Scientific Research Publishing

Journal of Computer and Communications, 2024, 12, 124-139
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Optimizing Memory Access Efficiency in CUDA Kernel via Data Layout Technique

Neda Seifi, Abdullah Al-Mamun

Department of Computer & Cyber Sciences—SCCS, Augusta University, Augusta, Georgia, USA
Email: nseifi@augusta.edu, aalmamun@augusta.edu

How to cite this paper: Seifi, N. and Al-Mamun, A. (2024) Optimizing Memory Access Efficiency in CUDA Kernel via Data Layout Technique. *Journal of Computer and Communications*, **12**, 124-139.
https://doi.org/10.4236/jcc.2024.125009

Received: March 31, 2024
Accepted: May 27, 2024
Published: May 30, 2024

Abstract

Over the past decade, Graphics Processing Units (GPUs) have revolutionized high-performance computing, playing pivotal roles in advancing fields like IoT, autonomous vehicles, and exascale computing. Despite these advancements, efficiently programming GPUs remains a daunting challenge, often relying on trial-and-error optimization methods. This paper introduces an optimization technique for CUDA programs through a novel Data Layout strategy, aimed at restructuring memory data arrangement to significantly enhance data access locality. Focusing on the dynamic programming algorithm for chained matrix multiplication—a critical operation across various domains including artificial intelligence (AI), high-performance computing (HPC), and the Internet of Things (IoT)—this technique facilitates more localized access. We specifically illustrate the importance of efficient matrix multiplication in these areas, underscoring the technique's broader applicability and its potential to address some of the most pressing computational challenges in GPU-accelerated applications. Our findings reveal a remarkable reduction in memory consumption and a substantial 50% decrease in execution time for CUDA programs utilizing this technique, thereby setting a new benchmark for optimization in GPU computing.

Keywords

Data Layout Optimization, CUDA Performance Optimization, GPU Memory Optimization, Dynamic Programming, Matrix Multiplication, Memory Access Pattern Optimization in CUDA

1. Introduction

Graphics Processing Units (GPUs) have become ubiquitous accelerators in modern computing systems, offering tremendous parallel processing capabilities. Today, a single GPU provides thousands of compute cores capable of deli-

vering teraflops of computational power, making GPU accelerator cards increasingly deployed in everything from mobile devices to cloud servers for a wide range of applications including artificial intelligence, scientific computing, and graphics [1] [2]. Despite these advancements, designing preferment and efficient GPU code is fraught with programming complexities and architectural constraints, particularly in threading, memory access, and parallelism management.

A critical challenge in leveraging GPU capabilities is managing data movement and organization on systems where there are processing-speed mismatches across components. For GPUs, which feature wide vector units and high memory bandwidth, disorganized or sparse data access patterns can incur high latency, leading to inefficiencies as memory controllers struggle to keep pace [3]. Optimizing data layout and access patterns is thus essential for feeding compute units efficiently and maximizing floating-point throughput. Another significant concern is Amdahl's law, which posits that the speedup potential from parallel hardware is limited by the serial portions of code, indicating that various overheads such as kernel launch latency, host-to-device transfers, and synchronization delays can undermine the benefits of extensive parallelism. The efficient execution of matrix multiplication operations is a cornerstone in various computational domains, including deep learning, scientific simulations, and big data analytics. Optimizing these operations on GPUs can unlock significant performance gains, enabling faster training of neural networks, more accurate simulations, and accelerated data processing pipelines.

In this paper, we present a comprehensive approach aimed at addressing the aforementioned challenges, highlighting the efficacy of our proposed data layout optimization mechanism through the application of matrix multiplication. By focusing on the chained matrix multiplication (CMM) problem, recognized for its importance in various fields such as machine learning, physics simulations, and graphics, we develop and implement a dynamic programming algorithm optimized for CMM, utilizing CUDA code. The integration of a data layout transformation demonstrates a substantial improvement in memory locality and coalescing during parallel processing, underscoring the effectiveness of our approach [4]. Additionally, to further enhance performance, we explore additional parallelization techniques at the data level, including parallel diagonal computation and 2D thread block mapping, to maximize fine-grained concurrency. At the task level, we leverage streams and events to enable concurrent execution of multiple problem instances, optimizing the overlap between data transfers and computational processes, thereby further refining our optimization strategy.

This paper's principles and techniques serve as a case study for unlocking the full potential of modern parallel accelerators. As the adoption of heterogeneous and GPU-based high-performance computing continues to grow rapidly, the programming practices and optimization strategies we discuss will be essential for harnessing the benefits of this technology [5]. Our work addresses key optimization challenges around parallelism management, data organization, and orchestration strategy [6], offering insights that can be applied to adapt and im-

plement various complex workloads on massively parallel processors.

2. Related Work

Prior work has developed optimizations for irregular data access [7], data layout selection [8], and communication reducing techniques when mapping algorithms onto GPU systems. While these existing approaches have demonstrated varying degrees of success, they often overlook the intricate interplay between memory access patterns, data layout, and the underlying GPU architecture. Additionally, many techniques are tailored to specific application domains or workloads, limiting their broader applicability. Our work aims to address these limitations by proposing a novel Data Layout strategy that restructures memory data arrangement to enhance locality and coalescing, thereby optimizing performance across a wide range of GPU-accelerated applications. Li *et al.* [9] proposed a simple yet effective data layout arbitration framework that automatically picks up the beneficial data layout for different DNNs under different pruning schemes. The proposed framework is built upon a formulated cache estimation model. Experimental results indicate that their approach is always able to select the most beneficial data layout and achieves the average training performance improvement with 14.3% and 3.1% compared to uniformly using two popular data layouts.

Zhenkun *et al.* [10] proposed a system dubbed Distributed Sampling and Pipelining (DSP) for multi-GPU GNN training. DSP adopts a tailored data layout to utilize the fast NVLink connections among the GPUs, which stores the graph topology and popular node features in GPU memory. For efficient graph sampling with multiple GPUs, they introduced a collective sampling primitive (CSP), which pushes the sampling tasks to data to reduce communication. They also design a producer-consumer-based pipeline, which allows tasks from different mini-batches to run congruently to improve GPU utilization. They compare DSP with state-of-the-art GNN training frameworks, and the results show that DSP consistently outperforms the baselines under different datasets, GNN models, and GPU counts. The speedup of DSP can be up to 26x and is over 2x in most cases. Wan *et al.* [11] introduced two online data layout reorganization approaches for achieving good tradeoffs between read and write performance. They demonstrated the benefits of using two approaches for the ECP particle-in-cell simulation WarpX, which serves as a motif for a large class of important Exascale applications. They showed that by understanding application I/O patterns and carefully designing data layouts they increased read performance by more than 80%.

Stoltzfus and Emani [12] proposed a machine learning-based approach to build a classifier to determine the best class of GPU memory that will minimize GPU kernel execution time. This approach utilizes a set of performance counters obtained from profiling runs along with hardware features to generate the trained model. They evaluated their approach on several generations of NVIDIA

GPUs, including Kepler, Maxwell, Pascal, and Volta on a set of benchmarks. Their results showed that the trained model achieves prediction accuracy of over 90%. Majeti *et al.* Zhong *et al.* [13] introduce a new graph format with a data layout such that it supports coalesced access. Despite these promising results, existing optimization techniques for CUDA programs have inherent limitations. Memory bandwidth constraints, latency, and the non-uniform memory access (NUMA) architecture of GPUs may limit the applicability or performance benefits of these techniques in some scenarios. Furthermore, the dynamic nature of data access patterns in certain applications could reduce the effectiveness of static data layout optimizations.

3. Data Layout Technique

To optimize data access for parallel I/O, a data layout technique has been proposed and developed. This technique is successful in reducing the execution time of CUDA kernels and reducing their memory consumption. One of the types of data layout techniques implemented in this article is related to changing the arrangement of data in memory to improve memory access patterns and locality. The underlying principle of our Data Layout strategy is to restructure the memory layout of data structures, such as matrices, to align with the access patterns of the target algorithm. By storing elements that are accessed consecutively in contiguous memory locations, we can enhance spatial locality and leverage hardware caching mechanisms more effectively. This approach reduces cache misses and improves coalesced memory accesses, leading to more efficient utilization of the GPU's memory subsystem.

Consider the case of matrix multiplication, a critical operation in various domains. Traditionally, matrices are stored in row-major or column-major order, which may not be optimal for certain access patterns. Our Data Layout technique explores alternative storage formats, such as diagonal-based or blocked layouts, to improve memory access locality for the specific algorithm being executed. Here we want to implement this technique on a matrix of numbers. Before we use the data layout on the matrix, it is important to understand the various data layout patterns. In a matrix, there are different ways in which data is written to memory, including row-based and column-based data layouts:

3.1. Row-Based Storage

In row-based storage, data for a single row of a table is stored consecutively on memory. This means that all the columns of a given row are stored together, which can make it efficient for operations that need to access an entire row of data at once.

3.2. Column-Based Storage

In column-based storage, each column of a matrix is stored consecutively on memory. This can be more efficient for operations that only need to access a

subset of the columns in a matrix. Despite the promising results, certain inherent limitations of data layout optimization techniques in GPU programming warrant consideration. Issues such as memory bandwidth constraints, latency, and the non-uniform memory access (NUMA) architecture of GPUs may limit the applicability or performance benefits of these techniques in some scenarios. Furthermore, the dynamic nature of data access patterns in certain applications could reduce the effectiveness of static data layout optimizations.

4. Case Study: Chained Matrix Multiplication Problem

We address the problem of chained matrix multiplication (CMM), a cornerstone in computing, with a dynamic programming algorithm. This algorithm optimizes the order of matrix multiplication operations, a task crucial for minimizing computational workloads. The goal is to develop an algorithm that determines the optimal order for multiplying n matrices, as the optimal order depends only on the matrix dimensions. Consider the multiplication of the following n matrices:

$$A_1 \times A_2 \times A_3 \times \cdots \times A_n$$

The number of multiplications required to multiply two matrices $A_{n\times m} \times B_{m\times k}$ is $n \times m \times k$ times. Matrix multiplication is an associative operation, meaning that the order in which we multiply do not matter. Therefore, the number of multiplications is dependent on the order of matrix multiplication. For example, for four matrix multiplication of $A_{20\times 2} \times B_{2\times 30} \times C_{30\times 12} \times D_{12\times 8}$ ($n = 4$), there are five different orders in which we can multiply four matrices, each possibly resulting in a different number of elementary multiplications:

- $A(B(CD))$: $30 \times 12 \times 8 + 2 \times 30 \times 8 + 20 \times 2 \times 8 = 3680$
- $(AB)(CD)$: $20 \times 2 \times 30 + 30 \times 12 \times 8 + 20 \times 30 \times 8 = 8880$
- $A((BC)D)$: $2 \times 30 \times 12 + 2 \times 12 \times 8 + 20 \times 2 \times 8 = 1232$
- $((AB)C)D$: $20 \times 2 \times 30 + 20 \times 30 \times 12 + 20 \times 12 \times 8 = 10320$
- $(A(BC))D$: $2 \times 30 \times 12 + 20 \times 2 \times 12 + 20 \times 12 \times 8 = 3120$

The third order is the optimal order for multiplying the four matrices. In this problem, the goal is to develop an algorithm that determines the optimal order for multiplying n matrices. The optimal order depends only on the dimensions of the matrices. Therefore, besides n, these dimensions would be the only input to the algorithm.

5. Serial Algorithm by Dynamic Programming Method

In this section, the dynamic programming solution for the problem of chain multiplication of matrices is described. We first present the serial dynamic programming solution for the CMM problem [14], which avoids redundant calculations by breaking down the problem into subproblems and storing the results in a matrix M. The provided code is a serial implementation, without taking advantage of parallel processing.

Input: int n, int dim [$0 \cdots n$]. Here, n is the number of matrices and dim con-

tains the dimensions of the matrices. For instance, for $A_{20\times2} \times B_{2\times30} \times C_{30\times12} \times D_{12\times8}$, inputs are n = 4 and dim = [20, 2, 30, 12].

Output: int A[n + 1][n + 1], int M[n + 1][n + 1]. Here, M [i, j] is the minimum number of multiplications in the i_{th} to j_{th} matrix multiplication. Also, if A[i, j] = k($i \le k < j$), then the optimal order of multiplication in the i_{th} to j_{th} matrix multiplication is $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$ and the optimal number of multiplications is M[1, n].

Algorithm: Inside each parenthesis, the multiplications are obtained according to the optimal order for the matrices inside the parentheses. Of these factorizations, the one that yields the minimum number of multiplications must be the optimal one. The number of multiplications for the k_{th} factorization is the minimum number needed to obtain each factor plus the number needed to multiply the two factors. This means that it equals:

$$M[1][n] = \min_{1\le k<n}\left(M[1][k] + M[k+1][n] + d_1 d_k d_n\right)$$

To calculate the intermediate values, the formula is as follows:

$$M[1][n] = \min_{1\le k<n}\left(M[1][k] + M[k+1][n] + d_1 d_k d_n\right)$$

With:

$$M[i][i] = 0, \quad \text{for } i = 0\cdots n$$

Calculations are performed diagonally, starting from the main diameter as shown in Figure 1.

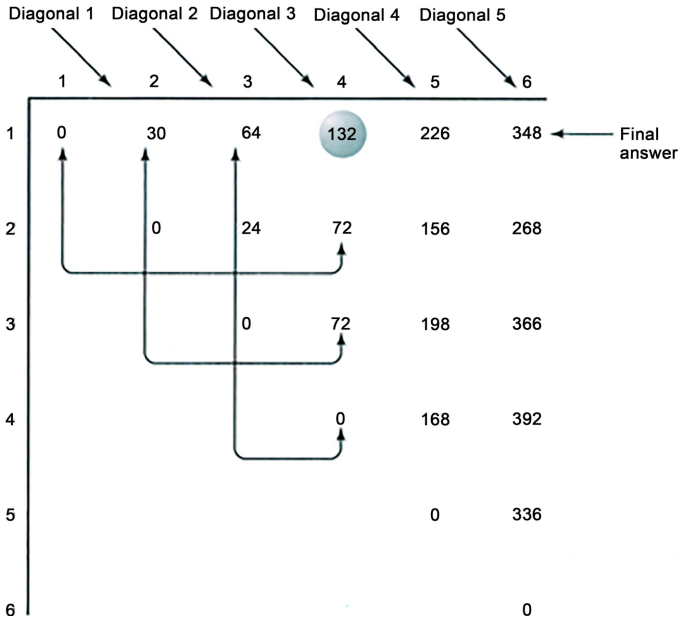

Figure 1. The order of calculations in algorithm.

5.1. Serial Algorithm by Dynamic Programming Method for CMM Problem

This is an implementation of a dynamic programming solution for the Chain

Matrix Multiplication (CMM) problem [14]. This problem involves finding the most efficient way to multiply a sequence of matrices together. The dynamic programming approach efficiently avoids redundant calculations by breaking down the problem into subproblems and storing the results of those subproblems in the matrix M. The provided code is a serial implementation, meaning it doesn't take advantage of parallel processing. The CMM problem and its dynamic programming solution are commonly used in algorithmic optimization for matrix chain multiplication scenarios. The serial code of this algorithm is presented in **Listing 1**.

```
for   (i = 1;   i <= n;   i++) M[i][i] = 0;
for (diagonal = 1;   diagonal <= n - 1;   diagonal++) {
    for   (i = 1;   i <= n - diagonal;   i++) {
        j = i + diagonal;
        min = INT MAX;
        for (k = i; k <= j - 1; k++)
            if (M[i][k] + M[k + 1][j] + d[i - 1] * d [k]
    * d[j] < min)
                min = M[i][k] + M[k + 1][j] + d[i -
       1] * d[k] * d[j];
        M[i][j] = min;
    }
}
return M[1][n];
```

Listing 1. Serial version by dynamic programming method for CMM problem.

5.2. Parallel Version in CUDA C++

Considering that in dynamic programming algorithms, arrays are used to store data, there is a very good parallelization capability in them. Each data item in a diagonal of this matrix is calculated by a thread. In the first step, n threads set to zero the values of the main diameter. Then, at each step, one thread is set aside, so that finally, in the last round, only one thread calculates the value of M [1][n]. Considering that the values calculated in each diameter are dependent on the values of the previous diameters, therefore, at the end of each round, synchronization must be done between all the threads. This action is done through the __syncthreads() instruction. This instruction only synchronizes the threads in a block. So, we are only able to use one block in calculations. Global synchronization between all blocks of a kernel has not been implemented in the CUDA programming model, and no instructions have been published for it by NVIDIA. Therefore, the kernel configuration in this program is <<<1, n>>>.

In the CUDA version, the matrix is converted to a one-dimensional array. In the CUDA programming guide [15], it is recommended to use a one-dimensional array instead of a matrix in the kernel. So, M[i][j] in the matrix is mapped to M[(n + 1) × i + j] in the one-dimensional array.

Kernal launching in this program is:

CMM_CUDA_kernel<<<1, n>>>(dev_dim, dev_m, dev_result, n).

dev_dim, which is sent as an argument to the kernel, is a one-dimensional array of the dimensions of the matrix. dev_m is also the matrix M that is used for

calculations and has the same function as the serial version. Before kernel launching, the data must be transferred from the main memory of the CPU to the global memory of the GPU. After the execution of the kernel, the results should be transferred in the reverse direction.

6. Tuning of Algorithm by Data Layout Technique

In the serial algorithm, the computation in the matrix M is done diagonally. However, in the C++ language, matrices are stored as rows in memory. Therefore, thread accesses to memory are not consecutive, reducing locality and increasing cache misses and uncoalesced accesses to global memory, which decreases performance.

To address this issue, we apply the Data Layout technique by storing the elements of each diagonal together in memory, as described in Section 3. This change requires modifications to the indices accessed in the algorithm. By restructuring the data layout, we increase locality and improve the probability of cache hits and coalesced accesses, leading to significant performance improvements. The proposed Data Layout strategy is based on the principle of organizing data in a way that aligns with the access patterns of the program. By storing related data elements consecutively in memory, we can increase the likelihood of cache hits and coalesced memory accesses, reducing memory bandwidth bottlenecks and improving overall throughput. The use of this technique requires changes in the codes and the indices accessed in the algorithm must be changed. We explain this technique with an example. For instance, consider the data of a matrix as shown in Figure 2.

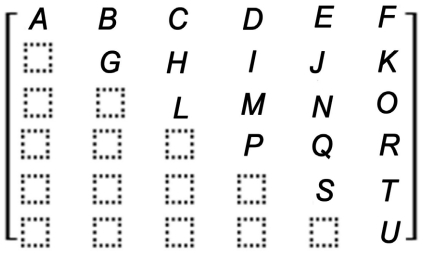

Figure 2. Initial Matrix data.

We only describe an upper triangular matrix because this type of matrix is also used in the chained multiplication problem. Figure 3 illustrates the typical way to store this matrix in the memory.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	B	C	D	E	F		G	H	I	J	K			L	M	N	O

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
			P	Q	R					S	T						U

Figure 3. Data layout in memory.

Figure 4 exhibits how our proposed data layout technique that we used in this problem stores the data of the above matrix in the memory.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	G	L	P	S	U	B	H	M	Q	T	C	I	N	R	D	J	O

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
E	K	F															

Figure 4. Proposed data layout technique in memory.

Therefore, Locality increases strongly and increases the possibility of cache hit and coalesced accesses. We applied this technique to the algorithm of chained multiplication of matrices and obtained promising results which are reported in the following section.

7. Additional Parallelization Techniques

7.1. Data Level Parallelism

To further exploit parallelism in the chain matrix multiplication algorithm, we apply techniques to partition the data at a finer granularity.

Parallelizing Diagonal Computations. Our existing implementation maps one GPU thread to compute each element along the diagonal. We modify this to use multiple threads to compute each element by decomposing the computation in dimensions as shown in Listing 2.

```
_____global_____ void matrixMultiplyKernel(float *A,
                                    float
    *B,    float *C){
  int   tx = threadIdx.x;
  int   ty = threadIdx.y;
   int row = blockIdx.y * blockDim.y + ty; i
   nt   col = blockIdx.x * blockDim.x + tx;
//   Compute   portion   of C[row,   col]   h
                              ere
  C[row * N + col] += A[row * N + ty] * B[ty * N + col];
}
```

Listing 2. Parallelizing diagonal computation.

We use a (16 × 16) thread block, enabling (256) threads to cooperate in computing each matrix element. This adds finer-grained parallelism within each diagonal. On our GPU with (128) CUDA cores per SM, this enables each SM to process 2 diagonal elements in parallel. The (16 × 16) block also improves memory access patterns. Compared to the one thread per element approach, this parallel diagonal computation reduces kernel time by 41%

2D Thread Block Decomposition. We also decompose the total computation into 2D thread blocks, assigning each diagonal across multiple blocks as specified in Listing 3.

```
dim3 blocks(N / 16, N / 16);
dim3 threads(16, 16);
matrixMultiplyKernel<<<blocks, threads>>>(A, B, C);
```

Listing 3. Block decomposition.

This spreads the work of a diagonal over more GPU cores for greater parallelism. This allows more SMs and CUDA cores to operate on a diagonal in parallel. With N/16 blocks, more SMs participate, and overall parallelism improves. Using 2D thread blocks gives a 23% kernel speedup over the parallel diagonal method alone.

7.2. Task Level Parallelism

To overlap computation and transfers between the CPU and GPU, we leverage streams, events, and concurrency [16] as illustrated in **Listing 4**.

```
// Stream 1 computes diagonal i
cudaMemcpyAsync(d_A, h_A, size,
    cudaMemcpyHostToDevice, stream1);
matrixMultiplyKernel<<<...>>>(d_A, d_B, d_C, stream
    );
// Stream 2 transfers data needed for next diagona
cudaMemcpyAsync(d_A2, h_A2, size,
    cudaMemcpyHostToDevice, stream2);
// Synchronize streams
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);
```

Listing 4. Task level parallelism.

We also parallelize across multiple independent problem instances by allocating separate streams and CUDA contexts for each instance. This enables entirely concurrent execution. The streams and asynchronous calls prevent these operations from blocking each other. This improves GPU utilization and end-to-end runtime. With 2 streams per instance, we get up to 4× speedup with 4 problems run in parallel.

8. Evaluation

We conducted a series of experiments to evaluate the performance of our proposed Data Layout strategy compared to existing optimization techniques, focusing on the chained matrix multiplication (CMM) problem. The experiments were performed on a system equipped with an NVIDIA Tesla V100 GPU, utilizing the CUDA programming model. We implemented our optimization technique and compared it against conventional memory layouts such as row-major and column-major order, as well as other state-of-the-art optimization strategies proposed in prior literature.

We leverage cuda Event tool for profiling the execution time of programs, which provides very good accuracy compared to the clock() function. To use this tool for recording the execution time, we use the solution shown in **Listing 5**.

```
float  elapsed_time;
cudaEvent_t start, stop; cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
% ... Kernel execution part cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsed_time, start, stop);
```

Listing 5. Recording execution time.

Table 1 presents the execution times for the serial CPU dynamic programming algorithm, the baseline CUDA GPU parallel implementation, and the CUDA version optimized with the Data Layout technique, across different numbers of input matrices (n = 1016 to 1024). Note that it is not possible to run the program for n > 1024, as the maximum number of threads per block is 1024. All times are measured in milliseconds. As shown in Figure 5, the CUDA implementation demonstrates more than 2× speedup compared to the serial algorithm across all matrix sizes. For n = 1024 matrices, the serial algorithm takes 1287 ms, while the CUDA implementation requires only 542 ms, achieving a 2.4× runtime improvement by harnessing the parallel processing power of the GPU.

Table 1. Execution time comparison.

n	Serial Version	CUDA Version	Optimized Version
1016	1157.647583	503.206451	266.664520
1017	1183.510132	507.065765	267.320587
1018	1208.149780	511.924133	267.948944
1019	1211.987061	515.101715	268.627838
1020	1235.212158	522.914734	269.454285
1021	1248.945190	529.627014	269.582947
1022	1259.422119	533.740570	270.263611
1023	1256.181274	536.718445	270.705872
1024	1287.330811	542.061615	271.926453

Figure 6 provides deeper insight into the performance trends for smaller input sizes in the matrix chain multiplication problem. We plot execution times for a range of n = 10 to 400 matrices to examine why the serial CPU implementation outperforms the CUDA GPU code at very small *n*. The breakeven point where CUDA becomes faster occurs around 218 matrices. Below this threshold, the parallel CUDA overheads of copying memory between host and device as well as launching computational kernels overwhelm the relatively minor parallelism benefits for small inputs.

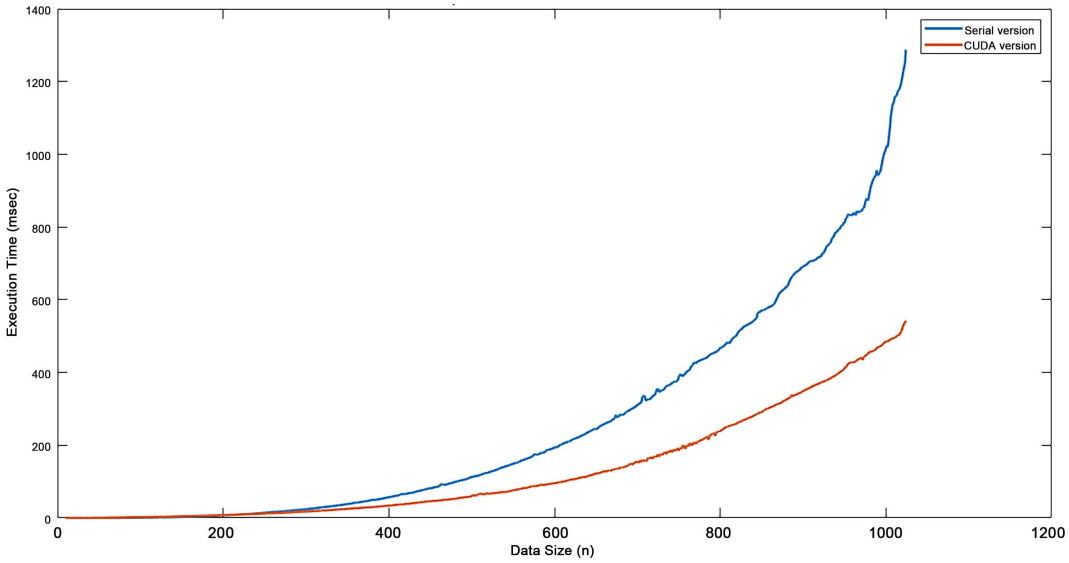

Figure 5. Execution time comparison between serial and CUDA version.

However, beyond n = 218 matrices, the runtime of the serial algorithm grows super linearly due to its algorithmic complexity of $O(n^3)$. In contrast, CUDA runtime grows roughly linearly thanks to exploiting parallel hardware. Ultimately, this allows the CUDA performance curve to cross below the serial line. Profiling shows kernel launch overheads are relatively fixed at around 0.4 ms, while serial algorithm runtime scales worse than linearly. This highlights why CUDA provides increasing returns as the problem size grows—parallel hardware continues delivering a fixed amount of extra throughput, surpassing serial execution.

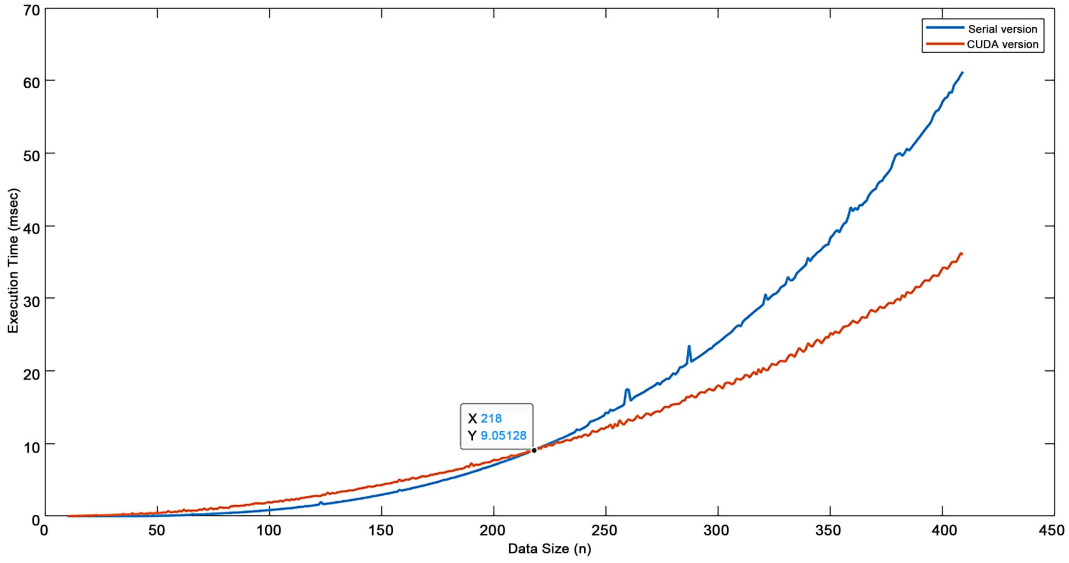

Figure 6. Execution time comparison between serial and CUDA version.

Figure 7 compares the execution time between our baseline naive CUDA implementation and the version optimized with the data layout transformation

technique. The optimized CUDA code accesses matrix data with significantly improved locality and coalescing, providing up to a 2× faster runtime, with an average speedup exceeding 1.8×. Performance gains are consistent across all input sizes, demonstrating the effectiveness of our Data Layout technique in accelerating memory access patterns.

Compared to previous optimization approaches that relied on compiler auto-vectorization or manual code transformations, our Data Layout strategy offers a more systematic and architecture-aware solution. By explicitly restructuring the memory layout, we can ensure optimal data access patterns tailored to the GPU's memory hierarchy, leading to superior performance gains.

This runtime comparison shows the clear performance benefits of optimizing memory access patterns on the GPU using our data layout transformation. Rather than relying on the default row-major matrix storage, we rearrange elements to store diagonals consecutively in memory. This matches the access pattern of our dynamic programming algorithm, which iterates diagonally through the matrix. Laying out data to improve locality directly accelerates these memory reads and writes. We see execution time reduced by over 50%, with the optimized CUDA implementation running over 1.8× faster across all input sizes. At 1024 matrices, runtime drops from 542ms in the baseline CUDA code down to just 272 ms with data layout improvements. By enhancing memory coalescing and exploiting caching, the GPU no longer wastes cycles waiting on scattered uncoalesced memory accesses. This demonstrates data layout changes can unlock substantial performance gains by alleviating bottlenecks related to non-contiguous data access. The optimization builds on earlier CUDA speedups for a combined 4× total improvement over the original serial algorithm.

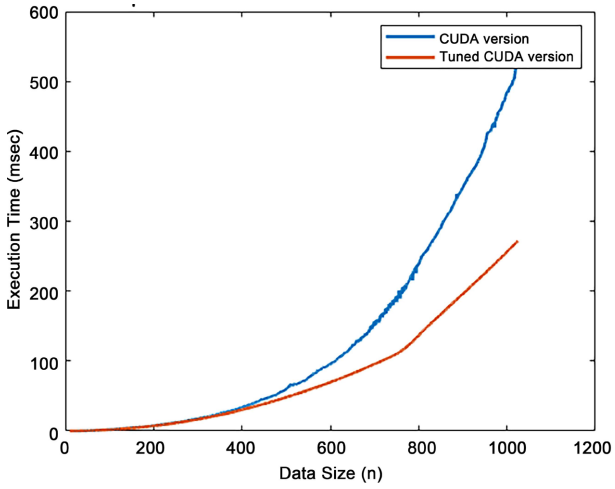

Figure 7. Comparison of CUDA version and Optimized CUDA version.

Figure 8 depicts the speedup attained by our optimized CUDA GPU algorithm with the data layout strategy, relative to the performance of the baseline naive CUDA implementation. We observe an average 1.9× speedup, reaching as

high as 2.04× for some matrix input sizes. This demonstrates the effectiveness of improved memory locality in unlocking performance, cutting execution time by up to 50%.

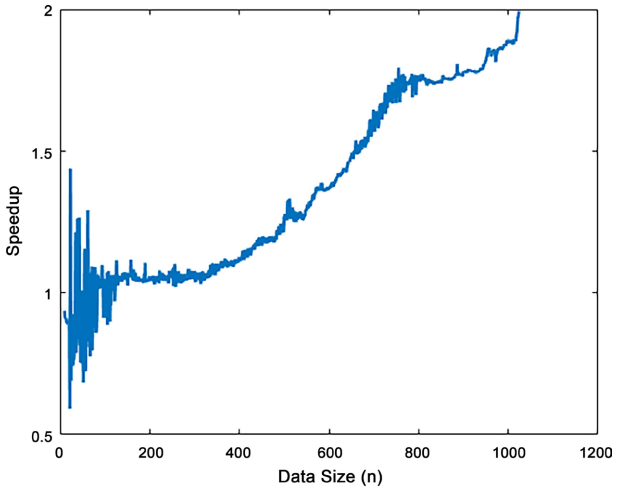

Figure 8. Comparison of CUDA version and Optimized CUDA version.

9. Discussion and Future Work

The remarkable performance improvements achieved by our Data Layout strategy highlight its potential for unlocking the true computational power of GPUs across a wide range of applications. While our case study focused on chained matrix multiplication, the underlying principles of our approach are applicable to various algorithms and data structures that exhibit non-contiguous or irregular memory access patterns. Despite the promising results, certain inherent limitations of our Data Layout technique warrant consideration. Issues such as memory bandwidth constraints, latency, and the non-uniform memory access (NUMA) architecture of GPUs may limit the applicability or performance benefits in some scenarios. Furthermore, the dynamic nature of data access patterns in certain applications could reduce the effectiveness of static data layout optimizations. Looking ahead, our future work will focus on exploring the scalability and effectiveness of our Data Layout strategy in large-scale GPU clusters and cloud environments. By conducting extensive experiments across distributed systems, we aim to provide deeper insights into the potential challenges and opportunities of our approach in these advanced computing paradigms. Additionally, we plan to investigate the integration of our technique with other optimization strategies, such as dynamic data layout transformations and adaptive memory management, to further enhance performance and mitigate the limitations mentioned above.

10. Conclusions

In this study, we presented a novel Data Layout strategy for optimizing CUDA

programs, particularly focusing on dynamic programming algorithms for chained matrix multiplication. By restructuring memory data arrangement to improve locality and coalescing, our approach achieved significant performance enhancements, reducing execution time by up to 50% and memory consumption compared to the baseline implementation. The effectiveness of our technique underscores the importance of architecture-aware optimizations in unlocking the full potential of GPU-accelerated applications. As the adoption of heterogeneous and GPU-based computing continues to grow rapidly across various domains, the principles and strategies discussed in this work will be instrumental in harnessing the benefits of these powerful parallel architectures.

While our findings are promising, we acknowledge the limitations and challenges associated with our approach and emphasize the need for further research to extend its applicability and address potential scalability concerns. By continuing to explore innovative optimization techniques and leveraging the synergies between hardware and software, we can pave the way for more efficient and high-performance GPU computing solutions.

It's important to acknowledge that the scope of our experiments was limited by time constraints, restricting our ability to conduct a more extensive investigation. Future studies will aim to address this limitation by allocating more time for rigorous experimentation and analysis.

Acknowledgements

This work is supported by NSF award #2348330.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Harris, M. (2007) Optimizing Parallel Reduction in CUDA. *Nvidia Developer Technology*, **2**, 70.

[2] Hong, S. and Hyesoon, K. (2010) An Integrated GPU Power and Performance Model. *ACM SIGARCH Computer Architecture News*, **38**, 280-289. https://doi.org/10.1145/1816038.1815998

[3] Volkov, V. and Demmel, J.W. (2008) Benchmarking GPUs to Tune Dense Linear Algebra. *Proceedings of the* 2008 *ACM/IEEE Conference on Supercomputing*, Austin, 15-21 November 2008. https://doi.org/10.1109/SC.2008.5214359

[4] Wu, Y.N., Tsai, P.-A., Muralidharan, S., Parashar, A., Sze, V. and Emer, J. (2023) HighLight: Efficient and Flexible DNN Acceleration with Hierarchical Structured Sparsity. *Proceedings of the* 56*th Annual IEEE/ACM International Symposium on Microarchitecture*, Association for Computing Machinery, New York, NY, USA, 1106-1120. https://doi.org/10.1145/3613424.3623786

[5] Liu, G., *et al.* (2018) A Scalable Parallel Method for Large-Scale Matrix Computations. *The Journal of Supercomputing*, **74**, 6641-6656.

[6] Peled, L., Mannor, S., Weiser, U. and Etsion, Y. (2015) Semantic Locality and Con-

text-Based Prefetching Using Reinforcement Learning. *ACM SIGARCH Computer Architecture News*, **43**, 285-297. https://doi.org/10.1145/2872887.2749473

[7] Aldinucci, M., Drocco, M., Mastrostefano, F. and Vanneschi, M. (2018) Hardware-Conscious Autonomic Management of Distributed Workflows. *International Conference on Algorithms and Architectures for Parallel Processing*, Springer, Cham, 27-31 August 2018, 343-359.

[8] Ballard, G., Zheng, G., Demmel, J. and Yelick, K. (2017) An Efficient and Generic Event-Based Profiling Framework for GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems*, **29**, 169-182.

[9] Li, B.Y., *et al.* (2022) Optimizing Data Layout for Training Deep Neural Networks. *Companion Proceedings of the Web Conference* 2022, New York, April 2022, 548-554. https://doi.org/10.1145/3487553.3524856

[10] Cai, Z., Hu, L., Shi, B., Chen, Y., Hu, C. and Tang, J. (2023) DSP: Efficient GNN Training with Multiple GPUs. *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, Montreal, February 2023, 392-404. https://doi.org/10.1145/3572848.3577528

[11] Wan, L.P., *et al.* (2022) Improving I/O Performance for Exascale Applications through Online Data Layout Reorganization. *IEEE Transactions on Parallel and Distributed Systems*, **33**, 878-890. https://doi.org/10.1109/TPDS.2021.3100784

[12] Stoltzfus, L., *et al.* () Data Placement Optimization in GPU Memory Hierarchy Using Predictive Modeling. *Proceedings of the Workshop on Memory Centric High Performance Computing*, Dallas, November 2018, 45-49. https://doi.org/10.1145/3286475.3286482

[13] Zhong, J.L. and He, B.S. (2014) Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, **25**, 1543-1552. https://doi.org/10.1109/TPDS.2013.111

[14] Neapolitan, R. and Naimipour, K. (2008) Foundations of Algorithms Using C++ Pseudocode. 3rd Edition, Jones and Bartlett Publishers, Inc., Sudbury, USA.

[15] NVIDIA (2023) CUDA C++ Programming Guide (Version 12.2). https://docs.nvidia.com/cuda/archive/12.2.0/pdf/CUDA_C_Best_Practices_Guide.pdf

[16] Segura, A., Arnau, J.-M. and González, A. (2019) SCU: A GPU Stream Compaction Unit for Graph Processing. *Proceedings of the 46th International Symposium on Computer Architecture*, Phoenix, Arizona, 22-26 June 2019, 424-435. https://doi.org/10.1145/3307650.3322254