	0.47
Kernel 1.2	1.516158e-02	0.48
Kernel 1.3	1.520322e-02	0.48
Kernel 1.4	9.374832e-03	0.78
Kernel 1.5	8.779180e-03	0.83
Kernel 1.6	1.535488e-02	0.48

2500x1250 × 1250x2500		
Kernel	**Time (s)**	**Speedup**
cuBLAS	4.591308e-02	Base
Magma	4.597054e-02	-
Cutlass	4.585048e-02	-
Kernel 1.1	1.094561e-01	0.42
Kernel 1.2	1.060087e-01	0.42
Kernel 1.3	1.059902e-01	0.42
Kernel 1.4	6.605146e-02	0.70
Kernel 1.5	6.103240e-02	0.75
Kernel 1.6	1.008036e-01	0.46

3700x1850 × 1850x3700		
Kernel	**Time (s)**	**Speedup**
cuBLAS	1.387915e-01	Base
Magma	1.391940e-01	-
Cutlass	1.400576e-01	-
Kernel 1.1	3.528659e-01	0.39
Kernel 1.2	3.418122e-01	0.41
Kernel 1.3	3.418136e-01	0.41
Kernel 1.4	2.110148e-01	0.66
Kernel 1.5	1.951196e-01	0.71
Kernel 1.6	3.100027e-01	0.45

5000x2500 × 2500x5000		
Kernel	**Time (s)**	**Speedup**
cuBLAS	3.473582e-01	Base
Magma	3.477916e-01	-
Cutlass	3.517793e-01	-
Kernel 1.1	8.676185e-01	0.40
Kernel 1.2	8.405639e-01	0.41
Kernel 1.3	8.404142e-01	0.41
Kernel 1.4	5.197686e-01	0.67
Kernel 1.5	4.805611e-01	0.72
Kernel 1.6	7.719532e-01	0.45

Table 4.12: Execution Times and Speedups versus cuBLAS for Selected Matrix Sizes in Figure 4.10

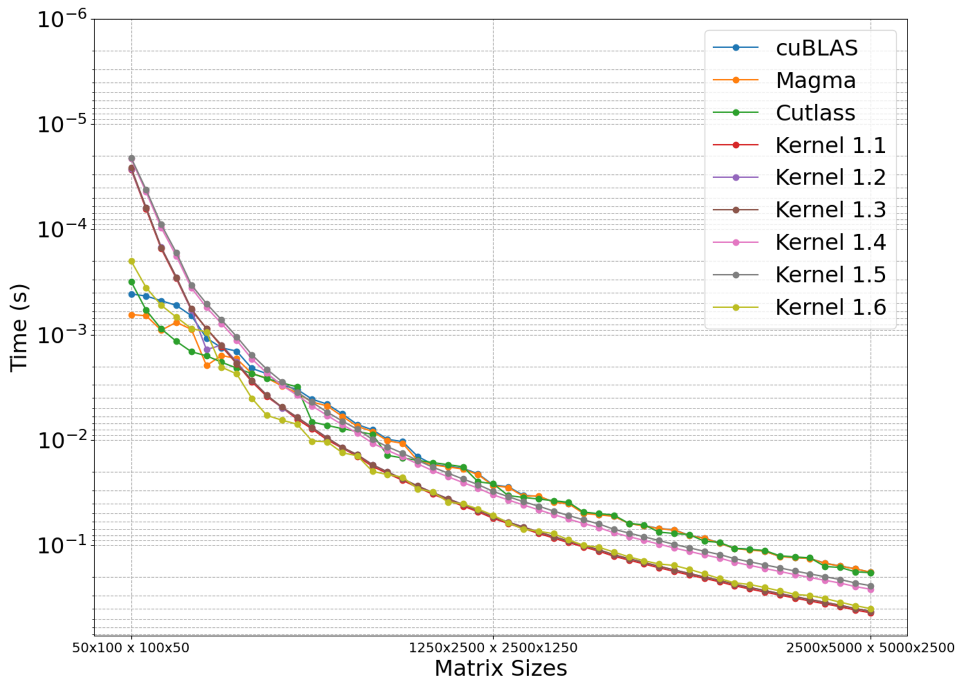


Figure 4.11: Tall Matrix Multiplication with Incremental Height Expansion (GP1)

50x100 × 100x50		
Kernel	**Time (s)**	**Speedup**
cuBLAS	4.107780e-04	Base
Magma	6.443246e-04	-
Cutlass	3.122314e-04	-
Kernel 1.1	2.691720e-05	15.26
Kernel 1.2	2.659260e-05	15.45
Kernel 1.3	2.580000e-05	15.92
Kernel 1.4	2.132940e-05	19.26
Kernel 1.5	2.082000e-05	19.73
Kernel 1.6	1.992544e-04	2.06

650x1300 × 1300x650		
Kernel	**Time (s)**	**Speedup**
cuBLAS	4.100579e-03	Base
Magma	4.355892e-03	-
Cutlass	6.738631e-03	-
Kernel 1.1	7.861985e-03	0.52
Kernel 1.2	7.614052e-03	0.54
Kernel 1.3	7.624921e-03	0.54
Kernel 1.4	4.738672e-03	0.87
Kernel 1.5	4.383328e-03	0.94
Kernel 1.6	1.023081e-02	0.40

1250x2500 × 2500x1250		
Kernel	**Time (s)**	**Speedup**
cuBLAS	2.676374e-02	Base
Magma	2.712074e-02	-
Cutlass	2.600989e-02	-
Kernel 1.1	5.542298e-02	0.48
Kernel 1.2	5.368690e-02	0.50
Kernel 1.3	5.372770e-02	0.50
Kernel 1.4	3.320181e-02	0.81
Kernel 1.5	3.077677e-02	0.87
Kernel 1.6	5.186124e-02	0.52

1850x3700 × 3700x1850		
Kernel	**Time (s)**	**Speedup**
cuBLAS	7.122932e-02	Base
Magma	7.161114e-02	-
Cutlass	7.718647e-02	-
Kernel 1.1	1.770758e-01	0.40
Kernel 1.2	1.715653e-01	0.42
Kernel 1.3	1.715790e-01	0.42
Kernel 1.4	1.060529e-01	0.67
Kernel 1.5	9.804100e-02	0.73
Kernel 1.6	1.554374e-01	0.46

2500x5000 × 5000x2500		
Kernel	**Time (s)**	**Speedup**
cuBLAS	1.807691e-01	Base
Magma	1.798234e-01	-
Cutlass	1.820425e-01	-
Kernel 1.1	4.369935e-01	0.41
Kernel 1.2	4.233834e-01	0.43
Kernel 1.3	4.231624e-01	0.43
Kernel 1.4	2.612904e-01	0.69
Kernel 1.5	2.416050e-01	0.75
Kernel 1.6	3.984211e-01	0.45

Table 4.13: Execution Times and Speedups versus cuBLAS for Selected Matrix Sizes in Figure 4.11

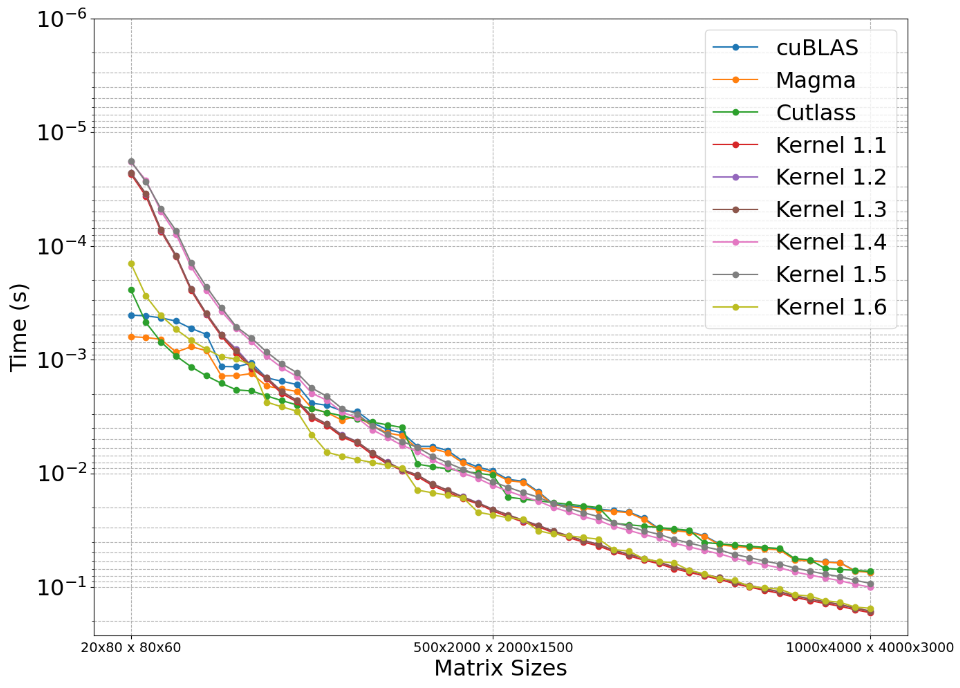


Figure 4.12: Random Growth Matrix Multiplication (GP1)

$20x80 \times 80x60$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	4.063626e-04	Base
Magma	6.290272e-04	-
Cutlass	2.432350e-04	-
Kernel 1.1	2.327800e-05	17.46
Kernel 1.2	2.261660e-05	17.97
Kernel 1.3	2.280720e-05	17.82
Kernel 1.4	1.830900e-05	22.19
Kernel 1.5	1.789160e-05	22.71
Kernel 1.6	1.420094e-04	2.86

$260x1040 \times 1040x780$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	2.416288e-03	Base
Magma	2.682960e-03	-
Cutlass	2.711322e-03	-
Kernel 1.1	3.272803e-03	0.74
Kernel 1.2	3.149111e-03	0.77
Kernel 1.3	3.148750e-03	0.77
Kernel 1.4	1.980184e-03	1.22
Kernel 1.5	1.781123e-03	1.36
Kernel 1.6	4.566019e-03	0.53

$500x2000 \times 2000x1500$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	9.579462e-03	Base
Magma	9.882397e-03	-
Cutlass	1.040493e-02	-
Kernel 1.1	2.144290e-02	0.45
Kernel 1.2	2.077706e-02	0.46
Kernel 1.3	2.077181e-02	0.46
Kernel 1.4	1.280077e-02	0.75
Kernel 1.5	1.191610e-02	0.80
Kernel 1.6	2.326026e-02	0.41

$740x2960 \times 2960x2220$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	3.159405e-02	Base
Magma	3.194780e-02	-
Cutlass	3.083724e-02	-
Kernel 1.1	6.904851e-02	0.46
Kernel 1.2	6.676215e-02	0.47
Kernel 1.3	6.676156e-02	0.47
Kernel 1.4	4.117048e-02	0.77
Kernel 1.5	3.805485e-02	0.83
Kernel 1.6	6.135010e-02	0.51

$1000x4000 \times 4000x3000$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	7.409793e-02	Base
Magma	7.448496e-02	-
Cutlass	7.285081e-02	-
Kernel 1.1	1.681367e-01	0.44
Kernel 1.2	1.626889e-01	0.46
Kernel 1.3	1.629660e-01	0.46
Kernel 1.4	1.003698e-01	0.74
Kernel 1.5	9.275250e-02	0.80
Kernel 1.6	1.537992e-01	0.48

Table 4.14: Execution Times and Speedups versus cuBLAS for Selected Matrix Sizes in Figure 4.12

The figures 4.9 to 4.12 exhibit very similar performance, warranting a combined analysis. Unlike on HELIOS, Kernel 1.6 is not the best performer on GP1. Instead, Kernels 1.4 and 1.5 lead our implementations, outperforming benchmarking libraries for smaller matrices and maintaining competitive performance for larger matrices. Kernel 1.5, with added padding, consistently surpasses Kernel 1.4.

Below are Figures 4.13 and 4.14 illustrating the performance for extremely thin matrices.

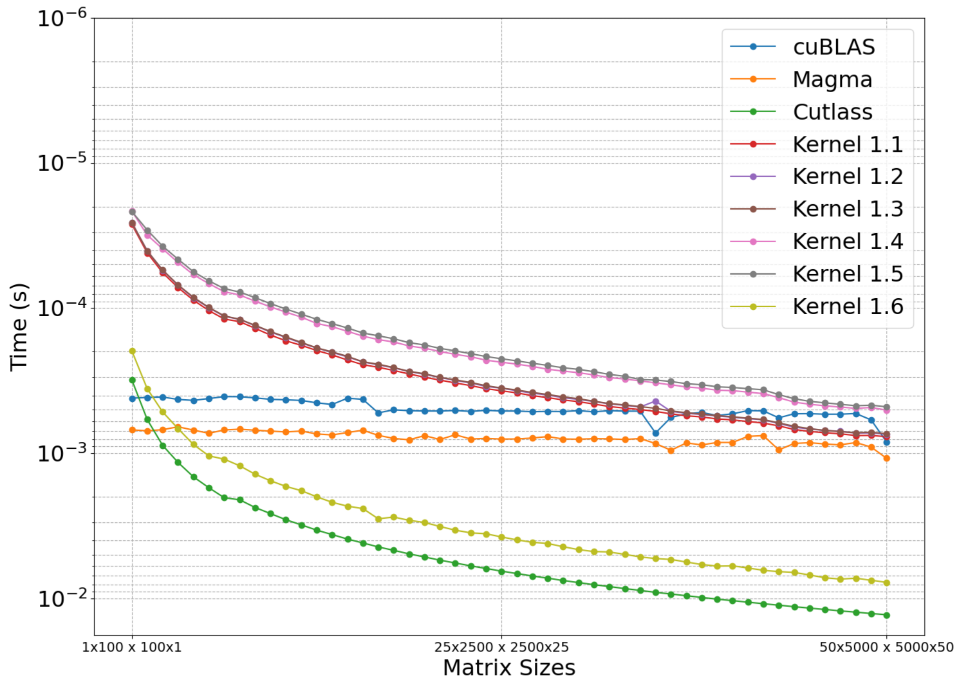


Figure 4.13: Extremely Thin Matrix Multiplication 1 (GP1)

$1x100 \times 100x1$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	4.174446e-04	Base
Magma	6.911668e-04	-
Cutlass	3.124926e-04	-
Kernel 1.1	2.649900e-05	15.75
Kernel 1.2	2.580800e-05	16.18
Kernel 1.3	2.575120e-05	16.21
Kernel 1.4	2.146280e-05	19.45
Kernel 1.5	2.161760e-05	19.31
Kernel 1.6	1.969714e-04	2.12

$12x1200 \times 1200x12$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	4.329318e-04	Base
Magma	7.036666e-04	-
Cutlass	3.117458e-03	-
Kernel 1.1	1.806498e-04	2.40
Kernel 1.2	1.742016e-04	2.49
Kernel 1.3	1.722254e-04	2.51
Kernel 1.4	1.154104e-04	3.75
Kernel 1.5	1.101730e-04	3.93
Kernel 1.6	1.811358e-03	0.24

$25x2500 \times 2500x25$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	5.121798e-04	Base
Magma	7.989602e-04	-
Cutlass	6.497798e-03	-
Kernel 1.1	3.714784e-04	1.38
Kernel 1.2	3.564402e-04	1.44
Kernel 1.3	3.557368e-04	1.44
Kernel 1.4	2.360404e-04	2.17
Kernel 1.5	2.235238e-04	2.29
Kernel 1.6	3.777255e-03	0.14

$36x3600 \times 3600x36$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	5.671834e-04	Base
Magma	9.544672e-04	-
Cutlass	9.333862e-03	-
Kernel 1.1	5.358032e-04	1.06
Kernel 1.2	5.151700e-04	1.10
Kernel 1.3	5.111838e-04	1.11
Kernel 1.4	3.377816e-04	1.68
Kernel 1.5	3.206178e-04	1.77
Kernel 1.6	5.394775e-03	0.11

$50x5000 \times 5000x50$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	8.364764e-04	Base
Magma	1.084482e-03	-
Cutlass	1.297283e-02	-
Kernel 1.1	7.696240e-04	1.09
Kernel 1.2	7.404746e-04	1.13
Kernel 1.3	7.341410e-04	1.14
Kernel 1.4	5.045064e-04	1.66
Kernel 1.5	4.789112e-04	1.75
Kernel 1.6	7.755272e-03	0.11

Table 4.15: Execution Times and Speedups versus cuBLAS for Selected Matrix Sizes in Figure 4.13

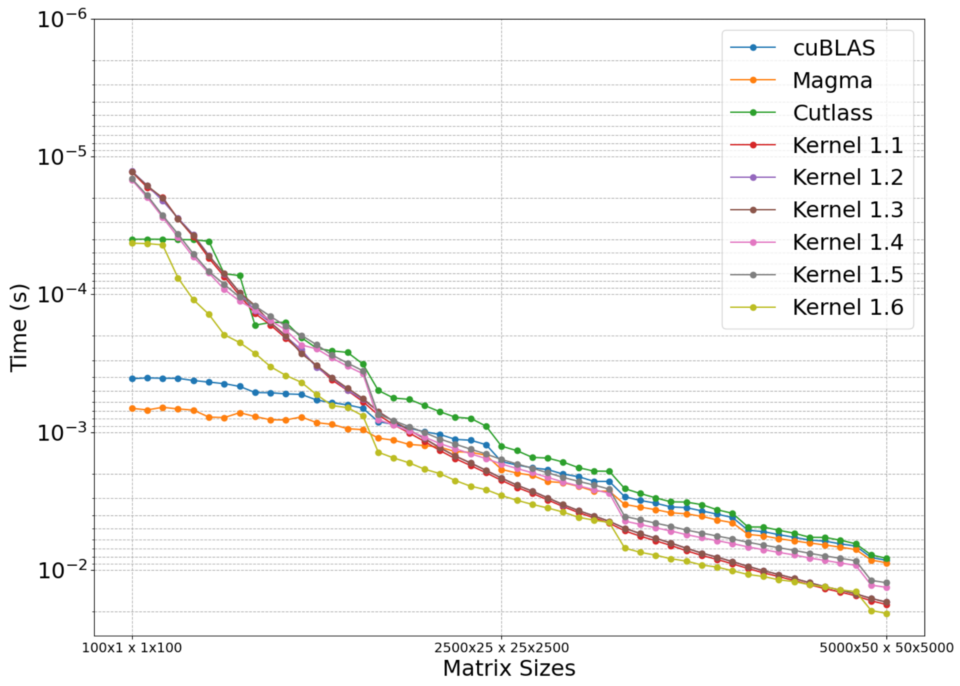


Figure 4.14: Extremely Thin Matrix Multiplication 2 (GP1)

$100x1 \times 1x100$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	4.092408e-04	Base
Magma	6.709350e-04	-
Cutlass	3.981080e-05	-
Kernel 1.1	1.291980e-05	31.68
Kernel 1.2	1.275700e-05	32.08
Kernel 1.3	1.289480e-05	31.74
Kernel 1.4	1.475120e-05	27.74
Kernel 1.5	1.449440e-05	28.23
Kernel 1.6	4.254640e-05	9.62

$1200x12 \times 12x1200$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	5.315040e-04	Base
Magma	7.750456e-04	-
Cutlass	2.056302e-04	-
Kernel 1.1	2.610522e-04	2.04
Kernel 1.2	2.516972e-04	2.11
Kernel 1.3	2.686534e-04	1.98
Kernel 1.4	2.326226e-04	2.28
Kernel 1.5	1.978738e-04	2.69
Kernel 1.6	4.342042e-04	1.22

$2500x25 \times 25x2500$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	1.635547e-03	Base
Magma	1.861891e-03	-
Cutlass	1.262913e-03	-
Kernel 1.1	2.238492e-03	0.73
Kernel 1.2	2.152749e-03	0.76
Kernel 1.3	2.152854e-03	0.76
Kernel 1.4	1.702564e-03	0.96
Kernel 1.5	1.577944e-03	1.04
Kernel 1.6	2.884250e-03	0.57

$3600x36 \times 36x3600$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	3.487033e-03	Base
Magma	3.840732e-03	-
Cutlass	3.202283e-03	-
Kernel 1.1	6.637966e-03	0.53
Kernel 1.2	6.371678e-03	0.55
Kernel 1.3	6.371485e-03	0.55
Kernel 1.4	5.210599e-03	0.67
Kernel 1.5	4.822324e-03	0.72
Kernel 1.6	8.291719e-03	0.42

$5000x50 \times 50x5000$		
Kernel	**Time (s)**	**Speedup**
cuBLAS	8.505293e-03	Base
Magma	8.867052e-03	-
Cutlass	8.220526e-03	-
Kernel 1.1	1.779696e-02	0.42
Kernel 1.2	1.709922e-02	0.50
Kernel 1.3	1.709909e-02	0.50
Kernel 1.4	1.337232e-02	0.64
Kernel 1.5	1.236803e-02	0.69
Kernel 1.6	2.063878e-02	0.41

Table 4.16: Execution Times and Speedups versus cuBLAS for Selected Matrix Sizes in Figure 4.14

Figure 4.13 is remarkable because it demonstrates that Kernels 1.4 and 1.5 outperform not only our implementations, but also the benchmarking libraries. Figure 4.14 further highlights the advantage of Kernels 1.4 and 1.5, which are faster for smaller matrices and maintain competitiveness with the benchmarking libraries for larger sizes.

Overall, on the GP1 device, Kernel 1.5 emerges as the fastest, outperforming the other kernels and benchmarking libraries in most scenarios. This highlights the effectiveness of our optimization strategies, particularly for extremely thin matrices. Kernel 1.5's consistent performance demonstrates its adaptability and efficiency across various matrix configurations. In comparison to the HELIOS device, Kernel 1.5 on the GP1 device shows a significant improvement, being more competitive with the benchmarking libraries and often surpassing them in performance, especially for smaller and extremely thin matrices. This difference underscores the impact of architectural variations, as Kernel 1.6, which utilizes register blocking, was the best on HELIOS. The GP1 architecture appears to handle these optimizations differently, favoring the tiling technique employed in Kernel 1.5.

4.2.6 Dense Matrix Transposition

This subsection evaluates the performance of our dense matrix transposition kernels against Magma on both the HELIOS and GP1. Each graph in this subsection visually illustrates the performance of different kernels across a range of matrix sizes. Beneath each graph, tables are provided that detail the execution times and the speedup factors versus Magma for selected matrix sizes.

4.2.6.1 HELIOS

The graph below evaluates the performance of our transposition kernels, including Kernel 2.4 for in-place transposition, against the benchmarking libraries during the transposition of square matrices.

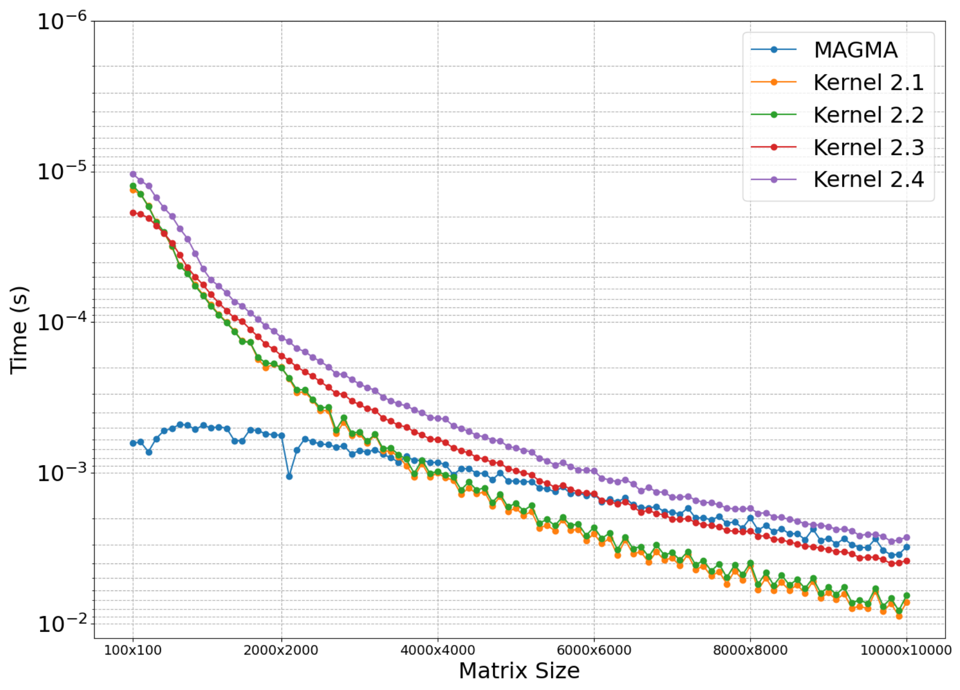


Figure 4.15: Square Matrix Transposition (HELIOS)

100x100		
Kernel	**Time (s)**	**Speedup**
MAGMA	6.360937e-04	Base
Kernel 2.1	1.326925e-05	47.94
Kernel 2.2	1.236090e-05	51.46
Kernel 2.3	1.880590e-05	33.82
Kernel 2.4	1.037435e-05	61.31

5000x5000		
Kernel	**Time (s)**	**Speedup**
MAGMA	1.133390e-03	Base
Kernel 2.1	1.714383e-03	0.66
Kernel 2.2	1.591544e-03	0.71
Kernel 2.3	9.660831e-04	1.17
Kernel 2.4	6.838526e-04	1.66

10000x10000		
Kernel	**Time (s)**	**Speedup**
MAGMA	3.093129e-03	Base
Kernel 2.1	7.205122e-03	0.43
Kernel 2.2	6.498019e-03	0.48
Kernel 2.3	3.821392e-03	0.81
Kernel 2.4	2.667357e-03	1.16

Table 4.17: Execution Times and Speedups versus MAGMA for Selected Matrix Sizes in Figure 4.15

Figure 4.15 demonstrates that all kernels significantly outperform Magma for matrices up to approximately 4000×4000. Beyond this size, Kernels 2.1 and 2.2 experience a slowdown, whereas Kernels 2.3 and 2.4 maintain competitive performance, with Kernel 2.4 slightly outpacing Magma, highlighting the efficiency of its in-place transposition implementation.

Subsequent figures, 4.16 and 4.17, illustrate the performance of rectangular matrices.

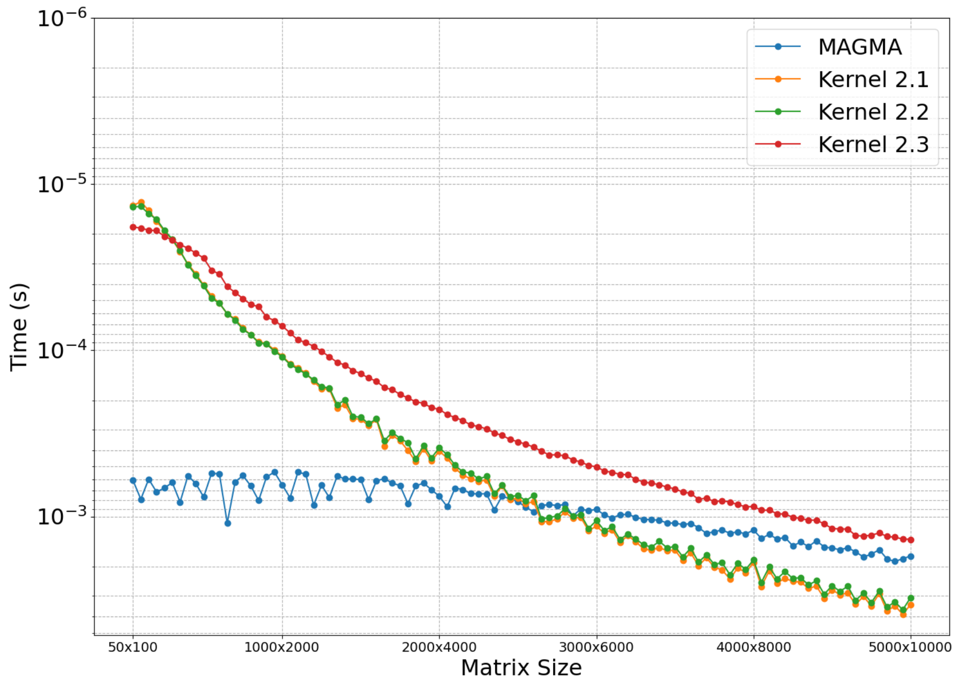


Figure 4.16: Wide Matrix Transposition with Incremental Width Expansion (HELIOS)

50*x*100		
Kernel	**Time (s)**	**Speedup**
MAGMA	6.054758e-04	Base
Kernel 2.1	1.348820e-05	44.89
Kernel 2.2	1.374270e-05	44.06
Kernel 2.3	1.807510e-05	33.49

2500*x*5000		
Kernel	**Time (s)**	**Speedup**
MAGMA	8.113358e-04	Base
Kernel 2.1	7.742828e-04	1.05
Kernel 2.2	7.429002e-04	1.09
Kernel 2.3	3.551663e-04	2.28

5000*x*10000		
Kernel	**Time (s)**	**Speedup**
MAGMA	1.725197e-03	Base
Kernel 2.1	3.379470e-03	0.51
Kernel 2.2	3.066386e-03	0.56
Kernel 2.3	1.371727e-03	1.26

Table 4.18: Execution Times and Speedups versus MAGMA for Selected Matrix Sizes in Figure 4.16

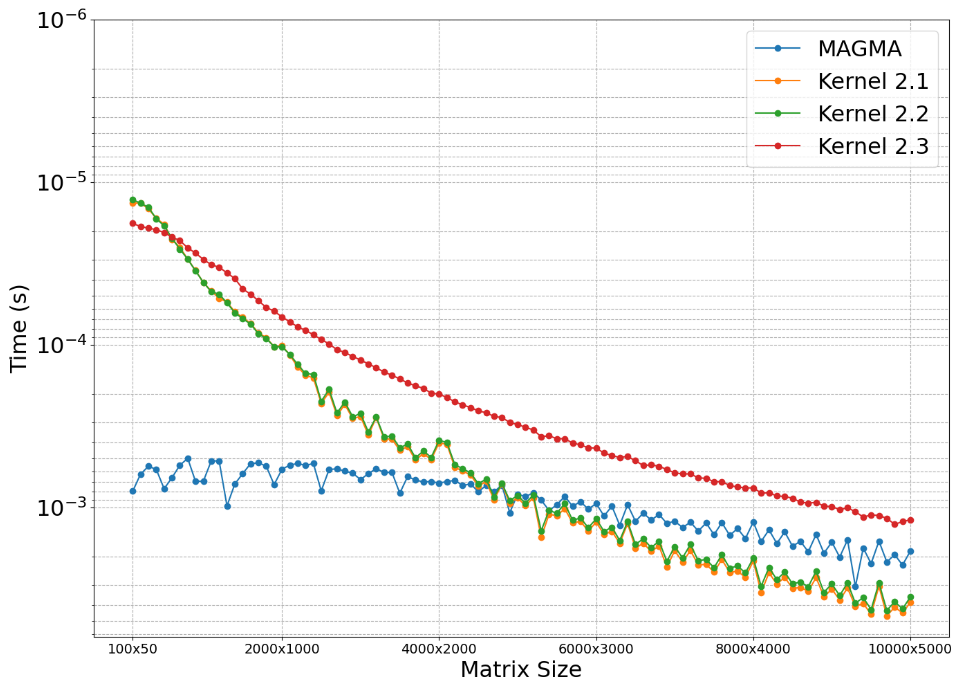


Figure 4.17: Tall Matrix Transposition with Incremental Height Expansion (HELIOS)

100*x*50		
Kernel	**Time (s)**	**Speedup**
MAGMA	7.957498e-04	Base
Kernel 2.1	1.347945e-05	59.03
Kernel 2.2	1.281065e-05	62.11
Kernel 2.3	1.792200e-05	44.40

5000*x*2500		
Kernel	**Time (s)**	**Speedup**
MAGMA	8.287412e-04	Base
Kernel 2.1	8.744578e-04	0.95
Kernel 2.2	8.440223e-04	0.98
Kernel 2.3	3.099267e-04	2.67

10000*x*5000		
Kernel	**Time (s)**	**Speedup**
MAGMA	1.856964e-03	Base
Kernel 2.1	3.835175e-03	0.48
Kernel 2.2	3.554660e-03	0.52
Kernel 2.3	1.191753e-03	1.56

Table 4.19: Execution Times and Speedups versus MAGMA for Selected Matrix Sizes in Figure 4.17

Both Figures 4.16 and 4.17 show that Kernel 2.3 consistently outperforms Magma, making it the most optimized solution for rectangular matrix transpositions.

Lastly, the performance of very thin matrices is tested in Figure 4.18.

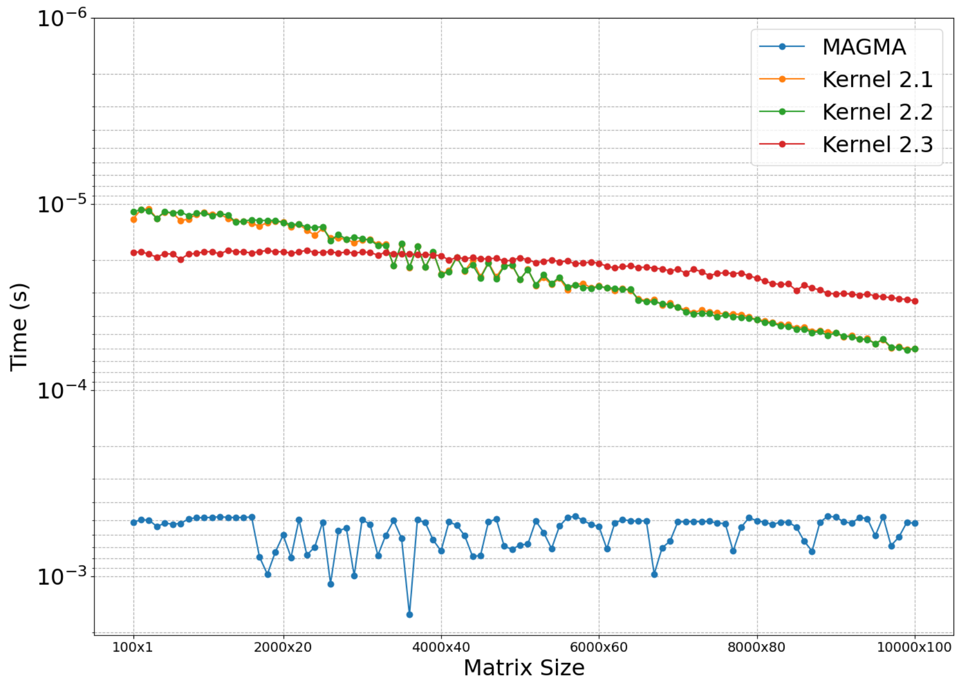


Figure 4.18: Extremely Thin Matrix Transposition (HELIOS)

100x1		
Kernel	**Time (s)**	**Speedup**
MAGMA	5.122348e-04	Base
Kernel 2.1	1.204925e-05	42.51
Kernel 2.2	1.102865e-05	46.45
Kernel 2.3	1.823005e-05	28.10

5000x50		
Kernel	**Time (s)**	**Speedup**
MAGMA	6.794136e-04	Base
Kernel 2.1	2.548490e-05	26.66
Kernel 2.2	2.542265e-05	26.72
Kernel 2.3	1.946810e-05	34.90

10000x100		
Kernel	**Time (s)**	**Speedup**
MAGMA	5.189014e-04	Base
Kernel 2.1	6.009410e-05	8.63
Kernel 2.2	5.970145e-05	8.69
Kernel 2.3	3.311470e-05	15.67

Table 4.20: Execution Times and Speedups versus MAGMA for Selected Matrix Sizes in Figure 4.18

Figure 4.18 reveals that Kernels 2.1 and 2.2 are slightly more efficient than Kernel 2.3 for matrices up to the size of approximately 4000×40, after which Kernel 2.3 becomes faster. All kernels demonstrate superior performance compared to Magma, emphasizing their effectiveness across varying dimensions and configurations.

In conclusion, on the device specified in 4.4, the performance analysis reveals that our CUDA transposition kernels are highly effective, particularly in specialized configurations. Kernels 2.3 to 2.4 generally outperform the Magma, especially notable in Kernel 2.4's superior in-place transposition for large square matrices. Kernel 2.3 excels in managing rectangular matrices, consistently delivering optimal performance. Even in challenging conditions with extremely thin matrices, our kernels maintain a competitive edge.

4.2.6.2 GP1

The graphs below illustrate the performance of our transposition kernels on the GP1 device, allowing for a comparative analysis against the HELIOS device.

Figures below present the results for square and various rectangular matrix configurations.

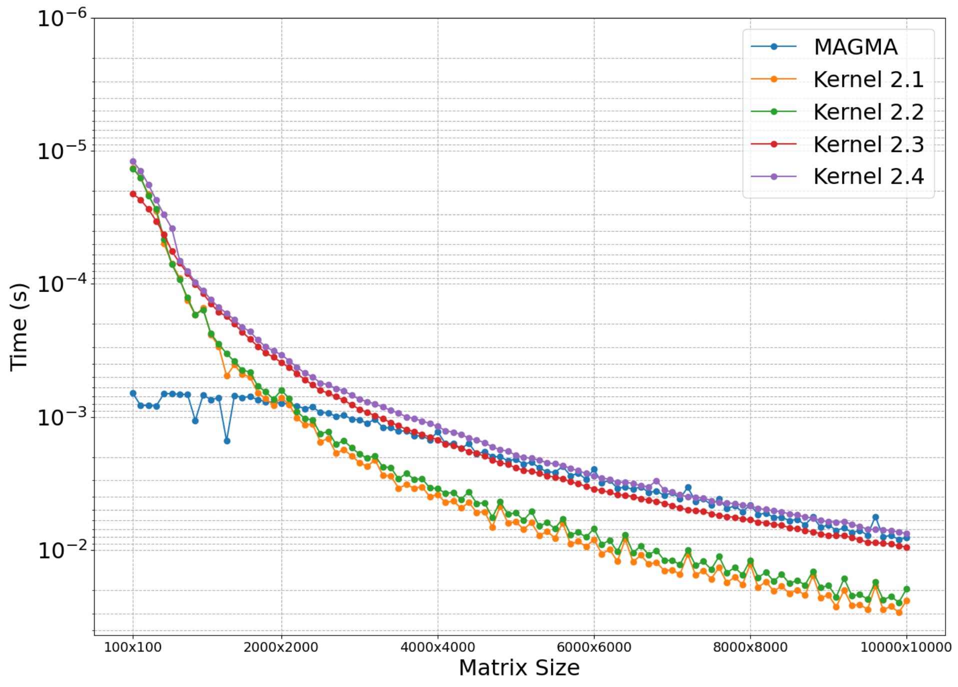


Figure 4.19: Square Matrix Transposition (GP1)

100*x*100		
Kernel	**Time (s)**	**Speedup**
MAGMA	6.626254e-04	Base
Kernel 2.1	1.344615e-05	49.27
Kernel 2.2	1.357220e-05	48.82
Kernel 2.3	2.109340e-05	31.41
Kernel 2.4	1.195765e-05	55.41

5000*x*5000		
Kernel	**Time (s)**	**Speedup**
MAGMA	2.071773e-03	Base
Kernel 2.1	6.139651e-03	0.34
Kernel 2.2	5.266100e-03	0.39
Kernel 2.3	2.415373e-03	0.86
Kernel 2.4	1.937911e-03	1.07

10000*x*10000		
Kernel	**Time (s)**	**Speedup**
MAGMA	8.081182e-03	Base
Kernel 2.1	2.396712e-02	0.34
Kernel 2.2	1.956313e-02	0.41
Kernel 2.3	9.526299e-03	0.85
Kernel 2.4	7.535703e-03	1.07

Table 4.21: Execution Times and Speedups versus MAGMA for Selected Matrix Sizes in Figure 4.19

In Figure 4.19, the transposition of square matrices shows that all kernels perform comparably, maintaining consistent efficiency. Kernel 2.4, which performs an in-place transposition, is the best, outperforming the others.

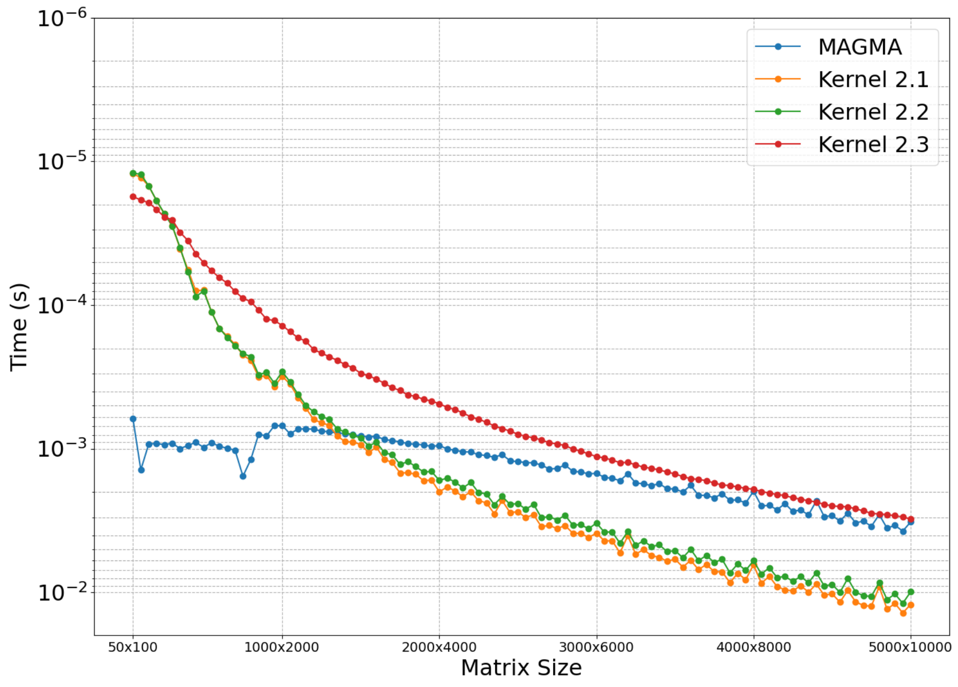


Figure 4.20: Wide Matrix Transposition with Incremental Width Expansion (GP1)

50*x*100		
Kernel	**Time (s)**	**Speedup**
MAGMA	6.196810e-04	Base
Kernel 2.1	1.221605e-05	50.73
Kernel 2.2	1.205160e-05	51.42
Kernel 2.3	1.759010e-05	35.23

2500*x*5000		
Kernel	**Time (s)**	**Speedup**
MAGMA	1.229944e-03	Base
Kernel 2.1	2.748773e-03	0.44
Kernel 2.2	2.405007e-03	0.51
Kernel 2.3	7.931828e-04	1.55

5000*x*10000		
Kernel	**Time (s)**	**Speedup**
MAGMA	3.236129e-03	Base
Kernel 2.1	1.213547e-02	0.27
Kernel 2.2	9.921461e-03	0.33
Kernel 2.3	3.080433e-03	1.05

Table 4.22: Execution Times and Speedups versus MAGMA for Selected Matrix Sizes in Figure 4.20

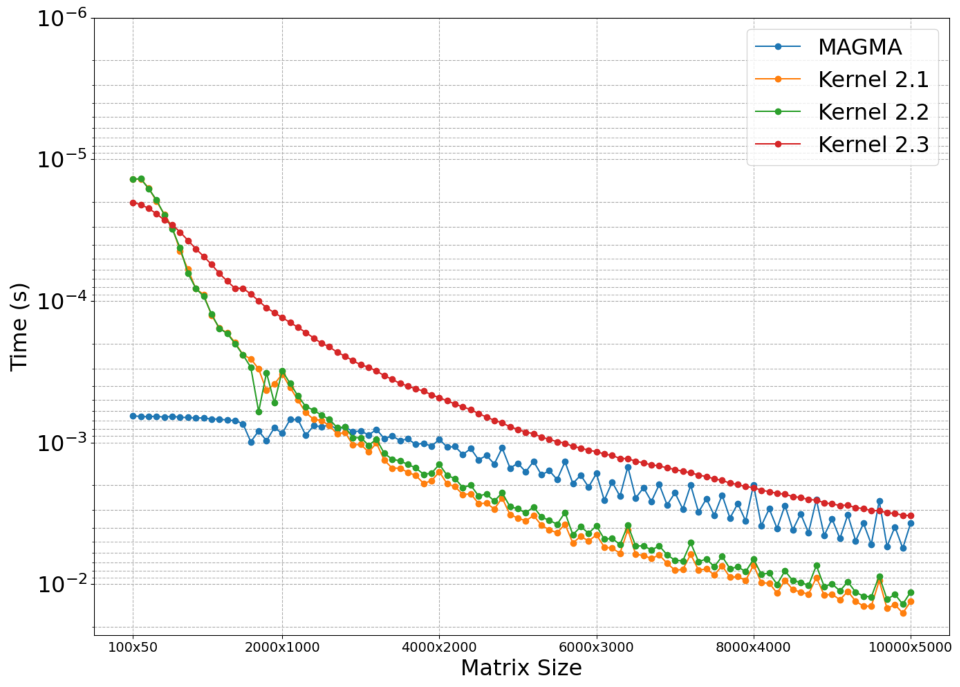


Figure 4.21: Tall Matrix Transposition with Incremental Height Expansion (GP1)

100*x*50		
Kernel	**Time (s)**	**Speedup**
MAGMA	6.451399e-04	Base
Kernel 2.1	1.375110e-05	46.92
Kernel 2.2	1.380395e-05	46.74
Kernel 2.3	2.015850e-05	32.00

5000*x*2500		
Kernel	**Time (s)**	**Speedup**
MAGMA	1.402077e-03	Base
Kernel 2.1	3.408027e-03	0.41
Kernel 2.2	2.933987e-03	0.48
Kernel 2.3	8.064713e-04	1.74

10000*x*5000		
Kernel	**Time (s)**	**Speedup**
MAGMA	3.702901e-03	Base
Kernel 2.1	1.323696e-02	0.28
Kernel 2.2	1.133947e-02	0.33
Kernel 2.3	3.268887e-03	1.13

Table 4.23: Execution Times and Speedups versus MAGMA for Selected Matrix Sizes in Figure 4.21

In Figures 4.20 and 4.21, the kernels maintain similar performance levels, indicating that our implementations handle rectangular matrices effectively on the GP1 device.

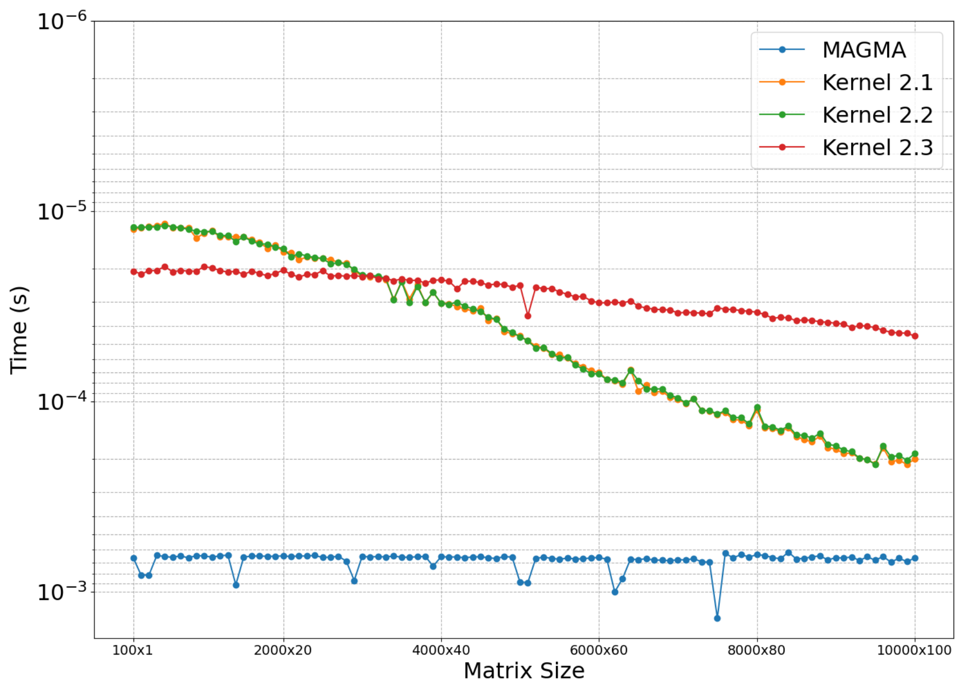


Figure 4.22: Extremely Thin Matrix Transposition (GP1)

100*x*1		
Kernel	**Time (s)**	**Speedup**
MAGMA	6.599223e-04	Base
Kernel 2.1	1.247475e-05	52.90
Kernel 2.2	1.215480e-05	54.29
Kernel 2.3	2.063755e-05	31.98

5000*x*50		
Kernel	**Time (s)**	**Speedup**
MAGMA	8.866718e-04	Base
Kernel 2.1	4.523585e-05	19.60
Kernel 2.2	4.571545e-05	19.40
Kernel 2.3	2.449925e-05	36.19

10000*x*100		
Kernel	**Time (s)**	**Speedup**
MAGMA	6.634437e-04	Base
Kernel 2.1	2.004289e-04	3.31
Kernel 2.2	1.873964e-04	3.54
Kernel 2.3	4.526190e-05	14.66

Table 4.24: Execution Times and Speedups versus MAGMA for Selected Matrix Sizes in Figure 4.22

Figure 4.22 examines the performance for extremely thin matrices. Here, for smaller matrices, Kernels 2.1 and 2.2 outperform Kernel 2.3, but for larger sizes, Kernel 2.3 becomes significantly faster. All kernels demonstrate consistent efficiency, maintaining competitiveness with the benchmarking libraries.

Overall, the GP1 device shows that our transposition kernels exhibit performance characteristics similar to those on the HELIOS device. This suggests that the architectural differences between GP1 and HELIOS do not significantly impact the performance of our transposition kernels. Kernel 2.3, which excelled in rectangular matrix transpositions on HELIOS, and Kernel 2.4, known for its in-place transposition efficiency, maintain competitive performance on GP1.

Conclusion

The goal of this bachelor's degree project was to explore and implement efficient parallel algorithms for dense matrix operations, focusing specifically on multiplication and transposition. The project successfully developed a comprehensive suite of optimized kernels, significantly improving upon initial implementations (Kernel 1.1 and Kernel 2.1) by employing advanced techniques and methods.

These kernels were tested on both Tesla (HELIOS) and Pascal (GP1) architectures, confirming their accuracy and efficiency across different hardware platforms. Performance comparisons with well-established GPU libraries from NVIDIA demonstrated that our kernels exhibit competitive performance. In particular, the multiplication kernels showed superior performance in smaller matrix sizes across both platforms. Additionally, these kernels exhibited significantly better performance on the Pascal architecture compared to Helios, underscoring the impact of architectural differences. In contrast, the transposition kernels demonstrated consistent performance on both architectures, nearly always outperforming Magma.

The multiplication Kernel 1.5 has been developed into a function capable of handling both standard matrix multiplication and multiplication with transposed matrices. Additionally, functions for both in-place and out-of-place matrix transposition have been developed from Kernel 2.3 and 2.4. These functionalities, along with the entire benchmarking environment and all developed kernels, will be added to the TNL library following this work's publication. The update will also include documentation with practical usage examples.

In conclusion, the algorithms developed in this thesis not only enhance the functionality of the TNL library but also contribute valuable tools to the field of computational science.

Appendix A

Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
HIP	Heterogeneous-compute Interface for Portability
TNL	Template Numerical Library
WMMA	Warp-level Matrix Multiply Accumulate

Appendix B

Profiling Tables

Metric	Kernel 1.1 (Avg)	Kernel 1.2 (Avg)	Kernel 1.3 (Avg)	Kernel 1.4 (Avg)	Kernel 1.5 (Avg)	Kernel 1.6 (Avg)
Instr. per Warp	4626.0	4311.1	4235.0	2591.5	2592.5	24903
Global Load Thr. (GB/s)	127.80	132.90	132.17	210.23	227.18	47.254
Global Store Thr. (GB/s)	4.4548	4.6326	4.6071	7.3284	7.9193	4.3926
SM Load Trans. per Req.	1.940085	2.000000	2.000000	2.000000	1.500000	4.500000
SM Store Trans. per Req.	1.496247	1.984820	1.984820	2.000000	2.000000	5.000000
Warp Exec. Eff. (%)	100.00	100.00	100.00	100.00	100.00	100.00
Issued IPC	0.407371	0.393491	0.383543	0.375813	0.406589	0.158391
Achieved Occupancy	0.846299	0.846157	0.846527	0.840715	0.839794	0.253719
SM Utilization	Low (2)	Low (2)	Low (2)	Low (3)	Low (2)	Low (1)
DRAM Read Thr. (GB/s)	8.9935	10.071	8.9339	20.751	22.791	2.7453
DRAM Write Thr. (GB/s)	1.1111	1.1635	1.1513	1.8462	1.9788	1.5252

Table B.1: Comparative Performance Metrics of Dense Matrix Multiplication Kernels

Metric	Kernel 2.1 (Avg)	Kernel 2.2 (Avg)	Kernel 2.3 (Avg)	Kernel 2.4 (Avg)
Instr. per Warp	161.259375	168.161458	39.299805	45.000000
Global Load Thr. (GB/s)	226.185	237.51	329.33	314.06
Global Store Thr. (GB/s)	226.185	237.51	164.66	157.03
SM Load Trans. per Req.	1.996032	2.000000	2.000000	2.000000
SM Store Trans. per Req.	1.988281	1.998677	1.123047	2.000000
Warp Exec. Eff. (%)	99.71	99.89	99.97	100.0
Issued IPC	0.255455	0.285753	0.799932	0.398982
Achieved Occupancy	0.716286	0.645097	0.761484	0.632813
SM Utilization	Low (1)	Low (1)	Low (1)	Low (1)
DRAM Read Thr. (GB/s)	88.771	87.755	79.054	49.225
DRAM Write Thr. (GB/s)	86.960	84.177	77.767	72.069

Table B.2: Comparative Performance Metrics of Dense Matrix Transposition Kernels

Bibliography

[1] Lecture on Dense Linear Algebra. *MIT Course 18.337: Introduction to Computational Science and Engineering*. Available at: http://courses.csail.mit.edu/18.337/2004/book/Lecture_04-Dense_Linear_Algebra.pdf. Accessed: 2024-02-20.

[2] Wikipedia. Row- and column-major order. Available at: https://en.wikipedia.org/wiki/Row-_and_column-major_order. Accessed: 2024-04-04.

[3] CPU vs. GPU: What's the Difference? *Intel*. Available at: https://www.intel.com/content/www/us/en/products/docs/processors/cpu-vs-gpu.html. Accessed: 2024-02-23.

[4] CUDA C++ Programming Guide. *NVIDIA Corporation*. Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 2024-02-20.

[5] NVIDIA CUDA C Best Practices Guide. *NVIDIA Corporation*. Available at: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/. Accessed: 2024-04-17.

[6] HIP Programming Guide, Version 5.3. *AMD Corporation*. Available at: https://docs.amd.com/bundle/HIP-Programming-Guide-v5.3/page/Introduction_to_HIP_Programming_Guide.html. Accessed: 2024-02-20.

[7] NVIDIA Developer Blog. Programming Tensor Cores in CUDA 9. Available at: https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/. Accessed: 2024-02-20.

[8] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, Jeffrey S. Vetter. *NVIDIA Tensor Core Programmability, Performance & Precision*. KTH Royal Institute of Technology and Oak Ridge National Laboratory.

[9] GPU Computing – HPDA-Python Documentation. *European Centre for Connected Systems (ENCCS)*. Available at: https://enccs.github.io/hpda-python/GPU-computing/. Accessed: 2024-02-20.

[10] Juan Gómez-Luna, I-Jui Sung, Li-Wen Chang, José María González-Linares, Nicolás Guil, and Wen-Mei W. Hwu. In-Place Matrix Transposition on GPUs. *ResearchGate*. Available at: https://www.researchgate.net/publication/273912700_In-Place_Matrix_Transposition_on_GPUs. Accessed: 2024-04-17.

[11] Oberhuber, T.; Klinkovský, J.; et al. *Template Numerical Library*. Available at: https://tnl-project.org. Accessed: 2024-02-23.

[12] Vector Norms in Machine Learning. *Machine Learning Mastery*. Available at: https://machinelearningmastery.com/vector-norms-machine-learning/. Accessed: 2024-02-20.

[13] CUDA Matrix-Matrix Multiplication (MMM) – An Overview. *SiBöhm.com*. Available at: `https://siboehm.com/articles/22/CUDA-MMM`. Accessed: 2024-02-20.

[14] Rajib Nath, Stanimire Tomov, and Jack Dongarra. *An Improved MAGMA GEMM For Fermi Graphics Processing Units*. Available at: `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=820d459a77e6e38570b3e53f661ebf2d3d2e7f6b`. Accessed: 2024-01-04.

[15] Basic Linear Algebra Subprograms (BLAS). *Netlib*. Available at: `https://www.netlib.org/blas/`. Accessed: 2024-02-20.

[16] Innovative Computing Laboratory, University of Tennessee, Knoxville. MAGMA: Matrix Algebra on GPU and Multicore Architectures. Available at: `https://icl.utk.edu/magma/`. Accessed: 2024-02-20.

[17] NVIDIA Developer Blog. CUTLASS: Fast Linear Algebra in CUDA C++. Available at: `https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/`. Accessed: 2024-02-20.

[18] AMD HIP Programming Guide. *AMD.com*. Available at: `https://www.amd.com/en/developer/browse-by-resource-type/documentation.html`. Accessed: 2024-06-15.

[19] Roofline Model. Available at: `https://www.sciencedirect.com/topics/computer-science/roofline-model`. Accessed: 2024-06-11.