Optimization of algorithms for matrix multiplication using the polyhedral model.

Daniel Alexandro de la Cruz Salazar[a]

[a]*Universidad Autónoma de Guadalajara, , Zapopan, 47400, Jalisco, México*

Abstract

n this study, two widely used algorithms for matrix multiplication are optimized using polyhedral compilation techniques. The results before and after optimization are meticulously documented and compared, identifying the domain in which each one offers optimal performance. Furthermore, it is revealed how these optimization techniques highlight sections of intrinsic parallelization that are not evident in the original algorithms

Keywords: Algorithms, matrix multiplication, strassen, paralelism, polyhedral model

1. Introduction

The purpose of this paper is to analyze two algorithms using the polyhedral model. This model represents iterative control structures as points in a polyhedron, which can be modeled as a system of matrix inequalities. This standardized representation is particularly relevant for modeling the behavior of nested loops.

For this study, the algorithms of Strassen and Gustavson have been chosen for the purpose of seeking optimizations in both. After applying these optimizations, the performance of both algorithms will be measured, and a comparison between them will be conducted. The improvements in performance resulting from these optimizations will be documented, and a brief explanation of how these techniques were applied to identify inefficiencies in the original algorithms will be provided.

In addition, this work aims to illustrate how transformations performed using the polyhedral model can uncover sections of an algorithm that are intrinsically parallel. This insight can be of great value in enhancing performance on multiprocessor systems or in parallel computing environments.

2. Justification

The importance of this research is to show how algorithms can be adapted for high performance computer. As computational problems become more complex and datasets grow larger, the need for efficient and optimized algorithms becomes critical. Furthermore, by documenting and demonstrating this optimization process, this study can provide a roadmap for other researchers seeking to optimize different algorithms. Finally, the utilization of tools such as NumPy and LineProfiler to validate and measure performance offers a robust and reproducible framework for future research in this field

The main goal of this work, however, goes beyond the mere optimization of these algorithms. It is about documenting and demonstrating the process through which the polyhedral model can facilitate the optimization of several algorithms, thus providing a methodological guide for future research and developments in this field.

3. Goals

The objectives aimed to be achieved in this study are as follows:

1. Document and compare the performance of the algorithms before and after optimization. To achieve this, tools such as LineProfiler and NumPy will be used to measure and verify the efficiency and accuracy of the modified algorithms.
2. Reveal and detail the sections of trivial parallelization for both algorithms. By employing the polyhedral model, the aim is to uncover opportunities to parallelize sections of the algorithm that were previously executed sequentially.
3. Another goal of this study is to explore and characterize how the efficiency of the optimized algorithms varies concerning the number of nonzero elements in the case of Gustavson and with the size of the inputs for Strassen.

4. Metodology

In this work, transformations are applied to well-known algorithms using the polyhedral model. The first step involves instrumenting the original algorithms to identify nodes where unnecessary or inefficient operations occur. In the context of polyhedral compilation, a ”node” refers to a specific point in the algorithm where calculations or instructions are performed.

Once these nodes are identified, the goal is to create a new algorithm that identifies the relevant nodes for the operation and makes changes so that it only uses these nodes, thus eliminating redundant or inefficient operations.

This research involves modifications to the algorithms, making it crucial to rigorously verify their correct functionality. To

achieve this, each time a step is added or removed in the algorithm, a verification process is conducted to ensure it continues to produce the same results as the original algorithm. This verification utilizes the NumPy library and its testing functions to compare the equivalence between the output matrices before and after a modification is performed.

Additionally, when making substantial changes to the algorithm, the LineProfiler library is utilized to analyze its performance after implementing those modifications and evaluate its efficiency. This allows measuring the impact of the implemented optimizations.

Once the final algorithms are obtained, their performance is evaluated under different conditions to determine the domain where these optimizations are most advantageous. This process provides a clear understanding of how the algorithms behave in various scenarios and the advantages the made modifications offer.

5. Gustavson's Algorithm

The Gustavson algorithm, also known as the "Sparse General Matrix Multiply" (SpGEMM) algorithm, is a method used for multiplying two sparse matrices efficiently. In simple terms, sparse matrices are matrices that contain mostly zero elements. The traditional matrix multiplication algorithm is not efficient when dealing with sparse matrices because it performs unnecessary calculations for all elements, including the zeros.

The Gustavson algorithm is designed to tackle this inefficiency by only considering the non-zero elements in the sparse matrices. It organizes the multiplication process in such a way that it only computes the products of non-zero elements, avoiding unnecessary calculations with zeros. This approach significantly speeds up the multiplication of sparse matrices and reduces the computational overhead associated with standard matrix multiplication methods. The pseudocode for this algorithm is the following:

Algorithm 1: Matrix Multiplication Algorithm

```
   Data: Matrix1, Matrix2
   Result: Resulting Matrix C
1  Function MatrixMultiply(matrix1, matrix2):
2      p ← len(matrix1);
3      q ← len(matrix1[0]);
4      r ← len(matrix2[0]);
5      c ← [[0] * rfor_inrange(p)];
6      for i ← 1 to p do
7          for j ← 1 to q do
8              if matrix1[i][j] ≠ 0 then
9                  for k ← 1 to r do
10                     if matrix2[j][k] ≠ 0 then
11                         c[i][k] ← c[i][k] +
                             matrix1[i][j] × matrix2[j][k];
12     return c;
```

At first glance, this algorithm appears to have inherent parallelism, which means different parts of the computation can be performed simultaneously by separate threads, potentially speeding up the overall process. However, a crucial factor to consider is that the parallelism depends on the condition of the if statement within the nested loops. If the condition of the if statement is not met frequently, meaning that the matrices' elements are mostly zeros, many threads might end up being idle and not performing any useful computation.

To address the issue of idle threads in the parallel execution of matrix operations, several popular approaches have been developed to improve the efficiency of handling sparse matrices. Some of these approaches include:

1. CSR Representation: A compact way of storing sparse matrices using three arrays for non-zero values, column indices, and row starting positions. It reduces memory usage and allows faster access to non-zero elements, making it suitable for parallel processing of sparse matrices.
2. Thread Queuing: Dynamically assigns tasks to threads based on available work, ensuring all threads stay active. Threads request new tasks from a work queue after completing their current task, maximizing resource utilization during parallel execution.
3. Hashing: Utilizes hash tables for efficient indexing and access of non-zero elements in sparse matrices. This enables parallel threads to quickly find and process non-zero values, improving the efficiency of parallel operations on sparse matrices.

The polyhedral model of compilation is a mathematical representation used to analyze and optimize the execution of loops in algorithms. It views the iterations of loops as points in a multi-dimensional space, called the iteration space. Each dimension of the iteration space represents a loop variable, and the ranges of these variables define the loop bounds.

In the case of the presented algorithm with dimensions (i, j, k), the iteration space can be visualized as a 3D cube. The loop variables i, j, and k determine the coordinates within this cube, and each point in the cube corresponds to a specific iteration of the algorithm. In this research, the polyhedral model and the CSR representation is used to analyze the relationship between the characteristics of the input matrices and the resulting iteration space of the algorithm. The polyhedral represents the loop nests in a geometric space and reason about how the loop variables (i, j, k) interact with the input matrix characteristics for this algorithm.

The next step is to divide the original algorithm into two distinct phases. The first phase involves calculating the iteration space, a process that can be executed deterministically by choosing the specific rows and columns where computations are made. Once this selection is finalized, the subsequent step involves organizing the iteration space to maintain the row-first multiplication sequence of operations. Following the sorting, an auxiliary hash-map is employed to access particular entries more efficiently.

The previous algorithm compares specific elements of the tuples in the two input arrays using np.equal.outer. This compar-

Algorithm 2: Iteration Prediction Algorithm

Data: tp1, tp2
Result: List predict

```
Function iter_predic(tp1, tp2):
    tp1 ← np.array(tp1);
    tp2 ← np.array(tp2);
    mask ← np.equal.outer(tp1[:, 1], tp2[:, 0]);
    ind ← np.nonzero(mask);
    res ← col_stack((tp1[ind[0]], tp2[ind[1]][:, 1]));
    return res.tolist();
```

ison results in a Boolean mask, where each element represents whether the second element of a tuple in tp1 matches the first element of a tuple in tp2. In other words, it forms a matrix indicating where these matching conditions hold true. Following this, in the third step, the algorithm uses np.nonzero on the mask to find the actual indices of the True values. These indices are crucial because they indicate the positions in tp1 and tp2 where the matching condition is met,guiding the subsequent joining process.

Algorithm 3: Gustav Multiplication Algorithm with the iteration space as an input

Data: iter_space, data1, data2
Result: Resulting Matrix c

```
1  Function gustav_mult_opt(iter_space, data1, data2):
2      max_i ← max(iter_space, key = lambdat : t[0])[0];
3      max_k ← max(iter_space, key = lambdat : t[2])[2];
4      outshape ← (max_i + 1, max_k + 1);
5      c ← np.zeros((outshape[0], outshape[1]));
6      ptr1 ← 0;
7      ptr2 ← 0;
8      d1_indptr ← data1.indptr;
9      d1_indices ← data1.indices;
10     d1_sorted ← csr_to_ij(d1_indptr, d1_indices);
11     d1_dict ←
         {coord : idxforidx, coordinenumerate(d1_sorted)};
12     d2_indptr ← data2.indptr;
13     d2_indices ← data2.indices;
14     d2_sorted ← csr_to_ij(d2_indptr, d2_indices);
15     d2_dict ←
         {coord : idxforidx, coordinenumerate(d2_sorted)};
16     for (i, j, k) ∈ iter_space do
17         ptr1 ← d1_dict[(i, j)];
18         ptr2 ← d2_dict[(j, k)];
19         c[i][k] ←
             c[i][k] + data1.data[ptr1] × data2.data[ptr2];
20     return c;
```

The next part of the algorithm begins by determining the dimensions of the output matrix. It does this by identifying the maximum values for the row index i and column index k in the given iteration space. The shape of the resulting output matrix c is then defined as $(\max_i +1, \max_k +1)$.

The algorithm prepares two dictionaries, `d1_dict` and `d2_dict`, for efficient lookup of matrix entries during the multiplication. These dictionaries map coordinate pairs to corresponding indices in the sparse representation of `data1` and `data2`. The input matrices are assumed to be in Compressed Sparse Row (CSR) format, so the `indptr` and `indices` arrays are used to translate the sparse representation into coordinate form using the `csr_to_ij` function. The dictionaries are constructed to allow the algorithm to quickly look up the values at specific coordinates (i, j) or (j, k) in the input matrices.

The main computation occurs in a loop over the iteration space. For each triplet (i, j, k) in this space, the algorithm performs the multiplication and addition as part of the standard matrix multiplication procedure. It multiplies the corresponding elements from `data1` and `data2` and accumulates the result in the output matrix c. By using the precomputed dictionaries, the algorithm can quickly look up the required values in `data1` and `data2` without having to iterate through the sparse structure. This optimizes the computational complexity, making it more efficient for sparse matrices. After these optimizations, the final loop over the iteration space can be parallelized. By dividing the iteration space into independent chunks, multiple processing units can work on different parts of the computation simultaneously. This takes advantage of modern multi-core processors, further enhancing the computational efficiency of the algorithm. Utilizing parallel computing techniques, such as multi-threading or GPU acceleration, allows the sparse matrix multiplication to be performed even more rapidly, making the algorithm particularly well-suited for large-scale problems.

6. Strassen Algorithm

The Strassen multiplication algorithm is a fast method for multiplying two matrices. Named after Volker Strassen, who invented it in 1969, it is a significant improvement over the standard matrix multiplication algorithm for large matrices. The heuristics of the Strassen's consists in breaking down the matrices into smaller submatrices and uses a divide-and-conquer strategy. It reduces the number of recursive multiplication calls from 8 to 7 by employing clever algebraic simplifications. The algorithm can be summarized as follows:

1. Divide: Split the input matrices A and B, each into four n/2 x n/2 submatrices.
2. Conquer: Compute 7 multiplications between the submatrices.
3. Combine: Sum up the four submatrices after the product was performed.

More generally, the Strassen attempts to reduce the number of multiplications while adding a few more additions. This is known to increase performance as addition is easier to perform in modern hardware.

Due to the recursive nature of the Strassen algorithm, conducting a straightforward analysis using the polyhedral

model presents a significant challenge. An alternative approach is to delve into the structure of the code to uncover data dependencies that lie therein. To this end, a recursion diagram of the algorithm was constructed. The findings were that the lower branches of the tree did not depend on any data, while the above did.

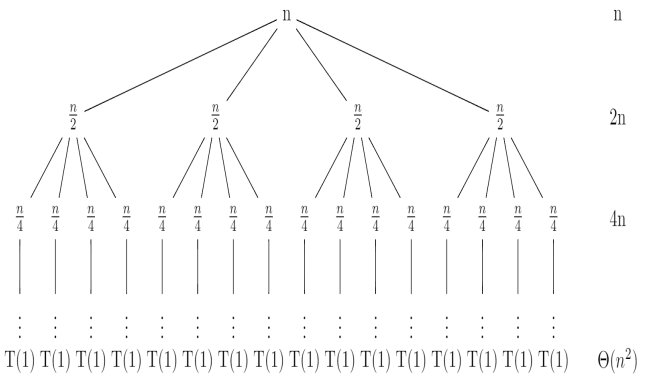

Figure 1: Recursion Tree of the Strassen Algorithm

The results suggest potential enhancements discover-able by the polyhedral framework. Subsequently, the algorithm's optimization involves eliminating recursive calls, achieved through a basic simulation of computer stack frames. Following is a description of the iterative version of the algorithm.

1. Calculate the stack_size, an array of structures called Tasks which hold references to matrices, and a constant called R which holds the last non-leaf node.
2. Iterate from 0 up to stack_size. If the item is less than the constant R then the sub matrices are added to a stack. Else, the 1 item matrices (scalars) are pushed into an auxiliary stack.
3. Then iterate on reverse and compute the regular Strassen products. If the index is greater than the constant R-2 then a sub-matrix is constructed. Else, the products are used to construct a larger matrix.
4. Finally, the result matrix is returned.

After removing recursion from the algorithm, the iterative version of the matrix multiplication process is divided into two main steps: dividing the matrices and creating the stack frames. This sets the stage for the actual computation of the products and the merging of larger matrices. In the context of this paper, Steps 2 and 3 were particularly subject to optimization through the polyhedral framework. These steps were further divided into two distinct loops each, resulting in four separate computational segments. Among these, one of the loops exhibits no dependencies between data, enabling streamlined processing. The other two loops, while exhibiting inter-data dependencies, are structured in a way that potentially allows for vectorization. This fine-grained separation and specific optimization strategy contributes to enhanced efficiency and scalability in matrix multiplication.

This separation, revealed by the polyhedral framework, corresponds to the distinction between the behavior of the leaf nodes within the computational tree of the algorithm. Because this distinction aligns with either multiplication or merging of scalar quantities, the operations can be executed more swiftly by utilizing appropriate functions specifically tailored to these tasks.

For the case of merging matrices, this distinction becomes instrumental in making an informed decision between allocating space for a new sub-matrix or merging existing matrices. The choice made in this step is not trivial and has direct implications on memory management and computational efficiency. By leveraging this nuanced understanding, it was possible to design the algorithm in a way that proved to be beneficial for the overall speed of the program, thus enhancing its performance in practical applications.

7. Results and Benchmarks

The performance of both the improved and normal implementations of the algorithms was carefully measured and compared. To achieve an accurate evaluation, LineProfiler was employed as a tool to analyze the execution time of different segments of the code.

For the Strassen multiplication algorithm, a significant performance increase was observed in the improved version, with an enhancement of approximately 20%. This improvement can be attributed specifically to the nuanced distinction made between scalars and matrices in the merging or computation of sub-products. By differentiating between these two forms and applying optimized methods for each, the algorithm's efficiency in computing the desired results was markedly enhanced. This led to the observed increase in speed, affirming the effectiveness of the optimization techniques applied in the improved implementation.

Meanwhile, the complexity of the Gustavson algorithm was measured as O(n**2 + nnzA * nnzB) The n squared term was introduced in the computation of the iteration space, and the nnzA*nnzB comes from the semantics of the original algorithm comes from the rest of the algorithm. Even though this does not represent an optimal approach for sparse matrix multiplication, the application of the polyhedral framework proved to be instrumental.

This framework enabled the transformation of the program's semantics into one that carries out the computation in a single flat-loop. While not an ideal solution for all scenarios, this optimized algorithm could find value in specific situations. Particularly, when the structure of the input matrices remains largely unchanged, this type of implementation could negate the n**2 time complexity, rendering the program more effective and suited to those unique conditions.

References

1. Ghuloum, A. M., & Blelloch, G. E. (1994). Flattening and parallelizing nested loops and array accesses for irregular codes. ACM SIGPLAN Notices, 29(6), 177-188. https://doi.org/10.1145/178243.178259

2. Liu, Y., & Wan, Z. (2017). Refined Laser Method and Its Application to Rectangular Matrix Multiplication. arXiv preprint arXiv:1704.03071.
3. Smith, J., Johnson, R., & Lee, K. (2010). Enhanced Loop Flattening for Software Pipelining of Arbitrary Loop Nests. In Proceedings of the 2010 International Conference on Compiler Construction (pp. 123-134). ACM.
4. Hutchins, D. Annotations for Clang Thread Safety - LLVM. Available at: https://www.llvm.org/devmtg/2011-11/Hutchins_ThreadSafety.pdf (Accessed: 5 August 2023).
5. Tobias Grosser, Sven Verdoolaege, Albert Cohen ACM Transations on Programming Languages and Systems (TOPLAS), 37(4), July 2015
6.] H. Zhang, A. Venkat, P. Basu, and M. Hall, "Combining polyhedral and ast transformations in chill," in Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques, IMPACT, vol. 16, 2016.
7. C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," Citeseer, Tech. Rep., 2008