GAMUT: Matrix Multiplication-Like Tasks on GPUs

Xincheng Xie[1], Junyoung Kim[1] and Kenneth Ross[1]

[1]Department of Computer Science, Columbia University

Abstract

Data scientists are often eager to develop new operators for data science and machine learning applications, which can be written as matrix multiplication-like nested loops. However, these matrix multiplication-like tasks are not supported in standard libraries and often struggle to achieve high performance. Deep learning compilers can optimize some of these operations, but they require significant time to search for optimal configurations. To address these issues, we introduce operators in relational algebra to extend matrix multiplications from a database point of view and develop a matrix multiplication-like task library for GPU. Our library, GAMUT, can recognize and generate optimized code for a variety of tasks, and we propose a tile-based model to incorporate relational algebra operators. Through experiments, we demonstrate that GAMUT achieves high performance with low compilation overhead compared to both matrix multiplication libraries and deep learning compilers.

Keywords

GPU, Data Analytics, Relational Algebra

1. Introduction

As memory capacity rises, more data analytical tasks are entirely put into RAM in order to eliminate the bottleneck of disk I/O. With the utilization of High-Bandwidth Memory (HBM) coupled with thousands of cores [1], graphics processing units (GPUs) have increasingly been used for data science tasks [2]. It can provide these tasks with higher memory throughput and parallelism than the CPU, which facilitates the development of data science, especially machine and deep learning.

Matrix multiplication is one of the most basic building blocks in data science tasks [3]. Many tasks can be represented as simple variants of standard matrix multiplication, which share a similar nested "`for`" loop structure. One typical example is convolution, which usually is implemented as matrix multiplication using toeplitz matrices [4]. Here are two additional examples of matrix multiplication-like tasks (MMLTs).

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < K; k++)
      N[i, j] += S[i, k]*W[k, j] +
        (S[i, k]*W[k, j] > thres[i]) * (S[i, k]*W[k, j] - thres[i]);
```

Figure 1: Example 1. A task that amplifies strong signals and generates a weighted combination of signals in different spectra.

```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < K; k++)
            C[Pzip[i], Rzip[j]] += P[i, k] * R[k, j];
```

Figure 2: Example 2. A task that calculates people's preference for eating at different restaurants.

The first example (Figure 1) comes from Amulet [5, 6], which is used to amplify strong signals in a machine learning application. Suppose that $S[i, :]$ represents a signal vector at location i with K features. Each feature corresponds to an intensity at each spectrum. $W[:, j]$ is a weight vector of observation j. Each observation is a weighted combination of the signal strength of each spectrum. And $N[i, j]$ is the score for each observation j at location i. This query filters out signals of spectra whose weighted intensity is less than the threshold at location i, and doubles the stronger parts of the scores.

Consider another example (Figure 2) from the recommendation system, which analyzes the willingness of people to eat at a restaurant. $P[i, :]$ corresponds to a weight vector of people i's taste. Each feature k can represent people's preference towards degree of spiciness, degree of saltiness, etc. $R[:, j]$ is a style vector of restaurant j, which records the style of dishes this restaurant advocates. $Pzip$ and $Rzip$ are the zip code of people's addresses and restaurants' addresses, respectively. Therefore, $C[Pzip[i], Rzip[j]]$ shows that people living in different areas prefer to eat which area's restaurants.

Although these tasks can be written by similar nested loops with standard matrix multiplications, they are not as well-studied as standard matrix multiplications whose efficient GPU algorithms have already been encapsulated by libraries such as cuBLAS [7] and CUTLASS [8]. Tuning these libraries manually is not flexible enough to fit various MMLTs. This requires a good understanding

Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23) —Workshop on Accelerating Analytics and Data Management Systems (ADMS'23), August 28 - September 1, 2023, Vancouver, Canada

xie.xincheng@columbia.edu (X. Xie); junyoung2@cs.columbia.edu (J. Kim); kar@cs.columbia.edu (K. Ross)

of GPU architecture and the large code base of those libraries.

On the other hand, compilers specifically for deep learning have been developed, such as TVM [9]. New operators can be passed to the compiler as nested loops, and computation optimizations such as loop reordering and tiling are applied to generate efficient GPU codes. Compared with libraries, compilers are more flexible but spend more time on compiling, from several minutes to hours depending on the problem size. And for each size, codes need to be regenerated. Although some works, such as ROLLER [10], try to reduce compilation time, they limit tasks to those that can be expressed as lambda functions.

As a result, MMLTs face the trade-off between implementation flexibility and compiling performance [11]. Matrix multiplication libraries can compile once and run multiple times but with limited extendability; while deep learning compilers can compile more matrix multiplication-like queries but with high compilation time, and the compiled codes cannot be reused for queries with different operand sizes. These issues obstruct the application of matrix multiplication-like operations in the machine learning community due to slow performance and low scalability [5, 6].

In this paper, we propose Gamut[1], a growing library that can generate and optimize GPU codes for MMLTs. We extend the range of MMLT studied in Amulet [5, 6] by introducing operators from relational algebra [12], so MMLTs are not limited to simply changing inner arithmetic. We create a declarative syntax for programmers to specify an MMLT. Our library then parses the inputs and recognizes corresponding operators. To better integrate these operators into current fast matrix multiplication algorithms on GPU, we present a tile-based iterator model which fuses operators into data loading and storing iterators. After compilation into GPU machine code, meta information and object files are stored into our library. In this way, when similar queries, even with different operand sizes, arrive, they are not compiled again but are directly executed in order to amortize compilation overhead.

We study a wide range of MMLT examples. Through experiments, we show that our library performs similarly to matrix multiplication libraries for standard matrix multiplication. For matrix multiplication-like queries, Gamut achieves speedups compared to deep learning compilers while reducing a large amount of compilation time. Our compilation time is stable and manageable for different problem sizes.

The remainder of the paper is structured as follows. After providing background information in Section 2, we define the matrix multiplication-like tasks in Section 3, including the introduction of relational algebra operators that expand the scope of MMLTs. We then discuss the design of Gamut in Section 4 and evaluate its performance in Section 5.

[1]GPUs Applied to MUltiplication-like Tasks

2. Background

In this section, we describe Amulet [5, 6], the prior work of MMLT on CPU and TVM, a deep learning compiler. We also present a brief summary of GPU architecture.

2.1. Amulet

Amulet [5, 6] is built on an open-source compiler that can identify and optimize MMLTs. It applies an adaptive strategy that finds the best configuration during execution. After recognition of MMLT, Amulet transforms the three-level nested loop into a parameterizable tiled loop. At runtime, a small subset of input data is used to find the performant parameters, and then the remaining computation continues with the found parameters.

Amulet achieves speedups compared to the state-of-the-art compiler and decent performance compared to libraries, but it still has some limitations. As discussed in the previous section, Amulet only supports the MMLT that changes the inner arithmetic of the nested loop. In addition, although Amulet explores data parallelism (SIMD, Single Instruction Multiple Data), it is not concerned about task parallelism since CPUs have limited cores. GPUs use Single Instruction Multiple Thread (SIMT) models with thousands of cores, which have higher flexibility than SIMD instructions [13, 14]. This presents an opportunity for extending MMLTs to more diverse queries.

2.2. TVM

TVM [9] is a deep learning compiler that enables graph- and operator-level optimizations for a deep learning model on a wide range of different hardware platforms. Operators are translated into nested multi-level loops. After translation, they use optimizations such as loop tiling, cache read and write, etc., to rearrange the nested loops. TVM also optimizes computation graphs via operator fusion and data layout transformation techniques. The combination of all these techniques introduces a large search space. To search efficiently, hardwares are abstracted into a cost-based model, and TVM uses machine learning-based models such as XGBoost [15] to search optimal configurations. Despite all these search methods, TVM still needs a long time of tuning, ranging from days to weeks, depending on the size of a DNN model [10].

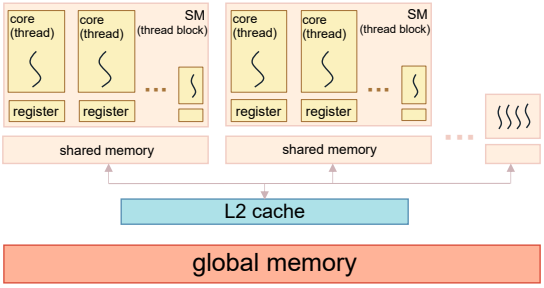

Figure 3: GPU Thread and Memory Architecture. (SM is the abbreviation for Streaming Multiprocessor.)

2.3. GPU Architecture

GPUs provide the ability to run thousands of threads in parallel by leveraging multi-core architecture and high-bandwidth memory [13, 14]. The simplified architecture is shown in Figure 3.

To organize a large number of threads, GPU uses thread hierarchy. The smallest unit of execution is the thread. Thirty-two threads are automatically grouped into a warp by hardware. Threads within the same warp execute the same instruction, known as SIMT (single instruction multiple threads). Multiple warps are then grouped into a thread block. The thread block is the basic scheduling unit and runs on a Streaming Processor. Note that when the resources that a thread block requires exceed the resources a streaming processor can provide, it will fail to run. A program can launch multiple thread blocks to compute.

Memory has a similar hierarchical structure to thread. The largest but slowest one is global memory, which is shared across all streaming processors. Programmers can allocate and manage global memory. L2 cache is also shared by all the thread blocks, but it is not programmable. It is smaller but faster than global memory. L1 cache, also called shared memory in the programming model, is local to all the threads within one thread block. Threads of one thread block cannot access others' shared memory. Different from the CPU's L1 cache, shared memory is explicitly manageable by programmers. The smallest but fastest layer is the register file, which is local to each thread.

3. Extension of Matrix Multiplication

In this section, we try to define matrix multiplication-like tasks and discuss how to apply relational algebra to MMLTs. We will focus our discussion on two-dimensional matrices. For higher-dimensional matrix multiplication, the problem can be decomposed into multiple two-dimensional matrix multiplications, each of which can be launched as a CUDA stream or processed by separate GPUs.

3.1. Generalizing Matrix Multiplication

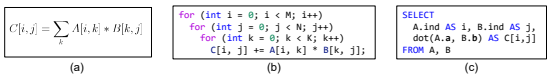

Figure 4: Three representations of standard matrix multiplication. (a) Mathematical Representation. (b) Nested Loop Representation. (c) SQL Representation.

The simplest form of matrix multiplication-like tasks involves all the three nested "`for`" loops, but this broad structure is too general to be optimized effectively. To address this, more specific characteristics are identified and utilized to generalize matrix multiplication.

First, three axes, typically denoted by i, j, and k, are independent. For instance, the starting point of the j axis is not related to the current state of the i axis. These axes can be categorized into two types: two spatial axes (i and j) and one reduction axis (k). The spatial axes correspond to the rows and columns of the resulting matrix. The reduction axis refers to the dimension that is collapsed during the calculation. In standard matrix multiplication, the reduction axis is typically the column of the first matrix and the row of the second matrix. By classifying the axes in this manner, we can categorize the input and output matrices of MMLTs into four types based on their indexing in the inner operator.

1. *Two spatial axes.* It is indexed by i, j or j, i. And this is the only type of output matrix since the reduction axis is always collapsed after calculation.
2. *One spatial axis with one reduction axis.* It is indexed by i, k or k, j and vice versa. The input matrices of standard matrix multiplication can be classified into this type.
3. *Only one spatial axis.* It is indexed by i or j. The input matrix is actually a vector.
4. *Only one reduction axis.* It is indexed by k.

While standard matrix multiplication uses the "`+=`" operator in the inner operation to accumulate the elements along the reduction axis into a single scalar value, MMLTs are not limited to this method. The accumulation operation could take on various forms, such as "`*=`", maximum, average, and so on.

	Nested Loop	SQL

Project	`for i in range(M):`
	`  for j in range(N):`
	`  for k in range(K):`
	`  c += A[i, k] * B[k, j]`
	`  R[i, j] = proj(c)`
		`SELECT`
		`  A.ind AS i, B.ind AS j,`
		`  dot(A.a, B.b) AS c,`
		`  proj(c) AS R[i,j]`
		`FROM A, B`

Select	`for i in range(M):`
	`  for j in range(N):`
	`  for k in range(K):`
	`  c += A[i, k] * B[k, j]`
	`  if c > 10: C_sparse.put(c)`
		`SELECT`
		`  A.ind AS i, B.ind AS j,`
		`  dot(A.a, B.b) AS C[i,j]`
		`FROM A, B`
		`WHERE C[i,j] > 10`

Aggregate	`for i in range(M):`
	`  for j in range(N):`
	`  for k in range(K):`
	`  C[A[i].ai, B[j].bi]`
	`  += A[i, k] * B[k, j]`
		`SELECT`
		`  SUM(dot(A.a, B.b)) AS C[A.ai, B.bi]`
		`FROM A, B`
		`GROUP BY A.ai, B.bi`

Join	`for i in range(M):`
	`  for j in range(N):`
	`  if A[i].ai != B[j].bi: continue`
	`  for k in range(K):`
	`  C[i, j] += A[i, k] * B[k, j]`
		`SELECT`
		`  A.ind AS i, B.ind AS j,`
		`  dot(A.a, B.b) AS C[i,j]`
		`FROM A, B`
		`WHERE A.ai = B.bi`

Figure 5: Conversion between relational algebra operators and nested loops.

3.2. Applying Relational Algebra

The form of MMLT described in the previous section only focuses on changing the inner operator, which is limited. To achieve further generality, we can view MMLT from a database perspective.

Considering a matrix A of size $M \times K$, we can treat it as a database table with M records and K attributes. In the case of standard matrix multiplication, where $C = AB^\top$ and B has size $N \times K$, we can interpret this as each row of the A table being dot producted with each row of the B table, which is equivalent to taking the Cartesian product of the two relations A and B.

Assuming that each table includes an *ind* column indicating the index of each row, $A.ind$ corresponds to i-th row in matrix A and $B.ind$ corresponds to j-th column in matrix $B^\top$. Additionally, let $A.\boldsymbol{a}$ represent a row vector composed of $A.column_1, A.column_2, \cdots, A.column_n$, and $B.\boldsymbol{b}$ is similar. Other columns in the table can represent additional information, which will be discussed later. As a result, matrix multiplication can be expressed as a straightforward SQL query, as shown in Figure 4. With this query as a starting point, we can explore the possibility of incorporating additional relational algebra operators. Next, we will review the frequently used relational algebra operators and discuss how to convert them back into nested loops, shown in Figure 5.

The `Project` operator corresponds to the inner product between two row vectors, and can also denote the operation applied to the resulting scalar output. In ML applications, it can be used to integrate element-wise functions such as activation layers. The `Select` operator determines whether the resulting scalar output is relevant. Typically, the predicate is selective, so the output is often a sparse matrix. In ML, it can function as a filter to eliminate undesired results. The `Aggregate` operator groups together output elements that share the same spatial information, for instance, max pooling in ML. To accomplish this, two additional indexing matrices are required for each spatial axis, which indicate the spatial information (e.g., column $A.ai$ and column $B.bi$). The `Join` operator filters out computations where the operands do not satisfy the predicate, thus performing a check before reduction takes place. It can be used to cluster data in the same group or classification.

We can also incorporate duplicate elimination by utilizing a hash table as an output data structure. For ordering a limited number of output records, we can employ a heap data structure.

4. Gamut Design

In this section, we discuss how we generate code for MMLT and the design of our library.

4.1. Fast Matrix Multiplication on GPU

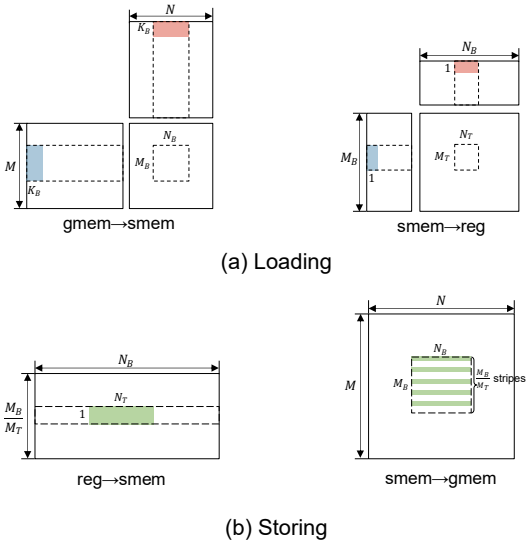

Figure 6: Fast Matrix Multiplication Algorithm on GPU.

We first provide a brief overview of CUTLASS's fast matrix multiplication algorithm on GPU. The whole process is shown in Figure 6

Assume that $C = AB$ has a problem size of $M \times N \times K$, where A is an $M \times K$ matrix, B is a $K \times N$ matrix, and C is an $M \times N$ matrix. The reduction computation of each element of C is independent, allowing for the distribution of computation to separate threads. To maintain good locality, each thread typically computes a block of elements in C of size $M_T \times N_T$. As a result, a thread must load

a $M_T \times K$ submatrix from A and a $K \times N_T$ submatrix from B. However, due to the limited size of registers within a thread, it may not be possible to hold these matrices, particularly if K is large. Therefore, each thread loads only a slice of the matrices at a time, which is a $M_T \times 1$ vector from A and a $1 \times N_T$ vector from B.

Using the thread hierarchy of the GPU, the algorithm groups adjacent threads into thread blocks, assuming a block matrix size of $M_B \times N_B$. Consequently, there are $\frac{M_B}{M_T} \times \frac{N_B}{N_T}$ threads within each thread block. To take advantage of the GPU's memory hierarchy, the algorithm typically fetches data in a step-by-step manner. Initially, one thread block fetches a $M_B \times K_B$ submatrix from A and a $K_B \times M_B$ submatrix from B into the shared memory. Subsequently, each thread retrieves its own vectors to registers and performs the computations.

After the computations are complete, storing the results is also performed in a step-by-step manner. Since each thread has an $M_T \times N_T$ sized register, each thread block produces a $M_B \times N_B$ sized result, which typically exceeds the size of shared memory. Therefore, only a $1 \times N_T$ sized slice of the resulting matrix from each thread is output from the register to the shared memory at a time. Similarly, a $\frac{M_B}{M_T} \times N_T$ sized matrix is output from the shared memory to the global memory. The entire process consists of M_T iterations.

4.2. MMLT Code Construction

Algorithm 1 Main Loop Pseudo-Code

```
1:  for kk := 1 to N step K_B do
2:      Load tiles to shared memory.
3:      TileModelIn(block)
4:      for k := 1 to K_B step 1 do
5:          Load slices to register.
6:          InnerOp(slices)
7:      end for
8:      for i := 1 to M_T step 1 do
9:          Store one row to shared memory.
10:         TileModelOut(block)
11:         Store one block to global memory.
12:     end for
13: end for
```

To better adapt the standard fast algorithm to MMLTs, the algorithm is divided into four components: loading iterators, inner computation, a storing iterator, and a skeleton main loop.

The pseudo-code for the main loop is shown in Algorithm 1, which remains unchanged during code generation. The loading iterator for each input loads a tile of data hierarchically from the global memory to the registers. The inner computation of one thread computes via small data slices, and the results are accumulated repeatedly until the final result is computed and then output hierarchically by the storing iterator.

Due to the hierarchical structure of GPU memory, the iterator consists of two components: thread block-level data transfer between global memory and shared memory, and thread-level data transfer between shared memory and register. Since the data is categorized into four types in Section 3.1, we can design an iterator for each type.

1. *One spatial axis with one reduction axis.* This type can reuse the method that the standard matrix multiplication algorithm uses to load its data. Assuming the size of the spatial axis is P, and the size of the reduction axis is K, the iterator of one thread block loads a $P_B \times K_B$ tile from the global memory to the shared memory. The iterator of one thread loads a $P_T \times 1$ slice from the shared memory to the register.
2. *Only one reduction axis.* Instead of loading a K_B sized vector into the shared memory and then loading one element into the register, we can directly load a K_B sized vector into the register every iteration since K_B is not large.
3. *Spatial axis only.* Since this type of data is irrelevant to the reduction axis, only a single loading operation is required for all the iterations. We can directly load a P_B sized vector into the register once from the global memory in the first iteration. In the following iterations, we can read directly from the register.

The inner computation then computes a block whose problem size is $M_T \times N_T \times 1$. It can directly read from the registers and output to a resulting register sized $M_T \times N_T$. The initialization of the resulting register may differ based on the accumulation operator. For instance, summation initializes the register to all zeros, while the maximum may initialize the register to the minimum number of data precision.

4.3. Tile-based Iterator Model

One simple approach for applying relational algebra operators like `Project`, `Select`, and `Aggregate` involves storing the intermediate results back into global memory before launching another kernel to process them. However, given the opportunity presented by threads within a thread block cooperating to store a tile of data from shared memory to global memory, it is possible to perform data transformations within the iterator. To this end, we propose a tile-based iterator model that integrates these operators with the data loading and storing process.

Table 1
Pesudo Code of each Tile-based Iterator Category

Category	Pseudo Code
Direct Mapping	1: Load ind_{row}, ind_{col} before storing. 2: For Element (i, j), atomicOp($output[ind_{row}[i], ind_{col}[j]]$, $value$)
Linear Mapping	1: Store #output for thread i in $lcounter[i]$. 2: **for** $i := 2$ **to** N_{thread} **do** ▷ Compute prefix-sum. 3: $lcounter[i] = lcounter[i] + lcounter[i-1]$ 4: **end for** 5: $curInd \leftarrow$ atomicAdd($gcounter, lcounter[N_{thread}]$) ▷ atomicAdd returns value before adding. 6: For i-th thread, storing process starts from $curInd + lcounter[i]$.
Nonlinear Mapping	1: **procedure** REG → SMEM 2: localLock.acquire() 3: Store data to *local*. 4: localLock.release() 5: **end procedure** 6: **procedure** SMEM → GMEM 7: **if** threadId == 0 **then** 8: globalLock.acquire() 9: Store data to *global*. 10: globalLock.release() 11: **end if** 12: synchronize() 13: **end procedure**

After obtaining the intermediate results, we identify two types of mappings: one-to-one mapping and many-to-one mapping. One-to-one mapping, like `Project`, can be directly computed in-register after reduction. Our tile-based iterator model, on the other hand, is designed to address the many-to-one mapping.

Tile-based iterators are categorized based on how they index into the output, with the pseudo-code available in Table 1. The first category is direct mapping, as seen in `Aggregate`, where the output position of each element is already known before computation. To facilitate this, two slices of indexing information, one for each spatial axis, each sized P_B, are loaded into shared memory before output. During output, we can use the information in shared memory to index, and because multiple elements are accumulated into one output position, atomic primitives provided by CUDA can be directly used.

The two remaining cases of the tile-based iterator model do not have knowledge of the output position prior to computation. In one of these cases, where the data is output to a linear data structure, such as `Select`, a global counter is used to indicate the next output position. The number of outputs for each thread is computed and stored in a local counter array in shared memory. Then, a single thread computes the prefix sum [16] and reserves a block for the thread block in global memory by incrementing the global counter using "`atomicAdd`". Each thread can then determine its output position by adding the prefix sum of the corresponding thread to the start pointer of the reserved block.

For the last case, where the data structure is nonlinear, like a heap or hash map, a hierarchical structure is used. This involves storing a local data structure for each thread block in shared memory, and a global data structure in global memory, along with a local lock in shared memory and a global lock in global memory. Each thread first acquires the local lock of its thread block, outputs its data to shared memory, and then one thread in the thread block acquires the global lock and merges the data in the local data structure into the global data structure.

We concentrate on natural join for the `Join` operator. Instead of using a hash map in hash join algorithms [17, 18], a set of CUDA streams is created, with each stream focusing on one key. The whole computation is divided into multiple small blocked MMLTs. For one blocked MMLT, we initially retrieve the indices of input matrices that match the key for which this stream is responsible. Thus, we can select data from global memory based on the computed indices instead of creating an additional copy for each blocked MMLT.

4.4. Growing Library

We use a declarative syntax similar to AMULET [5, 6], as illustrated in Figure 7, but with the added requirement for users to specify both the spatial and reduction axes explicitly. The range of var_n in the syntax denotes $[s_n, e_n)$. The loop in Figure 2 is rewritten in declarative form in Figure 8.

GAMUT generates code from the declarative syntax by

```
where (var1 in [s1, e1], var2 in [s2, e2]) when (/* Join Condition */) {
  accum = /* Initialization */;
  reduce (var3 in [s3, e3]) {
    /* Inner Computation */
  }
  /* Output Operator */
}
```

Figure 7: Declarative Syntax of GAMUT Query.

```
where (i in [0, M], j in [0, N]) {
  accum = 0;
  reduce (k in [0, K]) {
    accum += P[i, k] * R[k, j];
  }
  C[Pzip[i], Rzip[j]] += accum;
}
```

Figure 8: Declarative Syntax of Example 2.

first parsing the inner computation and selecting iterators based on the indexing type. Checks are also performed during parsing to ensure the query conforms to the MMLT definition. These checks ensure that there are only three axes and that two spatial axes are associated with the output matrix. Additionally, input matrices are identified as different types, and each input matrix is assigned an input iterator based on its recognized type. Then, the library recognizes the relational algebra operator pattern and selects the appropriate algorithm for that operator, which is fused into the output iterators.

The code generated by GAMUT is parameterized by a set of parameters described in Section 4.1, which is M_B, N_B, M_T, N_T, and K_B. However, not all combinations can be run on a GPU due to limitations on the number of registers, shared memory size, and the number of threads inside one thread block. To reduce the search space, we use a heuristic from CUTLASS that favors square-shaped thread tiles by fixing $M_B : N_B = 1 : 1$ and $M_T : N_T = 1 : 1$. Through experiments, we found that the problem size has little effect on the best configuration when launched thread blocks exceed the number of multiprocessors in GPU. The thread block under the best configuration tends to be close to the largest runnable setting. This is because larger thread blocks result in fewer launched thread blocks and fewer thread blocks running on each multiprocessor, which in turn reduces the overhead of context switching and minimizes cache thrashing while maximizing the throughput. Therefore, we use the scale-up-before-scale-out principle to avoid blind searching.

Upon installation of the library, GAMUT finds the best parameters for standard matrix multiplication and saves them as starting parameters taking into account the resource differences among various GPUs. For a new MMLT query, the library searches for the best configuration starting from the saved parameters. If the starting parameters cannot fully occupy a streaming processor of GPU, GAMUT scales up by increasing the size of tiles. Otherwise, for applications that have more data in each thread block, GAMUT decreases the tile footprint until the setting becomes runnable. The occupation can be calculated using the compilation information provided by the compiler and hardware specialization [19]. If there are multiple optional configurations, GAMUT launches one thread block for each configuration and selects the configuration with the best performance.

GPU code compilation takes a large amount of time compared to computation, even if the parameters are fixed. For example, a standard matrix multiplication with a problem size of $4096 \times 4096 \times 4096$ takes 5 seconds for compilation and only 80 milliseconds for computation. To amortize the compilation time, the parsing and parameter information, along with a hashing summary of the parse tree, are stored with the compiled object file into storage. This information is used to compare parse trees when encountering similar queries, and if a duplicate query is encountered, the compiled object is linked directly, thus saving compilation time.

5. Evaluation

The core of GAMUT is implemented in CUDA C++, and it utilizes a custom parser built by ANTLR [20] and a search algorithm implemented by CuPy [21].

All experiments[2] are conducted on an Nvidia T4 GPU with an Intel Haswell CPU running Ubuntu 22.04 and CUDA 11.7. It is assumed that the GPU memory is sufficient to accommodate all data. Each experiment is repeated 10 times, and the average execution time is reported.

The same performance measurement as Amulet, which is Scaled Processing Rate (SPR), is used [5, 6]. For a problem size of $M \times N \times K$ and execution time T in seconds, SPR is calculated as $\frac{MNK}{10^9 T}$. Higher SPR indicates better performance.

5.1. Standard Matrix Multiplication Performance

To begin with, we demonstrate that our library performs comparably to state-of-the-art matrix multiplication libraries in standard matrix multiplication tasks. We conducted experiments with square matrices (i.e., $M = K = N$) of orders 1024, 2048, 4096, 8192, 16384, and 32768, referred to as 1k, 2k, 4k, 8k, 16k, and 32k, respectively. Three baseline libraries were used in the experiments: cuBLAS, which is the most widely used library for linear algebra on GPUs; CUTLASS, an open-source

[2]Codes are available at https://github.com/xxcisxxc/GAMUT-release

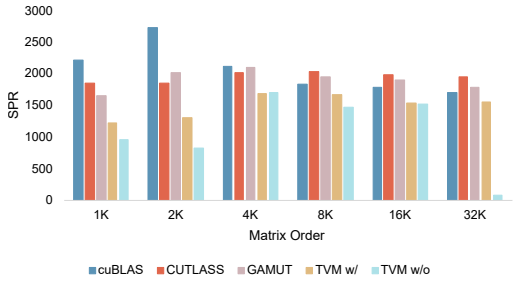

Figure 9: Standard Matrix Multiplication Performance (Higher is Better).

matrix multiplication library on GPUs with similar performance to cuBLAS; and TVM, a deep learning compiler. We test two different settings for TVM: one where we allow TVM to search from scratch with a fixed number of searches, and another where we provide TVM with CUTLASS's default configuration to make an apple-to-apple comparison with our library. As GAMUT focuses on optimizing the standard cubic matrix multiplication algorithm on GPUs, we do not compare it with algorithms such as the Strassen algorithm.

The performance comparison between our library and three other baselines, cuBLAS, CUTLASS, and TVM, is shown in Figure 9. The results reveal that cuBLAS and CUTLASS perform slightly better than our library due to their optimization techniques, such as avoiding memory bank conflict and software pipelining. These techniques, however, require a specific data layout and a larger shared memory footprint, resulting in reduced flexibility. TVM has reasonable performance when the configuration is given. For TVM without configuration, its compilation time increases with the problem size, shown by Table 2. As a result, our library is able to achieve high performance while keeping the compilation time constant and significantly less compared to TVM.

5.2. Performance on General Queries

To investigate the performance of other MMLT queries, we examined a set of queries presented in Figure 10. Queries 1 and 2 are examples from the introduction. Query 3 calculates the number of people-restaurant pairs within two areas where the mutual preference exceeds a threshold. It utilizes the `Project` and `Aggregate` operators. Query 4 only considers people and restaurants in the same area and involves a `Join` operation. Query 5 outputs individual people-restaurant pairs whose mutual preference is above a threshold. We assume that the threshold is selective, meaning only a few pairs qualify, resulting in a sparse matrix. Query 6 is similar to query

```
Query 1
where (i in [0, M], j in [0, N]) {
  accum = 0;
  reduce (k in [0, K]) {
    accum += S[i, k]*W[k, j] +
         (S[i, k]*W[k, j] > thres[i]) * (S[i, k]*W[k, j] - thres[i]);
  }
  N[i, j] = accum;
}

Query 2
where (i in [0, M], j in [0, N]) {
  accum = 0;
  reduce (k in [0, K]) {
    accum += P[i, k] * R[k, j];
  }
  C[Pzip[i], Rzip[j]] += accum;
}

Query 3
where (i in [0, M], j in [0, N]) {
  accum = 0;
  reduce (k in [0, K]) {
    accum += P[i, k] * R[k, j];
  }
  C[Pzip[i], Rzip[j]] += accum > val;
}

Query 4
where (i in [0, M], j in [0, N]) when (Pzip[i] == Rzip[j]) {
  accum = 0;
  reduce (k in [0, K]) {
    accum += P[i, k] * R[k, j];
  }
  C[Pzip[i], Rzip[j]] += accum;
}

Query 5
where (i in [0, M], j in [0, N]) {
  accum = 0;
  reduce (k in [0, K]) {
    accum += P[i, k] * R[k, j];
  }
  accum > thres ? C_sparse.add(accum);
}

Query 6
where (i in [0, M], j in [0, N]) {
  accum = 0;
  reduce (k in [0, K]) {
    accum += max(R[i, k], R[k, j]) * max(R[i, k], R[k, j]);
  }
  accum > thres ? C_sparse.add(accum);
}

Query 7
where (i in [0, M], j in [0, N]) {
  accum = 0;
  reduce (k in [0, K]) {
    accum += P[i, k] * R[k, j];
  }
  min_heap_128.add(accum);
}
```

Figure 10: Studied Queries.

5, but with a different arithmetic, seeking restaurants with complementary styles, where the style vector is normalized. Therefore, two style vectors can be combined through the maximum of each feature, and the dot product of the combined vector will have a high absolute value if the two vectors are complementary. Finally, the last query returns the top 128 mutual preferences, so the heap size is 128.

Different from Amulet [5, 6], Compilation of MMLTs on GPUs is more involved than CPUs because performance requires the efficient use of parallel resources. Tools such as TVM provide infrastructure to help pro-

Table 2
Compilation Time of TVM without configuration.

	1k	2k	4k	8k	16k	32k
TVM w/o-standard	2m 21.4s	4m 34.9s	15m 6.7s	33m 41.8s	47m 40.7s	51m 32.8s
TVM w/o-query 1	2m 29.6s	5m 4.8s	15m 49.9s	42m 26.4s	48m 33.8s	51m 16.7s

Table 3
Compilation Time of standard matrix multiplication for cuBLAS, CUTLASS, and GAMUT

	GAMUT	cuBLAS	CUTLASS
First compilation	3.321s	1.661s	4.933s
Object file exists	0.132s	0.153s	0.157s

Table 4
Compilation time of studied queries for GAMUT

	query 1	query 2	query 3	query 4	query 5	query 6	query 7
First compilation	3.629s	3.413s	3.410s	3.318s	5.425s	5.483s	6.269s
Object file exists	0.141s	0.144s	0.139s	0.138s	0.144s	0.149s	0.192s

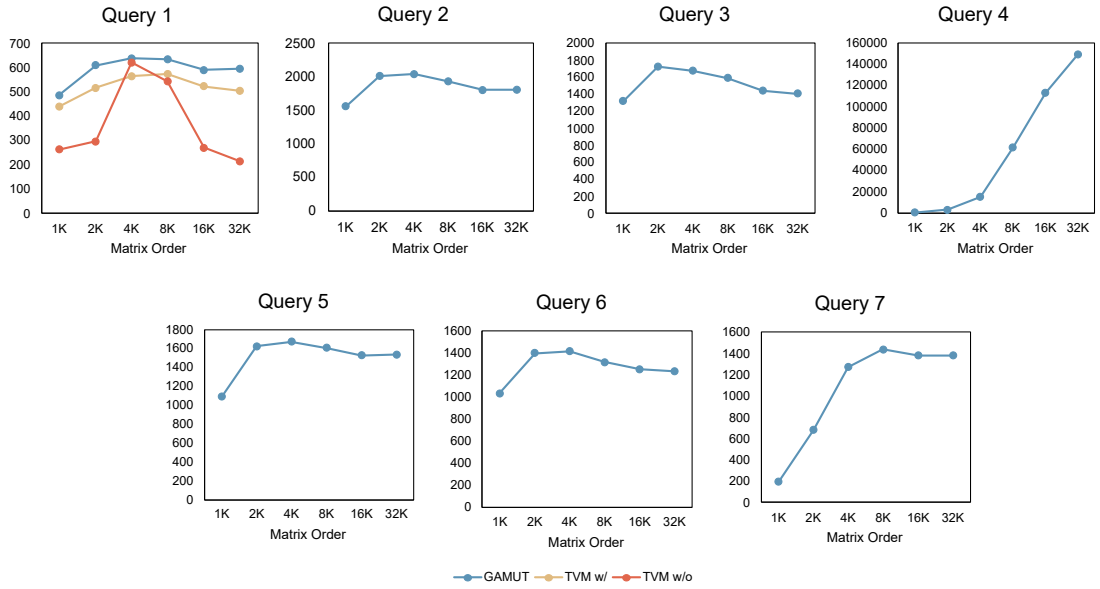

Figure 11: Matrix Multiplication-like Tasks Performance (Higher is better)

grammers avoid low-level parallel programming. However, due to the limitations of TVM, we were only able to compare GAMUT with TVM on Query 1. This is because TVM does not provide access to atomic or lock primitives, as confirmed by the TVM community [22], preventing the compilation of the other queries.

The results of all these queries are shown in Figure 11. It shows that TVM without default configuration has much lower performance. This may be because TVM does not optimize the loading of the extra threshold input.

Query 1 is approximately 3 times slower than standard matrix multiplications, even though it only modifies the inner loop computation. This is most likely because the modified computation does not fit the normal matrix-multiply-add structure that is optimized by specialized hardware in the GPU, leading to lower instruction throughput. Query 4 has a different performance pattern compared to the other queries because it is broken down into multiple smaller matrix multiplications, resulting in less computation than full multiplications. Therefore, it has a much higher SPR and is slower to converge. In the case of Query 7, it exhibits low performance

for small problem sizes due to lock contention. However, when the problem size increases, not all the thread blocks can be launched simultaneously, and contention overhead can be hidden by pipelining the computation with merging the local heap to the global one.

In comparison to Amulet [5, 6], which is based on single-threaded CPU execution, Gamut demonstrates significant improvements, being approximately 50 times faster for standard matrix multiplication and 75 times faster for Query 1. This suggests that GPUs, with their multiple threads and SIMT execution capabilities, show greater potential for parallelism and flexibility in performing a wide range of operators.

6. Conclusion

We propose Gamut, a GPU library for optimizing matrix multiplication-like tasks that parses a custom declarative syntax and generates optimized GPU codes for MMLT. It extends MMLT by introducing relational algebra operators and fuses them into MMLTs using a tile-based iterator model. Our experiments show that Gamut can optimize a wide range of MMLTs and performs similarly to state-of-the-art matrix multiplication libraries, while having faster compilation time and better performance than deep learning compilers. Moreover, it is more flexible and can recognize a larger scope of MMLTs than deep learning compilers, making it a potentially valuable tool for the ML community in developing new operators while achieving high performance with minimal effort. For future work, we plan to broaden the scope of MMLTs and enhance optimization for high-dimensional matrix operations, as well as explore the possibility of incorporating our library into existing deep learning frameworks. Furthermore, we will intend to our library to support various environments, such as embedded GPUs and distributed GPU clusters.

Acknowledgments

This work was supported by the National Science Foundation under grant IIS-2008295 and by a gift from Oracle.

References

[1] A. Shanbhag, S. Madden, X. Yu, A study of the fundamental performance characteristics of gpus and cpus for database analytics, in: Proceedings of the 2020 ACM SIGMOD international conference on Management of data, 2020, pp. 1617–1632.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: a system for large-scale machine learning., in: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16), volume 16, Savannah, GA, USA, 2016, pp. 265–283.

[3] BLASContributors, Blas (Basic Linear Algebra Subprograms), 2022. URL: https://netlib.org/blas/.

[4] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, MIT press, 2016.

[5] J. Kim, K. A. Ross, E. Sedlar, L. Stadler, Amulet: Adaptive matrix-multiplication-like tasks, in: Proceedings of the 19th International Workshop on Data Management on New Hardware, DaMoN '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 77–81. URL: https://doi.org/10.1145/3592980.3595301. doi:10.1145/3592980.3595301.

[6] J. Kim, K. Ross, E. Sedlar, L. Stadler, Amulet: Adaptive matrix-multiplication-like tasks, 2023. arXiv:2305.08872.

[7] NVIDIA, cuBLAS :: CUDA Toolkit Documentation, 2023. URL: https://docs.nvidia.com/cuda/cublas.

[8] A. Kerr, H. Wu, M. Gupta, D. Blasig, P. Ramini, D. Merrill, A. Shivam, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, M. Nicely, CUTLASS, 2022. URL: https://github.com/NVIDIA/cutlass.

[9] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, A. Krishnamurthy, TVM: An automated End-to-End optimizing compiler for deep learning, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), USENIX Association, Carlsbad, CA, 2018, pp. 578–594. URL: https://www.usenix.org/conference/osdi18/presentation/chen.

[10] H. Zhu, R. Wu, Y. Diao, S. Ke, H. Li, C. Zhang, J. Xue, L. Ma, Y. Xia, W. Cui, F. Yang, M. Yang, L. Zhou, A. Cidon, G. Pekhimenko, ROLLER: Fast and efficient tensor compilation for deep learning, in: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), USENIX Association, Carlsbad, CA, 2022, pp. 233–248. URL: https://www.usenix.org/conference/osdi22/presentation/zhu.

[11] T. Faingnaert, T. Besard, B. De Sutter, Flexible performant gemm kernels on gpus, IEEE Transactions on Parallel and Distributed Systems 33 (2022) 2230–2248. doi:10.1109/TPDS.2021.3136457.

[12] A. Silberschatz, H. F. Korth, S. Sudarshan, Database system concepts, McGraw-Hill Education, 2011.

[13] NVIDIA, Programming Guide :: CUDA Toolkit Documentation, 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[14] J. L. Hennessy, D. A. Patterson, Computer architecture: a quantitative approach, Elsevier, 2017.

[15] T. Chen, C. Guestrin, Xgboost: A scalable tree

boosting system, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 785–794. URL: https://doi.org/10.1145/2939672.2939785. doi:10.1145/2939672.2939785.

[16] M. Harris, S. Sengupta, J. D. Owens, Parallel prefix sum (scan) with cuda, GPU gems 3 (2007) 851–876.

[17] C. Balkesen, G. Alonso, J. Teubner, M. T. Özsu, Multi-core, main-memory joins: Sort vs. hash revisited, in: Proceedings of the VLDB Endowment, volume 7, VLDB Endowment, 2013, pp. 85–96.

[18] T. Kaldewey, G. Lohman, R. Mueller, P. Volk, Gpu join processing revisited, in: Proceedings of the Eighth International Workshop on Data Management on New Hardware, 2012, pp. 55–62.

[19] NVIDIA, NVCC :: CUDA Toolkit Documentation, 2023. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.

[20] T. Parr, Antlr (another tool for language recognition), 2023. URL: https://www.antlr.org/.

[21] R. Okuta, Y. Unno, D. Nishino, S. Hido, C. Loomis, Cupy: A numpy-compatible library for nvidia gpu calculations, in: Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS), 2017. URL: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

[22] T. Community, Discussion in tvm community, 2022. URL: https://discuss.tvm.apache.org/t/non-predefined-reduction/13719.