Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Instituto de Matemática e Estatística

Alexandre Ribeiro Fernandes Azevedo

Analyzing and Optimizing the Kronecker Tensor Product of Matrices

Rio de Janeiro
2021

Alexandre Ribeiro Fernandes Azevedo

Analyzing and Optimizing the Kronecker Tensor Product of Matrices

Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Orientador: Prof. Dr. Cristiana Barbosa Bentes

Coorientador: Prof. Dr. Maria Clícia Stelling Castro

Coorientador: Prof. Dr. Claude Tadonki

Rio de Janeiro

2021

Alexandre Ribeiro Fernandes Azevedo

Analyzing and Optimizing the Kronecker Tensor Product of Matrices

Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Aprovada em 14 de Setembro de 2021

Rio de Janeiro
2021

RESUMO

O produto de Kronecker, também chamado de Produto Tensorial, é uma operação fundamental da Álgebra Matricial, usada para modelar sistemas complexos usando descritores estruturados. Essa operação precisa ser calculada de forma eficiente, pois é uma peça importante para a modelagem desses sistemas em diversas áreas. Neste trabalho, nos concentramos na análise de desempenho da operação do produto vetor-kronecker, utilizando dois algoritmos diferentes. O primeiro algoritmo, que pode calcular a operação do produto vetor-kronecker para qualquer conjunto de matrizes. O segundo algoritmo realiza o mesmo cálculo, mas é capaz de explorar a esparsidade das matrizes de entrada. Com base na análise detalhada de desempenho, que tem foco nos acessos à memória, propusemos três otimizações: alterar o padrão de acesso à memória, reduzir o desbalanceamento de carga (para a versão paralela) e vetorizar manualmente algumas partes do código com instruções *(*intrinsics) Intel SSE4.2. Os resultados experimentais mostram uma melhor utilização da cache com as melhorias no padrão de acesso à memória e melhorias de desempenho com a vetorização manual.

Palavras-chave: Kronecker Algebra, Multiplicação Vetor-Matriz, Vetorização.

ABSTRACT

The Kronecker product, also called tensor product, is a fundamental matrix algebra operation, used to model complex systems using structured descriptors. This operation needs to be computed efficiently, since it is an important piece for modelling these systems in a number of different areas. In this work, we focus on the performance analysis of the vector-kronecker product operation, using two different algorithms. The first algorithm, that can calculate the vector-kronecker product operation for any set of matrices. The second algorithm, performs the same computation, but is capable of exploiting the sparsity of the input matrices. Based on the in-depth performance analysis, that focus on memory accesses, we proposed three optimizations: changing the memory access pattern, reducing load imbalance (for the parallel version) and manually vectorizing some portions of the code with Intel SSE4.2 intrinsics instructions. The experimental results show better cache usage with the improvements in the memory access pattern and performance improvements with the manual vectorization.

Keywords: Kronecker Algebra, Vector-matrix multiplication, Vectorization.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

SUMÁRIO

INTRODUCTION

Matrix Algebra is widely used in a large number of applications across several fields of Science and Technology [1, 2, 3]. From the most basic elementary matrix multiplications, to more sophisticated matrix operations, they are used for modelling complex physical and digital systems [4, 5]. In the recent decades, a prominent matrix multiplication, known as Kronecker Product, is receiving extra attention from scientists of different areas [6] due to its potential to help the modelling of complex systems.

An important application to the Kronecker Product appears with use of Stochastic Automata Networks for performance modelling issues associated with distributed and parallel computer systems [7]. Several systems have been developed for solving the associated Markov models [8, 9]. Algorithms and implementations that solve VKP are an important tool to these models [10, 11].

The Kronecker product is defined as a block matrix formed with a special multiplication between two matrices [12]. The complexity of forming the kronecker matrix from N matrices of size n_i is defined as $(\prod_{i=1}^{N} n_i^2)$. This complexity can make the matrix construction prohibitive.

In order to reduce the complexity of the matrix construction, some works use a different approach for computing the VKP. Instead of building the full matrix, they exploit the *normal factor* property in order to multiply each matrix individually [10].

In this work, we propose an in-depth study of the performance of two versions of the VKP proposed in [10]: one that uses full matrices as input, called VKP, and one that uses sparse matrix as input, called SpVKP. The performance study focus on the impact of the size and quantity of matrices on the cache memory behavior. Based on this performance study, we propose some optimizations to the two implementations. We improve the data access, and exploit vectorization of the most important loop. For VKP, we also study a parallel implementation and proposed a more efficient load balancing scheme.

We evaluated VKP and SpVKP using different input matrices, each input consists of a number N of matrices with the same size n_i. For VKP, we observed that, since this is a memory bound application, the optimization on the data access produced significant performance gains for large input matrices. The optimization proposed for balancing the load of the parallel application greatly reduced the imbalance of memory accesses among the threads and provided up to 3.9 of speedup. The vectorization also provided performance gains with speedups around 2.5.

For sparse matrices, we observed that SpVKP is consistently faster than VKP. We noticed a dynamic data access pattern during our performance analysis, which means that the memory behavior changes during the execution of each experiment. This changing pattern caused a memory access stride, that is the main focus of our optimizations for this algorithm. The data access optimization had no beneficial impact on the performance. The vectorization of the code was capable of increasing the performance of SpVKP for all input data.

This Dissertation is organized in the following chapters:

- Chapter 2 covers the background for this work, where we presented the definitions for the Kronecker product, the vector-kronecker product multiplication and the algorithms to compute the vector-kronecker product multiplication
- Chapter 3 consists of all the experiments performed using the VKP algorithm. This includes, the memory behavior investigation and all the optimizations performed in the original code.
- Chapter 4 details our efforts analyzing and optimizing the SpVKP Algorithm. This includes a detailed access pattern analysis and the performance optimization experiments.
- Chapter 5 concludes our work and shows possible future works that could be followed from this.

1 BACKGROUND

In this chapter we define the Kronecker product, the Vector-Kronecker Product Multiplication (VKP), and the Sequential and Parallel algorithms to compute the VKP.

1.1 Kronecker product

A vital mathematical operation that gained considerable importance throughout the years and is currently used in several different applications [11, 3] is the kronecker product, or tensor product of two matrices. This multiplication is defined for two matrices of any arbitrary sizes and results in a block matrix [13].

The Kronecker product of two matrices $A \in \mathbb{R}^{n_a \times m_a}$ and $B \in \mathbb{R}^{n_b \times m_b}$ is denoted by $A \otimes B$ and defined as the following $\in \mathbb{R}^{n_a n_b \times m_a m_b}$ matrix:

$$A \otimes B \equiv \begin{bmatrix} a_{11}\mathbf{B} & \dots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \dots & a_{mn}\mathbf{B} \end{bmatrix}$$

Taking two matrices, for instance,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}$$

The kronecker product $M = A \otimes B$ is as follows:

$$\left[\begin{array}{ccc|ccc|ccc}
a_{11}b_{11} & a_{11}b_{12} & a_{11}b_{13} & a_{12}b_{11} & a_{12}b_{12} & a_{12}b_{13} & a_{13}b_{11} & a_{13}b_{12} & a_{13}b_{13} \\
a_{11}b_{21} & a_{11}b_{22} & a_{11}b_{23} & a_{12}b_{21} & a_{12}b_{22} & a_{12}b_{23} & a_{13}b_{21} & a_{13}b_{22} & a_{13}b_{23} \\
\hline
a_{21}b_{11} & a_{21}b_{12} & a_{21}b_{13} & a_{22}b_{11} & a_{22}b_{12} & a_{22}b_{13} & a_{23}b_{11} & a_{23}b_{12} & a_{23}b_{13} \\
a_{21}b_{21} & a_{21}b_{22} & a_{21}b_{23} & a_{22}b_{21} & a_{22}b_{22} & a_{22}b_{23} & a_{23}b_{21} & a_{23}b_{22} & a_{23}b_{23} \\
\hline
a_{31}b_{11} & a_{31}b_{12} & a_{31}b_{13} & a_{32}b_{11} & a_{32}b_{12} & a_{32}b_{13} & a_{33}b_{11} & a_{33}b_{12} & a_{33}b_{13} \\
a_{31}b_{21} & a_{31}b_{22} & a_{31}b_{23} & a_{32}b_{21} & a_{32}b_{22} & a_{32}b_{23} & a_{33}b_{21} & a_{33}b_{22} & a_{33}b_{23}
\end{array}\right]$$

Now let us consider four different matrices M,N,P,Q of compatible orders. Let us also define I_n the identity matrix of order n. We have the following properties of the kronecker product:

Associativity:

$$M \otimes (N \otimes P) = (M \otimes N) \otimes P \tag{1.1}$$

Distributivity with respect to the addition of matrices:

$$M \otimes (N + P) = (M \otimes N) + (M \otimes P) \tag{1.2}$$

Relationship between the kronecker product and ordinary matrix product

$$(M \times N) \otimes (P \times Q) = (M \otimes N) \times (P \otimes Q) \tag{1.3}$$

Compatibility with ordinary matrix inversion

$$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1} \tag{1.4}$$

Compatibility with ordinary matrix transposition

$$(A \otimes B)^T = A^T \otimes B^T \tag{1.5}$$

Factorization

$$A \otimes B = (A \otimes I_{na}) \times (I_{nb} \otimes B) \tag{1.6}$$

By using Equation 1.1, it's possible to define a Kronecker product of N matrices $A^{(i)}$ of size $n_i \times m_i, i = 1,2,...,N$, denoted by $\otimes_{i=1}^{N} A^{(i)}$ which is a matrix of size $\prod_{n=1}^{N} n_i \times \prod_{n=1}^{N} m_i$. [10]

As can be seen above, the Kronecker product can be a considerable challenge to properly handle for both its size and complexity, in an attempt to turn $\otimes_{i=1}^{N} A^{(i)}$ into a more manageable operation, it's necessary to take all matrices $A^{(i)}$ to be square matrices of order $n_i, i = 1,2,3,....N$, this step followed by the use of Equation 1.6, it is possible to write,

$$\otimes_{i=1}^{N} A^{(i)} = \prod_{s=1}^{N} (I_{n1} \otimes ... \otimes I_{ns-1} \otimes A^s \otimes I_{ns+1} \otimes \cdot \otimes I_{nN}) \tag{1.7}$$

also important to mention that the ordinary matrix operation denoted by $\prod_{n=1}^{N}$ is commutative for this special set of factors.

1.2 Vector-Kronecker Product Multiplication

One important operation in tensor algebra is the multiplication of a vector by the kronecker product. As mentioned in the previous section, it can be difficult to construct the kronecker product matrix and some steps are taken in order to compute the multiplication without building the matrix.

Let us take N square matrices A^i of order n_i and a vector $x \in \mathbb{R}^{1 \times L}$ where $L = \prod_{n=1}^{N} n_i$. Our interest is to compute the product,

$$z = x \otimes_{n=1}^{N} A^{(i)} \tag{1.8}$$

The construction of the matrix $\otimes_{n=1}^{N} A^{(i)}$ to perform the multiplication 1.8 would take a large amount of space-memory in a computational environment, where the order of this matrix is $(\prod_{n=1}^{N} n_i)^2$. In addition, the naive approach in building it would yield a prohibitive complexity.

Considering N square matrices $A^{(i)}$ of order n_i, $i = 1,2, \cdots ,N$ a naive computation of the matrix-vector product yield

$$(\prod_{i=1}^{N} n_i)^2 \tag{1.9}$$

floating-point multiplications. Using the recurrent approach with the so-called *normalfactor*, this complexity drops down to

$$(\sum_{i=1}^{N} n_i) \times (\prod_{i=1}^{N} n_i). \tag{1.10}$$

If we consider the case of equal size matrices, i.e. $\forall i \in \{1,2,\cdots,n\}\ n_i = n$, the optimal complexity becomes

$$(Nn) \times (n^N) = Nn^{N+1} \tag{1.11}$$

The number of memory accesses is proportional to the floating-point operations (flops) complexity. However, the sustained performance (effective running time) will depend on the memory access pattern, which depends on the scheduling of the generic computing loop and how the storage is managed between the steps of the main loop.

In our experimental investigations, we might look for scenarios with nearly equivalent complexity. For this reason, we compute in the table 1 the values of the complexity for different scenarios.

N \ n_i	2	4	6	8	10	12	14	16	18	20
10	$2.1 * 10^{4}$	$4.2 * 10^{7}$	$3.6 * 10^{9}$	$8.6 * 10^{10}$	$1.0 * 10^{12}$	$7.4 * 10^{12}$	$4.1 * 10^{13}$	$1.7 * 10^{14}$	$6.4 * 10^{14}$	$2.1 * 10^{15}$
12	$9.8 * 10^{4}$	$8.1 * 10^{8}$	$1.6 * 10^{11}$	$6.6 * 10^{12}$	$1.2 * 10^{14}$	$1.3 * 10^{15}$	$9.5 * 10^{15}$	$5.4 * 10^{16}$	$2.5 * 10^{17}$	$9.8 * 10^{17}$
14	$4.6 * 10^{5}$	$1.5 * 10^{10}$	$6.6 * 10^{12}$	$4.9 * 10^{14}$	$1.4 * 10^{16}$	$2.1 * 10^{17}$	$2.2 * 10^{18}$	$1.6 * 10^{19}$	$9.4 * 10^{19}$	$4.6 * 10^{20}$
16	$2.1 * 10^{6}$	$2.7 * 10^{11}$	$2.7 * 10^{14}$	$3.6 * 10^{16}$	$1.6 * 10^{18}$	$3.5 * 10^{19}$	$4.9 * 10^{20}$	$4.7 * 10^{21}$	$3.5 * 10^{22}$	$2.1 * 10^{23}$
18	$9.4 * 10^{6}$	$4.9 * 10^{12}$	$1.1 * 10^{16}$	$2.6 * 10^{18}$	$1.8 * 10^{20}$	$5.7 * 10^{21}$	$1.1 * 10^{23}$	$1.4 * 10^{24}$	$1.3 * 10^{25}$	$9.4 * 10^{25}$
20	$4.2 * 10^{7}$	$8.8 * 10^{13}$	$4.4 * 10^{17}$	$1.8 * 10^{20}$	$2 * 10^{22}$	$9.2 * 10^{23}$	$2.3 * 10^{25}$	$3.9 * 10^{26}$	$4.6 * 10^{27}$	$4.2 * 10^{28}$

Table 1 – Complexity comparison

1.3 The VKP Algorithm

In [10], Tadonki and Philippe proposed a sequential and parallel implementation of the VKP algorithm. The sequential implementation essentially computes the multiplication of a vector by a kronecker product of matrices, this principal matrix is usually very large and known as descriptor. The descriptor is never completely formed and instead the algorithm performs all the computations using the smaller matrices that form it.

One version of the sequential algorithm is shown in Algorithm 1. The inputs are: N matrices A^p of size $n_p \times n_p$ and a vector X of size $\prod_{p=1}^{N} n_p$. These matrices are used one at a time and which matrix being used is controlled by the outermost loop in line 4. In order to perform all computation for each said matrix, the algorithm takes specific chunks of the main vector V and uses them in the computations, we control the indexes of these specific elements of the vector V with the loops from line 6 and line 7 which we store in our much smaller auxiliary vector U or size n_s in line 8 and 9.

The computations are performed in the two innermost loops which start at line 11 and end at line 16. The computations can be understood as the dot product between the vector U and the current matrix A^s, these will be done for each chunk of V selected in line 8 and 9 and updated back to the main vector at line 15, which is why it is saved in U.

Algorithm 1: Vector-kronecker product multiplication

```
1  V ← X
2  L ← $\prod_{p=1}^{N} n_p$
3  r ← 1
4  for $s \leftarrow N$ to $1$ do
5      l ← $L/n_s$
6      for $k \leftarrow 1$ to $l$ do
7          for $i \leftarrow 1$ to $r$ do
8              for $t \leftarrow 1$ to $n_s$ do
9                  $U[t] \leftarrow V[((k-1)*n_s+t-1)*r+i]$
10             end
11             for $j \leftarrow 1$ to $n_s$ do
12                 for $t \leftarrow 1$ to $n_s$ do
13                     scal ← scal + $A^{(s)}(t,j)*U[t]$
14                 end
15                 $V[((k-1)*n_s+j-1)*r+i]$ ← scal
16             end
17         end
18     end
19     r ← $r*n_s$
20 end
21 Z ← V
```

1.3.1 Shared Memory Parallel Implementation

The shared memory parallel implementation of this algorithm mainly focuses on parallelizing the loop at line 6. The idea is to split equally the computation among the threads, which correspond to the computation of the recursive step s with $A^{(s)}$. This approach has no communications between processors. This algorithm has an inefficiency, where the loops at line 6 and 7 will cause load imbalance among the threads as the loop in line 4 progresses. This is due to the fact that as the execution progresses, l will become smaller and r will become bigger. For the very last matrix, l should be 1 and the execution will be essentially sequential.

Algorithm 2: Parallel Vector-matrix multiplication

```
1  V ← X
2  L ← $\prod_{p=1}^{N} n_p$
3  r ← 1
4  p ← num_threads
5  for $s \leftarrow N$ to $1$ do
6      l ← $L/n_s$
7      #pragma omp parallel for
8      for $k \leftarrow 1$ to $l$ do
9          for $i \leftarrow 1$ to $r$ do
10             for $t \leftarrow 1$ to $n_s$ do
11                 $U[t] \leftarrow V[((k-1) * n_s + t - 1) * r + i]$
12             end
13             for $j \leftarrow 1$ to $n_s$ do
14                 for $t \leftarrow 1$ to $n_s$ do
15                     scal ← scal + $A^{(s)}(t,j) * U[t]$
16                 end
17                 $V[((k-1) * n_s + j - 1) * r + i]$ ← scal
18             end
19         end
20     end
21     r ← $r * n_s$
22 end
23 Z ← V
```

1.4 The Sparse VKP Algorithm

A matrix is considered sparse if many of its elements are zero or in a more general definition, we can consider any matrix to be sparse if its zero elements can be exploited to save computational power[14].

While the algorithm in 1.3 is capable of performing the Sparse Kronecker product-vector multiplication (SpVKP) for any type of matrix, it is not designed to take sparsity in consideration. This means that a large number of trivial multiplications might be performed unnecessarily when the matrix is sparse. In [10], Tadonki and Philippe also proposed an implementation that takes matrix sparsity into account, minimizing the amount of multiplications performed.

The SpVKP algorithm has some differences to the original VKP, the most important one being its possibility of only performing non-zero multiplications. This is done by adding an IF statement that allows the algorithm to avoid all zero elements in the matrices. In order to perform an IF statement before accessing the matrix elements, some modifications were necessary on the algorithm. The VKP algorithm builds U with contiguous parts of V, and performs a dot product of U with A. When the matrix is sparse, this operation should be performed differently to avoid the waste of memory space and unnecessary computation with zero elements. In SpVKP, the auxiliary vector U has to have the same size as V since the elements to be computed are not contiguous. For this reason, the dot product of U and A is not performed anymore. The algorithm performs the multiplication of each non-zero element. In addition, SpVKP needs to define the indexes of the non-zero elements A that will be computed, and the correspondent indexes in V and U for these elements.

The sequential version of this algorithm is shown in Algorithm 3. The inputs are: N matrices $A^{(p)}$ of size $n_p \times n_p$ and a vector X of size $\prod_{p=1}^{N} n_p$. The first for loop, that iterates over s in line 3, chooses which matrix is currently being used to perform the multiplication. Both the main vector and the auxiliary vector are initialized in lines 4 and 5. In line 1 r is initialized and m is initialized with the size of the main vector. m and r are the control variables for the innermost loops and limit the data access pattern, they are updated in lines 6 and 19, respectively. In line 12, we define the index that maps all of the vectors' positions that can be updated by every element of the matrix $A^{(p)}$ which is iterated by the loops in line 7 and 8. In line 12, we define the expression for that index of all vectors' positions that can be updated for each element of the matrix.

Algorithm 3: Sparse Vector-kronecker product multiplication

```
1  r ← 1
2  m ← $\prod_{p=1}^{N} n_p$
3  for $s$ ← $N$ to $1$ do
4      U ← X
5      V ← 0
6      m ← $m/n_s$
7      for $t$ ← 1 to $n_s$ do
8          for $j$ ← 1 to $n_s$ do
9              if $A^{(s)}(i,j) \neq 0$ then
10                 for $k$ ← 1 to $m$ do
11                     for $l$ ← 1 to $r$ do
12                         i ← $l + (j-1) \times r + (k-1) \times r \times n_s$
13                         $V[i] \leftarrow V[i] + A^{(s)}(t,j) \times U[i + (t-j) \times r$
14                     end
15                 end
16         end
17     end
18     U ← V
19     r ← $r * n_s$
20 end
21 Z ← V
```

2 RELATED WORKS

The vector-kronecker product multiplication has been widely used and studied in several different applications within different areas. This chapter presents previous research related to the kronecker product.

2.1 The Kronecker product and formalism

The rudiments of classical tensor algebra first appearance dates from the middle of the XIX century [15]. However, only recently it has gained its popularity among researchers and has been brought to the spotlight. Most recently, it has received a more general formulation, in the Generalized Tensor Algebra, which extends the definition of elements to include *functional elements*. These can be understood as function evaluated in R according to a set of parameters composed of the rows of one or more matrices [16]. The work by Van Loan [17] offers a diverse variety of models and well-known problems related to the kronecker product in different fields of research. Their work confirms a clear growth in the number of applications for the kronecker product.

The work by Brenner *et al.* [3] focuses on mathematical formalism and presents a concise comparison between Generalized Tensor Algebra(GTA) and classical Kronecker modeling of Stochastic Automata Networks. This comparison is justified by a considerable gain in both memory efficiency and lower CPU usage when using GTA formalism.

Dayar and Orhan [12] work is primarily based into optimizing the execution of the shuffle algorithm used in vector-kronecker product multiplication, which was proposed in order to improve the data locality of the algorithm. This optimization is done by proposing a modified version of the algorithm to reduce the number of FLOPS performed, this is accomplished by focusing the computation on the nonzero structure of the matrices and thus trying to avoid FLOPS that use zero rows and columns of these matrices.

2.2 Memory Optimization for the Kronecker product

An important application of the tensor algebra is its use in modeling Continuous Time Markov Chains(CTMC). In CTMC, the kronecker product takes the role of the generator matrix of the markov chain and thus holding all information regarding the system.

Buchholz *et al* [18] work presents a number of different algorithms for solving vector-kronecker multiplication problems. The algorithms are focused on reducing the amount of operations performed and the memory usage, by exploiting the sparsity of the matrices in the Kronecker products. Whenever this sparsity is not present, the algorithm will not perform as well as the original shuffle algorithm, most because the extra steps taken will not return any relevant results.

The work by Benoit *et al*[11] proposes a new algorithm that tries to reduce the memory cost of operating large kronecker product. The algorithm focuses on refining the size of the main vector by removing elements of value equal to zero. This approach allows to handle even larger systems and models. It's important to mention that, due to its refining and necessary reordering, the algorithm has a higher computational cost when compared to the original shuffle algorithm, when the amount of unnecessary computations is small.

The latest work by Buchholz *et al* [1] introduces a new technique for storing solution vectors by using a modern tensor representation called Hierarchical Tucker Decomposition (HTD), which paired with some truncation methods is capable of reducing memory requirements of a vector-kronecker product modelled system and also increasing its time efficiency.

2.3 Parallelization of the vector-Kronecker product multiplication

For large models, the processing time of the vector-Kronecker product multiplication can become considerably high. In this situation, parallel processing can be useful to reduce the computational time and exploit the large availability of parallel machines.

Dolev and Rosen [2] work presents a fully working system for calculating the kronecker product using optical apparatus, where a set of lasers, filters and sensors replace the common silicon processor. This work shows an important step into the diversification of solutions to well known mathematical problems, while still in early stages of development. This appears as exciting news for future of computers. It's interesting to mention that the presence of parallel architecture in such a recent technique confirms the importance of utilizing parallel models and formalism on our current studies.

Tadonki and Philippe [10] work shows shows the fundamental mathematical reasoning and groundwork on which the majority of our work is based on. The main difficulty while solving kronecker product problems is handling the large dimensions of the result matrix. They presented a recurrent algorithm that is capable of performing the vector-kronecker product multiplication by only calculating simple inner products of much smaller portions than the massive result vector. The most important aspect of his work is the parallel algorithms, with two different different approaches. The first one uses the shared-memory model and avoids the communication overhead, but limits itself to a smaller amount of processors. The second approach uses the message passing model and includes communication among processors, this allows the use of a larger amount of processors. The tradeoff in this approach is that the communication overhead can hinder the scalability of the method.

Tadonki latest work [19] proposes an algorithm to allow the vector-kronecker product multiplication to be calculated in large hybrid supercomputers with several nodes, avoiding redundant work to be performed. The focus of this work is the optimization of the communication among processors. The work presents an heuristic algorithm to construct an optimal topology for processors communication, that increases the scalability of the computation of his previous paper. It's important to mention that in this paper, this implementation uses a hybrid parallelization with the shared memory model on the computation nodes, allowing further improvement in the scalability.

Our work is based on these two last papers [10, 19] and extend them. Since the shared memory algorithm plays an important role in reducing the execution time, our goal here is to properly understand the cache usage and employ different optimization techniques to further reduce the computational cost of the problem.

3 ANALYSING THE PERFORMANCE OF THE VKP ALGORITHM

This chapter presents the performance analysis of the VKP algorithm and the optimizations proposed based on this performance analysis. For each optimizaton, we show the performance gains obtained.

3.1 Execution Environment

The experiments were performed on a Intel Core i7 930 with 8Gb of RAM memory running Ubuntu 16.04 LTS linux distribution. They consist of operating a vector-kronecker product multiplication with two square matrices of different order n_i. The implementation of the algorithm proposed in [10] was done in C. We compiled it with gcc version 5.4 using the optimization flag -O3, but we compare the use of -O3 and -O0 flags in section 3.2.3. We collected the L1 Data Cache accesses and hit rate using the Performance Application Programming Interface (PAPI) library [20] that accesses hardware performance information through a set of Performance Monitoring Counters. For the parallel environment, the Intel Core i7 930 has 4 physical cores.

3.2 Memory behavior of the VKP algorithm

In this section, we present a set of preliminary experiments on the cache behavior of Tadonki and Philippe [10].

3.2.1 L1 Cache Hit Ratio

Figure 1 shows the values of L1 hit ratio for different matrix sizes, N varies in $N = \{2, 4,6,...1000\}$. We can observe in this figure a considerable drop in the cache hit ratio when the matrices sizes are roughly bigger than 330×330. The detailed explanation of this drop is related to the inner loop of the algorithm and the hardware prefetching mechanisms.

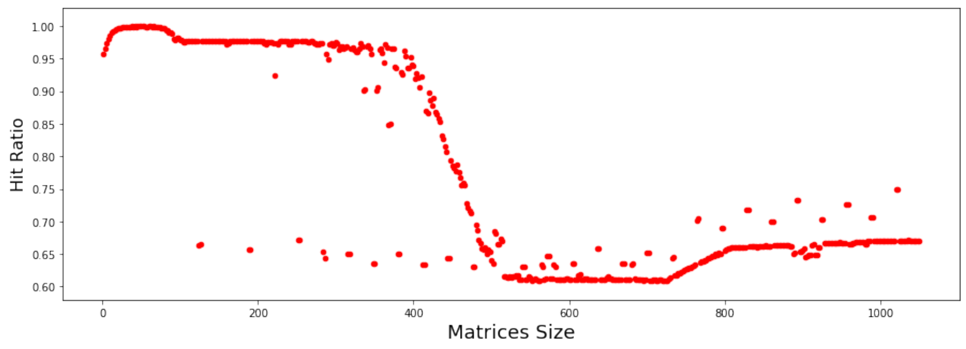


Figure 1 – L1 Hit ratio for increasing matrix sizes

The L1 data cache of the Intel Core i7 930 is a 8-way set associative with 32KB size, dividing this number by size of a float element, 4 bytes, the maximum amount of floats that can be stored in the cache is 8192. The cache line size is 64 bytes thus can store 16 float elements. Intel processors have two types hardware prefetchers for the L1 cache [21]. The L1 hardware prefetchers are called Data Cache Unit (DCU). The first prefecther, called DCU, fetches the next cache line into the L1 data cache. The second prefetcher, called DCU IP, recognizes the load history (based on the Instruction Pointer of previous loads), and this way it can determine whenever to prefetch additional lines with a stride.

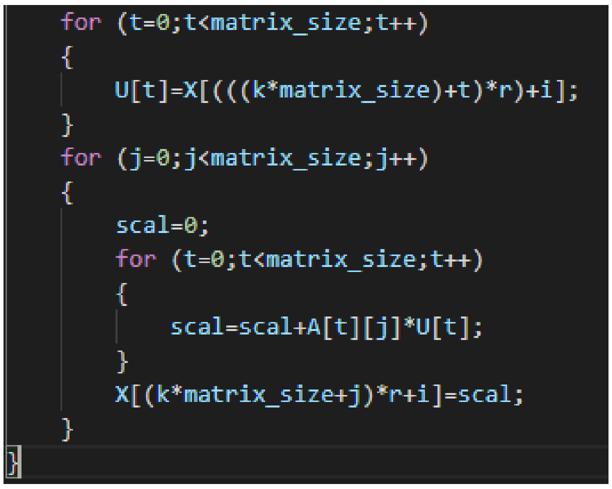
The two innermost loops of the Kronecker product are shown in Figure 2. They comprise the multiplication of the vector U of size n with the matrix A of size $n \times n$. From these two loops, the innermost one comprises a inner product between V and A_j, where A_j is the column j of matrix A. The majority of the cache accesses are concentrated on this innermost loop. In this way, the analysis of the L1 cache behavior focuses on the accesses of the three variables U, A and $scal$ inside this loop. The values of U and $scal$ always fit in L1 for any $n \leq 4000$. For values of $n \leq 90$, the whole matrix A also fits in the L1 cache. For values of $n > 90$, the cache performance depends heavily on the hardware prefetching mechanisms, because A would not fit completely into the L1 data cache.

In the innermost loop of the Kronecker product, A is accessed by columns. The problem is that A is stored by rows in the main memory. So, for each element $A[i,j]$ that generates a cache miss, the miss brings sixteen elements of a row of A in the cache line $A[i,j], A[i,j+1], A[i,j+2], ..., A[i,j+15]$. In addition the DCU prefetcher brings the next cache line, which comprises $A[i,j+16], A[i,j+17], A[i,j+18], ..., A[i,j+31]$. If the DCU IP prefetcher can detect the access pattern of columns of A, it will bring in advance the cache line that contains $A[i+1,j]$, this cache line will bring to L1 the values of $A[i+1,j], A[i+1,j+1], A[i+1,j+2], ..., A[i+1,j+15]$. When N is smaller than 330, what happens is that the values brought in advance will stay on the cache, and when the column $j+1$ is accessed it won't generate misses, since they were brought in advance in the cache lines already prefetched. For $N = 330$, when column j is accessed, the DCU prefetcher brings two cache lines for each access, which will bring a total of $330 \times 32 = 10560$ elements for the whole column, and this would exceed the cache size. So, the access of element $A[i,j+1]$ will generate a miss, since it is not stored in L1 anymore. This explain the drop in the curve after N reaches 330. For values greater than 330, for each access to $A[i,j]$ that generates a miss, the DCU IP prefetcher will bring $A[i+1,j]$. After that, $A[i+2,j]$ will generate a miss and bring $A[i+3,j]$, and so on. In this way, half of the column accesses will generate misses. Since the column accesses represent $\frac{1}{3}$ of the L1 accesses inside the innermost loop, the algorithm will generate around $\frac{1}{6}$ of faults, which gives a hit rate around 80%, as shown in the graph.

3.2.2 Cache Hit Ratio and Execution Time

In order to evaluate how the execution time of the vector-kronecker product is affected by the L1 cache hit ratio, we performed a set of experiments varying the matrices sizes and also varying the number of matrices. The idea is to vary the size of the problem, but maintaining roughly the same the number of memory accesses, this is done by using different input data that has a similar flops complexity. Table 2 shows the flops complexity for this set of input data, where N is the number of matrices and n the size of each matrix.

Table 3 shows for each matrix size and number of matrices, the L1 cache hit ratio, the execution time (in seconds), and the number of cache accesses. This set of experiments



```
for (t=0;t<matrix_size;t++)
{
    U[t]=X[(((k*matrix_size)+t)*r)+i];
}
for (j=0;j<matrix_size;j++)
{
    scal=0;
    for (t=0;t<matrix_size;t++)
    {
        scal=scal+A[t][j]*U[t];
    }
    X[(k*matrix_size+j)*r+i]=scal;
}
}
```

Figure 2 – The two innermost loops of the kronecker product proposed in []

n	N	flops complexity (10^{10})
4	14	1.5
7	10	2.0
11	8	1.9
23	6	2.0
290	3	2.1
2000	2	1.6

Table 2 – FLOPs complexity comparison

had the objective of showing that for a similarly complex experiment, the fact that using matrices larger than 330 has a low hit ratio when compared to several matrices of the same size, thus results in a higher execution time. For this set of experiments, the matrices used for the same execution are of the same size, this means that the number of memory accesses for these experiments is roughly given by $3 \times N \times n^{N+1}$, where N is the number of matrices and n is the size of each matrix.

n_i	N	hit ratio	time(s)	Memory accesses (10^9)
4	14	0.995508	25.031210	52
7	10	0.997272	26.661678	65
11	8	0.998218	23.418367	61
23	6	0.999068	23.707775	64
290	3	0.949813	28.200697	64
2000	2	0.642100	131.148971	64

Table 3 – L1 hit ratio, execution time and number of accesses for different matrix sizes and number of matrices that provide similar amount of computation

3.2.3 Impact of the Optimization Flags

We also evaluate the impact of the optimization flags in the cache behavior. We made the same experiments as the ones in Section 3.2.2 but using also the optimization flag

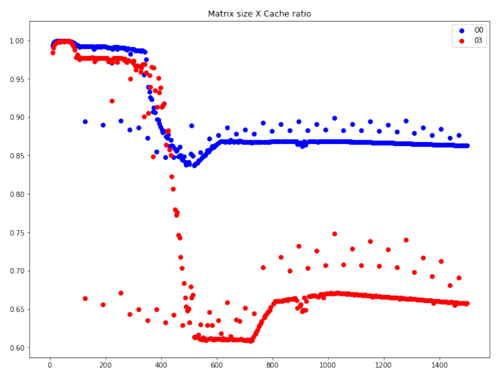
-O0. Table 4 shows the hit ratio, execution time and number of accesses for the same matrices as Table 3 for both the optimization flags -O0 and -O3.

Input Size		-O0			-O3		
n_i	N	hit ratio	time(s)	# accesses(10^9)	hit rate	time(s)	# accesses(10^9)
4	14	0.998923	104.06	220	0.995508	25.03	52
7	10	0.999253	115.72	240	0.997264	26.66	65
11	8	0.999468	101.51	208	0.998218	23.41	61
23	6	0.999702	101.07	203	0.999068	23.70	64
290	3	0.982493	102.64	194	0.949813	28.20	64
2000	2	0.856761	166.02	160	0.642100	131.14	64

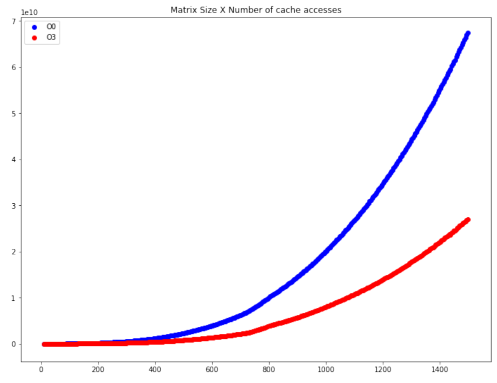
Table 4 – Results using -O0 and -O3 flags

It's interesting to point out that while the -O3 brought a decrease in the execution time, it also caused a drop in the hit ratio for bigger matrix sizes. This behavior could be explained by the -O3 changes to the code generated when compared to -O0. The number of cache accesses reduced drastically from the -O0 implementation to the -O3 implementation, this can be explained by the loop unrolling technique used in the -03 optimization. With loop unrolling, there is no need for indexes access inside the loop. Since the core of the algorithm consists of a nest of loops to compute the kronecker product, if we avoid the accesses to the loop indexes we will reduce number of accesses by roughly 35%. In other hand, by reducing index accesses, we reduce the number of hit without reducing the number of misses and thus reducing our hit ratio. While the -O3 flag enabled auto-vectorization, according to the *gcc* report generated by our code, no vectorization was done by the compiler.

Figure 3 shows two graphs, with different comparisons between both optimization flags, where blue dots are for -O0 data and red dots are for -O3. Figure 3a shows the hit ratio for different matrix sizes, similar to Figure 1. Figure 3b shows the number of accesses to the data cache for different matrix sizes. In these graphs the experiments were once again done with two square matrices of increasing size, N = 2,4,6,..., 1600 so we can keep the loop structure constant.



(a) Hit Ratio



(b) Number of Accesses

Figure 3 – Hit ratio and number of cache accesses for -O0 and -O3 flags

Figure 3a shows the hit ratio drop for both the -03 optimization and -O0 for values of $N > 330$, as it was explained in Section 3.2.1. Figure 3b shows the increase in number of accesses as the matrices size also increase.

3.2.4 Parallelization and Cache

In this section, we assess the impact of parallelization into the cache behavior. The parallel implementation is based on the shared memory Algorithm 2 proposed by Tadonki [10]. This implementation was executed with 4 threads in a 4-core processor.

Table 5 shows the L1 hit rate, time, number of accesses and the speedup for the executions. In this experiment, the same input from the previous section was used.

The hit ratios in table 5 are roughly the same as in the the sequential version, this is due the fact that each thread is pretty much independent from the others. As mentioned in Section 1.3.1 this implementation has a load imbalance, caused by the outermost loops from our Algorithm 2. The reason for this unbalancement is that the controlling variables for these loops are dependant to the matrix being operated, causing the parallel loop to decrease in size as the execution progresses.

n_i	N	L1 hit ratio	time(s)	# accesses (10^9)	Speedup
4	14	0.991617	7.693613	56	3.25
7	10	0.994701	9.093688	68	2.93
11	8	0.996490	8.149652	62	2.87
23	6	0.998020	8.793972	64	2.71
290	3	0.951355	15.478279	64	1.82
2000	2	0.568291	93.319208	64	1.4

Table 5 – Parallel experiment data

3.3 Optimizing the VKP algorithm

Based on the previous analysis of the memory behavior of the Kronecker product, we performed some optimizations to the code in order to increase its performance. This chapter presents these optimizations.

3.3.1 Optimizing the data access

As explained in Section 3.2.1, the innermost loop of the Kronecker product accesses a matrix A by columns. The problem of accessing A by columns is that for $N > 330$, there is a considerable drop in L1 hit ratio. Therefore, the first optimization made to the code is to transpose A and perform the inner loop of the Kronecker product by row, instead by column.

Figure 4 shows the L1 hit ratio of the optimized code as the matrices sizes increase. We can observe in this graph an almost constant high L1 hit ratio, regardless of the matrices sizes. The spike in the beginning of the graph, occurs because for $N < 90$, the whole matrices and vectors fit in L1 (as explained in Section 3.2.1). The comparison of this graph with the graph of Figure 1 shows that transposing A before computing the Kronecker product produces a better L1 performance.

Table 6 shows the hit ratio, number of memory accesses and execution times for the implementations accessing A by rows and by columns. We can observe that our optimization was able to reduce the execution time of the Kronecker product, for all matrices sizes. This reduction is more prominent for bigger matrices.

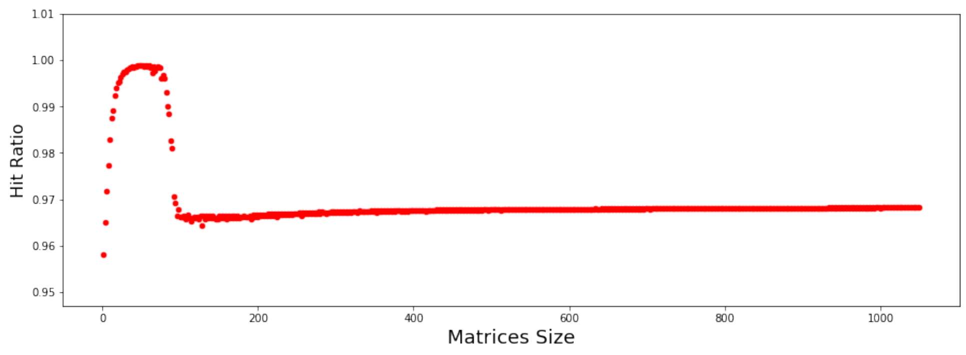


Figure 4 – L1 hit ratio for increasing matrices sizes

One interesting aspect of these results is that the implementation by rows could greatly reduce the number of memory accesses of the algorithm. For $N > 290$, the memory accesses of the implementation by row are half of the memory accesses of the implementation by column. This was an unexpected result.

To explain this behavior, we went through the assembly code of both implementations. We performed two small experiments and analyzed the assembly generated by gcc using -O3 optimization flag. In the first experiment, we used N = 2 and $n_i = 16$. Figures 5 and 6 shows the assembly code of the innermost loops of the Kronecker product when n = 16 for the implementation by column and by row, respectively.

Figure 5 shows that for N = 16, the compiler unrolls the innermost loop. In the code we can observe 16 repetitions of the operation in (3.1).

$$scal = scal + A[i][j] \times U[i] \tag{3.1}$$

These operations are performed by 16 movss instructions to bring the matrix value from memory to a xmm register, 16 mulss instructions to multiply the value of the matrix (in xmm) to the vector element that is in a memory position, and 16 addss results to accumulate the results. The compiler, however, put the instructions in a non-intuitive order in order to take advantage of the pipelines. What we can observe in this code is that, before the innermost loop, the compiler loads the matrix addresses into the registers r8 to r15, rbx, rcx, rbp, rdi, rsi. So, inside the innermost loop, the addresses are not computed. The problem is that, since there is not enough registers for all the addresses, the compiler uses the stack pointer for some addresses as marked in the figure in the red rectangles. So, for each execution of the innermost loop the implementation by column requires 3 more accesses to the memory for accessing the stack.

On the other hand, we can observe in Figure 6 that the implementation by row does not require the load of the matrix addresses to registers. Since the matrix is accessed by row the compiler access it by increasing the offset to the matrix first element by 0, 4, 8, 16,...,60.

Memory accesses in the assembly code are the instructions where one of the operands are registers between parenthesis. Counting the number of accesses of the two implementations, we obtain 16 accesses for the matrix, 16 accesses for the vector, 2 accesses for the indexes summing up 34 accesses. The implementation by columns has 3 more accesses to the stack, which produces roughly 10% more accesses. In this experiment, the number of memory accesses for the implementation by row is 20922, while the number of accesses for the implementation by column is 22891.

We also made a second experiment, where we used $N = 2$ and $n = 32$, in order to investigate the memory accesses when there is not enough registers to hold the addresses to the matrix. Surprisingly, the assembly code generated is completely different. Figures 7 and 8 show the assembly code for the innermost loops of the Kronecker product when N = 32 for column and row implementations, respectively. These figures show that for N = 32, the compiler uses less registers to build the loops. The innermost loop is no longer unrolled. The loop is reduced to 3 instructions: one move, one multiplication and one addition to perform the operation of (3.1), with two jne instructions defining the loop limits.

Despite using less registers, the drawback for the column implementation is that it requires one extra memory access for every loop repetition. The extra access is performed in the *movq* instruction that brings the next matrix memory address to a register. So, with one more memory access within the loop, the increase in the number of accesses in the assembly code is roughly 50%. This experiment got 203693 memory accesses with the column implementation and 140222 memory accesses with the row implementation.

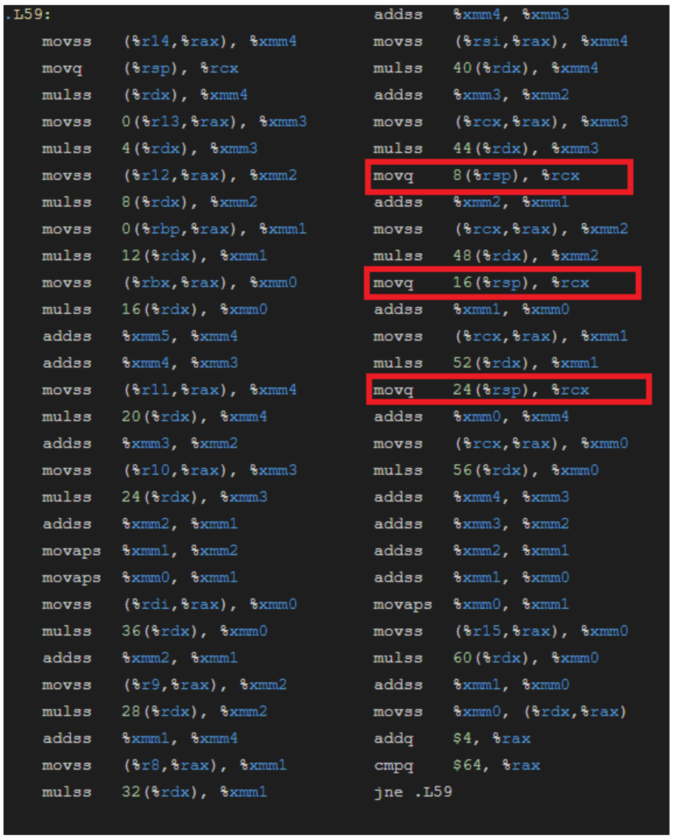
Input Size		Access per row			Access per column		
n_i	N	hit ratio	time(s)	# accesses (10^9)	hit ratio	time(s)	# accesses (10^9)
4	14	0.993969	23.33	40	0.995508	25.03	52
7	10	0.996295	25.38	48	0.997264	26.66	65
11	8	0.999468	21.75	44	0.998218	23.41	61
23	6	0.999702	22.19	44	0.999068	23.70	64
290	3	0.982493	21.51	43	0.949813	28.20	64
2000	2	0.856761	16.91	32	0.642100	131.14	64

Table 6 – L1 hit ratio, number of memory accesses and execution times for the original and optimized code

3.3.2 Load imbalance

As explained in Section 1.3.1, our parallel algorithm contains a load imbalance caused by lines 5 and 19 of Algorithm 1. As the outermost loop progresses, the loops from line 6 and 7 will change. We propose here a solution to the this thread imbalance problem and show this optimization.

Algorithm 4 shows our optimization for Algorithm 2 in order to improve the load balance among the threads. In this optimization, we have to define two different parallel regions, by using an IF statement, comparing the value of l and p, where p is the number of available processors. Whenever $l > p$, our optimized algorithm will perform exactly like Algorithm 2. However, when $l < p$ the parallel region will change to a inner loop allowing all processors to be used until the end of the execution.

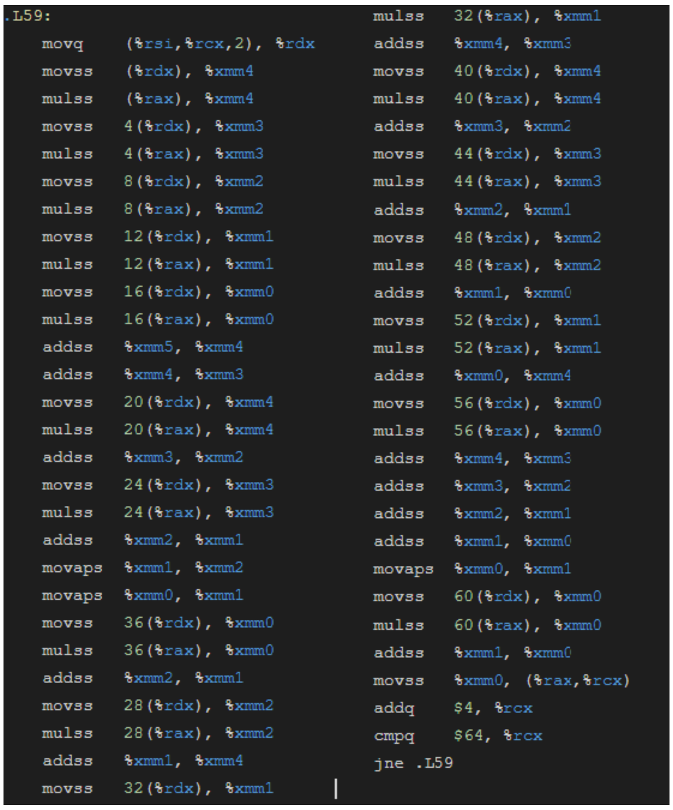


```
.L59:
    movss   (%r14,%rax), %xmm4
    movq    (%rsp), %rcx
    mulss   (%rdx), %xmm4
    movss   0(%r13,%rax), %xmm3
    mulss   4(%rdx), %xmm3
    movss   (%r12,%rax), %xmm2
    mulss   8(%rdx), %xmm2
    movss   0(%rbp,%rax), %xmm1
    mulss   12(%rdx), %xmm1
    movss   (%rbx,%rax), %xmm0
    mulss   16(%rdx), %xmm0
    addss   %xmm5, %xmm4
    addss   %xmm4, %xmm3
    movss   (%r11,%rax), %xmm4
    mulss   20(%rdx), %xmm4
    addss   %xmm3, %xmm2
    movss   (%r10,%rax), %xmm3
    mulss   24(%rdx), %xmm3
    addss   %xmm2, %xmm1
    movaps  %xmm1, %xmm2
    movaps  %xmm0, %xmm1
    movss   (%rdi,%rax), %xmm0
    mulss   36(%rdx), %xmm0
    addss   %xmm2, %xmm1
    movss   (%r9,%rax), %xmm2
    mulss   28(%rdx), %xmm2
    addss   %xmm1, %xmm4
    movss   (%r8,%rax), %xmm1
    mulss   32(%rdx), %xmm1
    addss   %xmm4, %xmm3
    movss   (%rsi,%rax), %xmm4
    mulss   40(%rdx), %xmm4
    addss   %xmm3, %xmm2
    movss   (%rcx,%rax), %xmm3
    mulss   44(%rdx), %xmm3
    movq    8(%rsp), %rcx
    addss   %xmm2, %xmm1
    movss   (%rcx,%rax), %xmm2
    mulss   48(%rdx), %xmm2
    movq    16(%rsp), %rcx
    addss   %xmm1, %xmm0
    movss   (%rcx,%rax), %xmm1
    mulss   52(%rdx), %xmm1
    movq    24(%rsp), %rcx
    addss   %xmm0, %xmm4
    movss   (%rcx,%rax), %xmm0
    mulss   56(%rdx), %xmm0
    addss   %xmm4, %xmm3
    addss   %xmm3, %xmm2
    addss   %xmm2, %xmm1
    addss   %xmm1, %xmm0
    movaps  %xmm0, %xmm1
    movss   (%r15,%rax), %xmm0
    mulss   60(%rdx), %xmm0
    addss   %xmm1, %xmm0
    movss   %xmm0, (%rdx,%rax)
    addq    $4, %rax
    cmpq    $64, %rax
    jne .L59
```

Figure 5 – Assembly code for the innermost loops when N=16, accessing the matrices by column

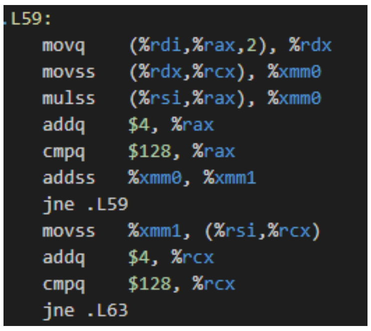
Algorithm 4: Optimized parallel Vector-matrix multiplication

```
1  V ← X
2  L ← $\prod_{p=1}^{N} n_p$
3  r ← 1
4  p ← num_threads
5  for $s \leftarrow N$ to $1$ do
6      l ← $L/n_s$
7      if $(l > p)$ then
8          #pragma omp parallel for
9          for $k \leftarrow 1$ to $l$ do
10             for $i \leftarrow 1$ to $r$ do
11                 for $t \leftarrow 1$ to $n_s$ do
12                     $U[t] \leftarrow V[((k-1)*n_s+t-1)*r+i]$
13                 end
14                 for $j \leftarrow 1$ to $n_s$ do
15                     for $t \leftarrow 1$ to $n_s$ do
16                         scal ← scal + $A^{(s)}(t,j) * U[t]$
17                     end
18                     $V[((k-1)*n_s+j-1)*r+i]$ ← scal
19                 end
20             end
21         end
22     else
23         for $k \leftarrow 1$ to $l$ do
24             #pragma omp parallel for
25             for $i \leftarrow 1$ to $r$ do
```



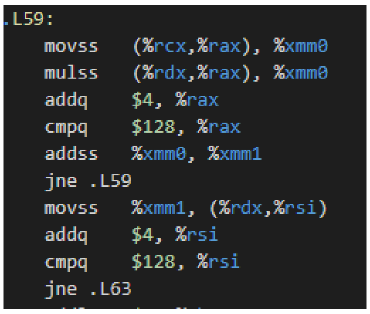
```
.L59:
    movq    (%rsi,%rcx,2), %rdx
    movss   (%rdx), %xmm4
    mulss   (%rax), %xmm4
    movss   4(%rdx), %xmm3
    mulss   4(%rax), %xmm3
    movss   8(%rdx), %xmm2
    mulss   8(%rax), %xmm2
    movss   12(%rdx), %xmm1
    mulss   12(%rax), %xmm1
    movss   16(%rdx), %xmm0
    mulss   16(%rax), %xmm0
    addss   %xmm5, %xmm4
    addss   %xmm4, %xmm3
    movss   20(%rdx), %xmm4
    mulss   20(%rax), %xmm4
    addss   %xmm3, %xmm2
    movss   24(%rdx), %xmm3
    mulss   24(%rax), %xmm3
    addss   %xmm2, %xmm1
    movaps  %xmm1, %xmm2
    movaps  %xmm0, %xmm1
    movss   36(%rdx), %xmm0
    mulss   36(%rax), %xmm0
    addss   %xmm2, %xmm1
    movss   28(%rdx), %xmm2
    mulss   28(%rax), %xmm2
    addss   %xmm1, %xmm4
    movss   32(%rdx), %xmm1
    mulss   32(%rax), %xmm1
    addss   %xmm4, %xmm3
    movss   40(%rdx), %xmm4
    mulss   40(%rax), %xmm4
    addss   %xmm3, %xmm2
    movss   44(%rdx), %xmm3
    mulss   44(%rax), %xmm3
    addss   %xmm2, %xmm1
    movss   48(%rdx), %xmm2
    mulss   48(%rax), %xmm2
    addss   %xmm1, %xmm0
    movss   52(%rdx), %xmm1
    mulss   52(%rax), %xmm1
    addss   %xmm0, %xmm4
    movss   56(%rdx), %xmm0
    mulss   56(%rax), %xmm0
    addss   %xmm4, %xmm3
    addss   %xmm3, %xmm2
    addss   %xmm2, %xmm1
    addss   %xmm1, %xmm0
    movaps  %xmm0, %xmm1
    movss   60(%rdx), %xmm0
    mulss   60(%rax), %xmm0
    addss   %xmm1, %xmm0
    movss   %xmm0, (%rax,%rcx)
    addq    $4, %rcx
    cmpq    $64, %rcx
    jne .L59
```

Figure 6 – Assembly code for the innermost loops when N=16, accessing the matrices by row



```
.L59:
    movq    (%rdi,%rax,2), %rdx
    movss   (%rdx,%rcx), %xmm0
    mulss   (%rsi,%rax), %xmm0
    addq    $4, %rax
    cmpq    $128, %rax
    addss   %xmm0, %xmm1
    jne .L59
    movss   %xmm1, (%rsi,%rcx)
    addq    $4, %rcx
    cmpq    $128, %rcx
    jne .L63
```

Figure 7 – Assembly code for the innermost loops when N=32, accessing the matrices by column



```
.L59:
    movss   (%rcx,%rax), %xmm0
    mulss   (%rdx,%rax), %xmm0
    addq    $4, %rax
    cmpq    $128, %rax
    addss   %xmm0, %xmm1
    jne .L59
    movss   %xmm1, (%rdx,%rsi)
    addq    $4, %rsi
    cmpq    $128, %rsi
    jne .L63
```

Figure 8 – Assembly code for the innermost loops when N=32, accessing the matrices by row

Tables 7 and 8 show the number of memory accesses, execution time and speedup for both the unbalanced version and the optimized version of the code. The optimization has greatly reduced the imbalance in terms of memory access and increased the speedup, especially for the larger matrices.

Input Size			Parallel Implementation	Optimized Parallel Implementation
n_i	N	thread ID	# access (10^9)	# accesses (10^9)
4	14	0	17	14
4	14	1	13	14
4	14	2	13	14
4	14	3	13	14
2000	2	0	40	16
2000	2	1	8	16
2000	2	2	8	16
2000	2	3	8	16

Table 7 – Number of memory accesses of the unbalanced version and the optimized version of the code.

Input Size		Sequential	Parallel		Optimized Parallel	
n_i	N	time(s)	time(s)	speedup	time(s)	speedup
4	14	25.03	7.69	3.25	6.5	3.85
2000	2	131.14	93.32	1.4	38.49	3.4

Table 8 – Execution time and speedup of the unbalanced version and the optimized version of the code.

3.3.3 Vectorization

Another optimization that we implemented in the vector-kronecker product multiplication is to exploit a feature that is very common in modern processors: the ability to perform arithmetic operations in parallel using the vector processing unit.

According to a report generated by gcc, the compiler was unable to vectorize the innermost loop of the Kronecker product. So, we implemented the operation of (3.1) using the Intel SSE intrinsics low-level instructions, which allows for operations of 128-bits variables. Our implementation uses single precision floats, thus all our vectorization uses vectors composed of 4 float elements.

The innermost loop of the Kronecker product is essentially a dot product between a vector U and one of the matrix columns, A_j.

Figure 9 shows our version of the vectorized dot product. It starts by defining four special SSE variables called `_m128`, which is a vector variable with four 32-bits float. The `mm_load_ps` instructions are used to load the vector variables with elements from the vectors A_j and U, denoted in the code as `a` and `b`. Which means that the variables `num1` and `num2` are respectively, portions of `a` and `b`. The `mm_mul_ps` instruction is used to multiply two `mm_128` variables and store the result in another `mm_128` variable, each element i of `num3` will be $a_i \times b_i$. The `mm_hadd_ps` instruction adds adjacent pairs of elements inside the vector variables and stores the results in a new `_m128`.

```
void ccoldot_SP (float *Z, float *a, float *b, int n)
{
    float tempcoldot = 0.0;
    // Defining the vector registers
    __m128 num1,num2,num3,num4;
    // Initializing the register num4 with 0
    num4 = _mm_setzero_ps();

    for(int i=0;i<n;i+=4)
  {
      // Loading the registers num1 and num2 with data from Aj and U
        num1=_mm_load_ps(a+i);
        num2=_mm_load_ps(b+i);

        // Multiplying num1 and num2 (the first step of the dot product).
        num3=_mm_mul_ps(num1,num2);

        // Sum of subtotals from the previous multiplication

        num3=_mm_hadd_ps(num3,num3);
        num3=_mm_hadd_ps(num3,num3);

        // Storing results into a

        num4=_mm_add_ps(num4,num3);
    }
    _mm_store_ss(&tempcoldot,num4);

    *Z = tempcoldot;
}
```

Figure 9 – The innermost loop of the Kronecker product implemented using SSE instructions.

Table 9 shows the execution time results for both the sequential and the vectorized implementations. The performance gains are noticeable for all matrix sizes, and more pronounced for larger matrices. The vectorized implementation uses the optimized version where the matrix is accessed by rows, so part of the gains are due to the gains in memory and cache accesses.

Input Size		Sequential Implementation	Vectorized Implementation	
n_i	N	time(s)	time(s)	speedup
4	14	23.33	14.10	1.77
8	9	13.10	5.53	2.37
24	6	31.18	11.16	2.79
2000	2	16.91	7.00	2.41

Table 9 – Experimental results of vectorization

4 ANALYSING THE PERFORMANCE OF THE SpVKP ALGORITHM

In this chapter, we show the performance analysis of the SpVKP algorithm. We also show the optimization techniques proposed and analyze their peformance impact.

4.1 Execution Environment

To keep our results consistent with the analysis of VKP, all experiments for SpVKP were performed in the same environment as specified in Section 3.1.

4.2 Performance Analysis

In this section, we present the performance of SpVKP compared to VKP. Since they are both memory bound algorithms, we analize the cache behavior of SpVKP and how its memory access pattern changes with different input.

4.2.1 Execution Time

Table 10 shows the execution time (in seconds) of SpVKP compared to VKP for different sizes of N matrices (we used the same matrices sizes as in 3.3.3) with different values of sparsity r. We can observe in this table that for almost all input data, the SpVKP is considerably faster than VKP. We explain this performance results with the memory behavior analysis.

Input Size			VKP			SpVKP		
n_i	N	r	time(s)	Hit Ratio	# (10^9)	time(s)	Hit Ratio	# (10^9)
4	14	0.06	19.19	0.994	40	4.29	0.926	6.2
4	14	0.12	19.29	0.994	40	6.12	0.925	8.5
4	14	0.25	19.22	0.994	40	9.91	0.923	13.4
8	9	0.08	11.71	0.992	23	2.82	0.914	3.2
8	9	0.15	11.65	0.992	23	4.86	0.910	5.3
8	9	0.21	11.67	0.992	23	6.42	0.909	7.0
24	6	0.06	30.39	0.982	60	9.22	0.874	7.4
24	6	0.16	30.38	0.982	60	22.80	0.870	17.0
24	6	0.21	30.35	0.982	60	29.68	0.869	22.4

Table 10 – Execution time of SpVKP compared to VKP

4.2.2 L1 Cache behavior

Table 11 shows the L1 hit ratio and the number of accesses to L1 performed (#) for SpVKP and VKP, for the same input matrices as in Table 10. We can observe in this table that for almost all input data, SpVKP performs only a fraction of the accesses of the original VKP. Although the performance is improved using the SpVKP, the hit ratio has a drop in all tests, which is likely the cause for the lack of performance gain in the case with matrices of size 24 and sparsity 0.21. The hit ratio drop in SpVKP is explained by its intricate memory access pattern.

Input Size			VKP			SpVKP		
n_i	N	r	time(s)	Hit Ratio	# (10^9)	time(s)	Hit Ratio	# (10^9)
4	14	0.06	19.19	0.994	40	4.29	0.926	6.2
4	14	0.12	19.29	0.994	40	6.12	0.925	8.5
4	14	0.25	19.22	0.994	40	9.91	0.923	13.4
8	9	0.08	11.71	0.992	23	2.82	0.914	3.2
8	9	0.15	11.65	0.992	23	4.86	0.910	5.3
8	9	0.21	11.67	0.992	23	6.42	0.909	7.0
24	6	0.06	30.39	0.982	60	9.22	0.874	7.4
24	6	0.16	30.38	0.982	60	22.80	0.870	17.0
24	6	0.21	30.35	0.982	60	29.68	0.869	22.4

Table 11 – L1 hit ratio and number of accesses for SpVKP and VKP

4.2.3 Analysing the access pattern

The access pattern of the SpVKP algorithm, presented in section 1.4, is defined by the for loops of lines 10 and 11. These loops change their limits as the execution progresses. This means that for each matrix we expect these loops to be different and thus the access pattern will also be different. Our algorithm contains two different memory access strides and these strides affects the cache behavior results depending on the size of the matrix being used.

To better understand the effect of the stride on the cache behavior, let us consider an example with a input matrix A of size 4, with only two non-zero elements marked as red and blue circles in the matrix. The resulting vector is an array of size 16. Figure 10 illustrates the matrix A and the vectors V and U used in the multiplication. The element marked in red in A is multiplied it by each red element in the vector U and added to the red elements in vector V. This shows that for each access of a $A(i{,}j)$, we have 4 accesses in V and U, with a stride of 4 (the size of a row). Whenever the size of the matrix row exceeds the size of the cache line, all the accesses in V and U will generate cache misses. In the processor used in our experiments the cache line can hold at most 16 float elements.

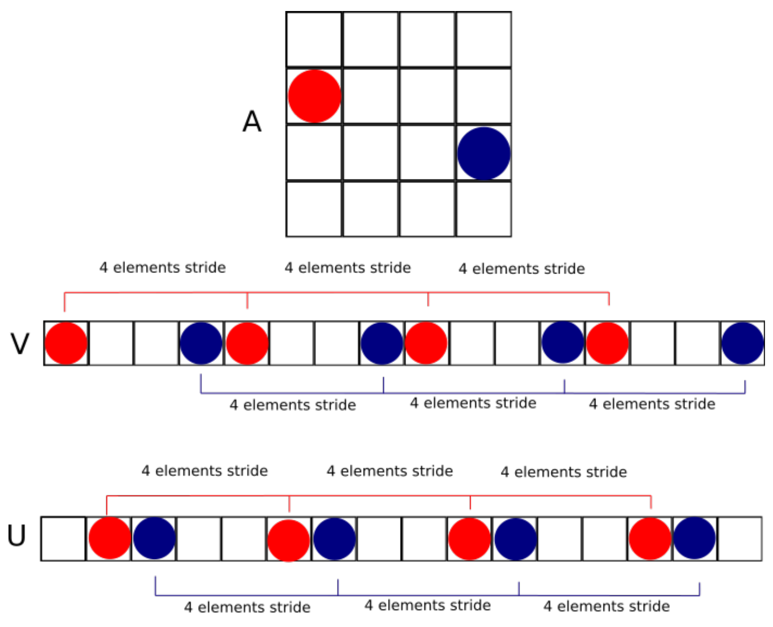


Figure 10 – Data access pattern for s=2

On the following loop iteration (controlled by s), the stride of accesses changes in V and U. We show it in Figures 11 and 12, where two different pairs of A elements are considered. The elements accessed in V and U are contiguous. However, there is a stride between the accesses of elements with different colors in V (for the example of Figure

11) or in U (for the example of Figure 12). In this case, for the accesses of the elements that are contiguous, the bigger the matrix row, the better, since the algorithm take more advantage of the whole cache line for the U and V accesses.

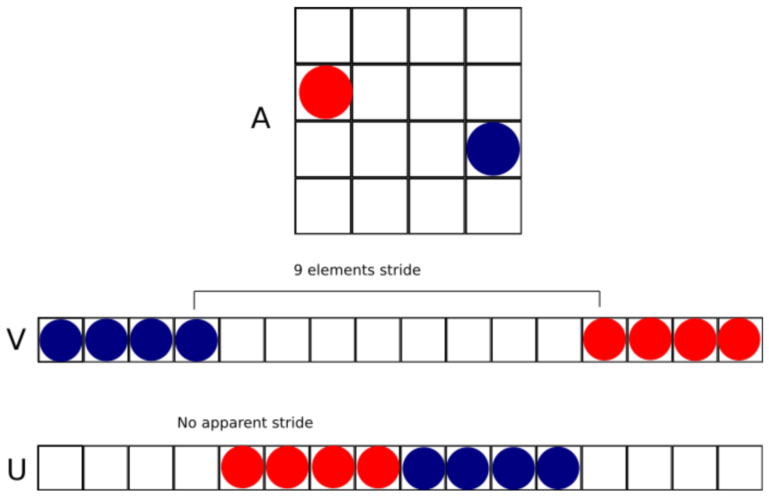


Figure 11 – Data access pattern for s=1

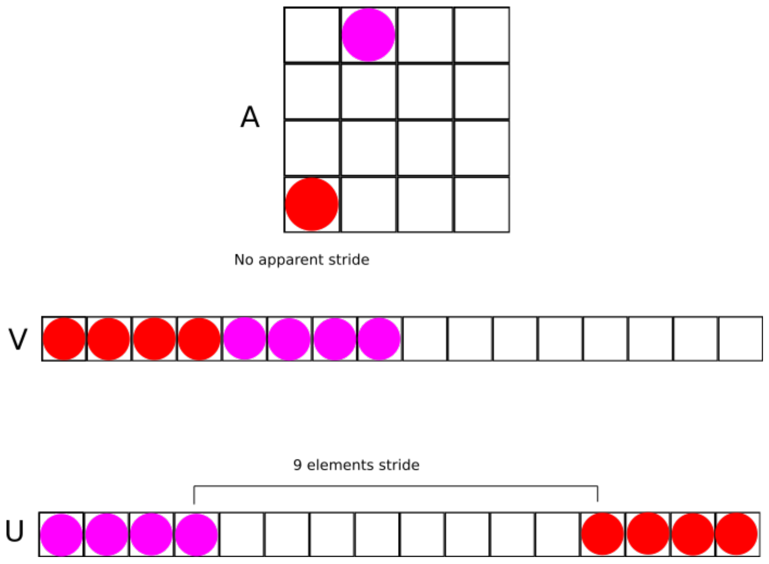


Figure 12 – Data access pattern for s=1

In our tests with matrices of size 24, there's a noticeable drop in the hit ratio and consequently, in the overall performance. For these larger matrices, the stride in the beginning of the execution is very significant. Similarly to what can be seen in Figure 10 most accesses that happen for the first matrix are likely to be a miss. The majority of the accesses happen in the innermost loop, where the multiplication is performed. If we consider that most memory accesses performed while the first, out of six, matrix is being operated are misses due to the pattern stride, the hit ratio drop is explained by this stride.

4.3 Optimizing the SpVKP algorithm

4.3.1 Data storage

The SpVKP algorithm expects that the sparse input matrices are stored in the traditional matrix representation, with all the zero elements. One possible optimization of the SpVKP algorithm is to avoid the IF statement (of line 9) by using a different storage scheme for the sparse matrix. We used a storage scheme known as coordinate format (CF) scheme.

The CF scheme consists of three arrays, AA, JR and JC, of size N_z, where N_z is the number of non-zero elements of the matrix. The arrays contains:

- AA is a real array that holds all of the non-zero elements of the matrix A.
- JR is an integer array that holds the column index of each non-zero element.
- JC is an integer array that holds the row index of each non-zero element.

Algorithm 5 is the SpVKP algorithm adapted to implement the CF storage scheme, that we call CF-SpVKP. In this algorithm the for loops that iterate over the matrix elements (lines 7 and 8 of Algorithm 3) are replaced by a single for loop that iterates over the position of the new array $AA^{(s)}$ (line 7 of Algorithm 5). With these changes, we do not need to test each element because $AA^{(s)}$ contains only non-zero elements.

Algorithm 5: CF-SpVKP

```
1  r ← 1
2  m ← $\prod_{p=1}^{N} n_p$
3  for $s$ ← $N$ to $1$ do
4      U ← X
5      V ← 0
6      m ← $m/n_s$
7      for $j$← $1$ to $N_z$ do
8          for $k$← $1$ to $m$ do
9              J$_z$← $JR^{(s)}(j) - JC^{(s)}(j)$
10             for $l$← $1$ to $r$ do
11                 i ← $l + (JC^{(s)}(j) - 1) \times r + (k-1) \times r \times n_s$
12                 $V[i] \leftarrow V[i] + AA^{(s)}(j) \times U[i + J_z \times r]$
13             end
14         end
15     end
16     U ← V r ← $r * n_s$
17 end
18 Z ← V
```

We compared the original SpVKP algorithm with CF-SpVKP. In our tests, we considered that the input matrices were already stored in a CF scheme.

Table 12 shows the execution time (in seconds), the L1 hit ratio and the number of accesses to the L1 (#) for both the SpVKP and the CF-SpVKP for different sizes of N matrices with different values of sparsity r. The CF storage scheme increased the number of accesses. This occurs because, of the accesses of the index arrays $JC^{(s)}$ and $JR^{(s)}$. Despite the removal of the IF statement, no performance gain was observed, probably the

Input Size			SpVKP			CF-SpVKP		
n_i	N	r	time(s)	Hit Ratio	# accesses (10^9)	time(s)	Hit Ratio	# accesses (10^9)
4	14	0.06	4.29	0.926	6.2	4.35	0.926	7
4	14	0.12	6.12	0.925	8.5	6.34	0.924	10
4	14	0.25	9.91	0.923	13.4	10.30	0.923	16
8	9	0.08	2.82	0.914	3.2	2.91	0.913	4
8	9	0.15	4.86	0.910	5.3	5.22	0.910	7
8	9	0.21	6.42	0.909	7.0	6.76	0.909	9
24	6	0.06	9.22	0.874	7.4	9.39	0.875	9
24	6	0.16	22.80	0.870	17.0	23.16	0.870	22
24	6	0.21	29.68	0.869	22.4	30.62	0.868	29

Table 12 – Execution time and cache behavior of CF-SpVKP compared to SpVKP

branch prediction of the Intel processor is working well for SpVKP and the cost of the IF statement is not that pronounced.

4.3.2 Vectorization

Another optimization performed on the SpVKP algorithm to improve its performance is to exploit vectorization.

4.3.2.1 SpVKP Vectorized

In order to exploit the SIMD instructions present in our processor, the elements of vector V and U should be accessed continuously. As explained in 4.2.3, after the first matrix is operated, the accesses are continuous in V and U for the multiplication of each matrix element. Therefore, we can vectorize the innermost loops using the SSE instructions:

- $\_mm\_load\_ps$ is used to load four packed single-precision float elements into a 128-bit *xmm* register
- $\_mm\_load1\_ps$ is used to load one packed single precision float into all 4 positions of the 128-bit *xmm* register
- $\_mm\_mul\_ps$ is used to multiply the contents of two packed single-precision in a *xmm* register
- $\_mm\_add\_ps$ is used to add the contents of two packed single-precision in a *xmm* register
- $\_mm\_store\_ps$ is used to store four packed single-precision float elements into the specified memory address

Figure 13 shows the innermost loops of our code, a set of *xmm* registers are used to store 4 packed single-precision float elements from each array U and V and add their contents together after the the matrix-array multiplication.

Explicar as variáveis.

The Vectorized SpVKP algorithm was compared to SpVKP and the results are shown in Table 13. This table shows the execution time (in seconds), the L1 hit ratio and the number of accesses to the L1 (#) for both the SpVKP and Vectorized SpVKP using different sizes n_i of matrices N and sparsity r. The vectorization of the innermost loop gave performance gains and decreases in the number of memory accesses.

The drop in the accesses can be explained by the usage of the $\_mm\_load\_ps$ instruction, which instead of loading a single element for every access, it loads 4. As expected,

```
//Innermost Loops of the Vectorized SpVKP code
            for (int k=0;k<m;k++)
              {
                int p = (j*r)+(k*r*matrix_size);

                for (int l=p;l<p+r;l+=4)
                {
                  //Initializing xmm register variables
                  __m128 a,b,num1,tmp;

                  //Load array contents in to the xmm register variables
                  a=_mm_load_ps(V+l);
                  b=_mm_load_ps(U+(l+(t-j)*r));

                  //Load the matrix A elements for multiplication

                  tmp=_mm_load1_ps(&A[t][j]);

                  //Perform the multiplication and sum of the data.
                  //**Refer to the SpVKP Algorithm for details

                  b=_mm_mul_ps(b,tmp);
                  num1=_mm_add_ps(a,b);
                  a=num1;

                  //Store the results back into array V
                  _mm_store_ps(V+l,a);
                }
              }

```

Figure 13 – The innermost loop of the Kronecker product implemented using SSE instructions.

Input Size			SpVKP			Vectorized SpVKP		
n_i	N	r	time(s)	Hit Ratio	# accesses (10^9)	time(s)	Hit Ratio	# accesses (10^9)
4	14	0.06	4.07	0.926	6.2	3.70	0.915	5
4	14	0.12	5.93	0.925	8.5	5.16	0.909	6.7
4	14	0.25	9.79	0.923	13.4	8.09	0.902	10
8	9	0.08	2.68	0.914	3.2	2.28	0.894	2.5
8	9	0.15	4.62	0.910	5.3	3.86	0.884	4
8	9	0.21	6.20	0.909	7.0	5.08	0.881	5.1
24	6	0.06	8.70	0.874	7.4	7.75	0.797	4.4
24	6	0.16	21.51	0.870	17.0	19.06	0.776	9.8
24	6	0.21	28.07	0.869	22.4	24.85	0.771	12.5

Table 13 – Execution time and cache behavior of Vectorizes SpVKP compared to SpVKP

this caused a higher impact on larger matrices due the fact that they will better exploit the cache line, when compared to smaller matrices.

We observed that the hit ratio for Vectorized SpVKP has decreased when compared to SpVKP. This occurs because the number of accesses is reduced, but the accesses that were avoided were hit accesses as shown in Table 14. This table shows the number of accesses and the number of L1 misses for both SpVKP and Vectorized SpVKP. This is the reason why the performance gains of Vectorized SpVKP against SpVKP are small.

Input Size			SpVKP		Vectorized SpVKP	
n_i	N	r	# misses (10^8)	# accesses (10^9)	# misses (10^8)	# accesses (10^9)
4	14	0.06	4.5	6.2	4.2	5
4	14	0.12	6.4	8.5	6.1	6.7
4	14	0.25	10.3	13.4	9.8	10
8	9	0.08	2.8	3.2	2.6	2.5
8	9	0.15	4.8	5.3	4.6	4
8	9	0.21	6.4	7.0	6.1	5.1
24	6	0.06	9.3	7.4	8.9	4.4
24	6	0.16	2.2	17.0	2.1	9.8
24	6	0.21	2.9	22.4	2.8	12.5

Table 14 – Number of misses for Vectorired SpVKP and SpVKP.

4.3.2.2 Exploiting Shuffle operations

Going further into vectorizing SpVKP, we implemented the first loop of VKP (line 3 of Algorithm 3) in a different way. The first iteration is the one with the biggest access stride. So, in this iteration we exploited shuffle operations to reduce the stride in the acccesses. The other iterations are vectorized as in 4.3.2.1.

Due to our processor being an older model, we only have access to SSE instructions, which limits our SIMD operations to *xmm* 128 bit registers. For this, we only performed shuffling operations for matrices of size 4.

By loading the first 16 elements of V and U into 8 *xmm* registers we can shuffle the contents of the vectors. The idea is to store continuously the elements that will be accessed to multiply one element of A. Figure 14 shows an example of using the shuffle to maintain the data continuous. The elements with the same color will be accessed consecutively to perform a multiplication of one element of A, but they are not stored continuous in memory. After two shuffle operations are performed, the elements that are accessed consecutively are store continuously in memory. After shuffling the data, we implemented the same vectorization scheme of the one presented in Section 4.3.2.1.

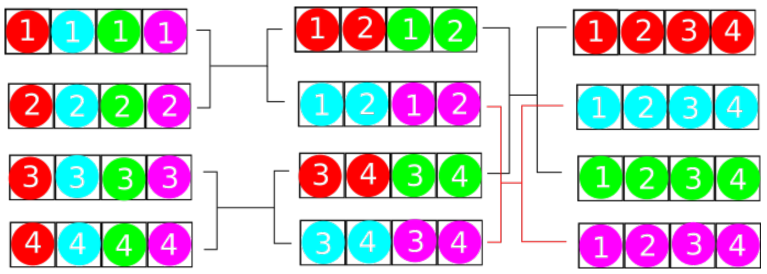


Figure 14 – Shuffle operations in the vector V and U

Input Size			SpVKP			Vec-Shuffle SpVKP		
n_i	N	r	time(s)	Hit Ratio	# accesses (10^9)	time(s)	Hit Ratio	# accesses (10^9)
4	14	0.06	4.07	0.926	6.2	4.91	0.899	5.7
4	14	0.12	5.93	0.925	8.5	6.2	0.895	6.9
4	14	0.25	9.79	0.923	13.4	8.65	0.891	9.5

Table 15 – SpVKP vs Vec-Shuffle SpVKP

Table 15 shows the time (in seconds), Hit ratio and number of accesses (10^9) for both the SpVKP and the Vectorized Shuffled SpVKP (called Vec-Shuffle SpVKP). The shuffle only brought an increase to performance for the matrices with sparsity 0.25, This is due the lack of elements to exploit the shuffling for inputs with a smaller value of r. Considering only 1/4 of the array's elements will actually be used, most of the data shuffled will not used. It is also important to take note that we had to unshuffle the arrays once the multiplication was performed, so the data was correctly stored back in memory.

CONCLUSIONS

This work proposed an in-depth investigation of the performance of the VKP multiplication in terms of memory behavior and computational power. This investigation considered two different algorithms for full (VKP) and sparse matrices (SpVKP). The performance analysis focused on identifying possible optimizations that could reduce computational cost and memory usage.

From this investigation, we proposed three optimizations to the original VKP algorithm and two for the SpVKP algorithm.

For the VKP, the first optimization focused on changing the data access pattern in order to improved the cache usage of the code. The second focused on reducing a load imbalance present in the parallel implementation. The last one was to manually vectorize the innermost loop. It was observed during our experiments that the compiler was unable to auto-vectorize this loop.

We found that the load imbalance and the vectorization brought a significant improvement to the execution time. The load imbalance was mostly removed for our set of input data. The vectorization brought a significant increase to the code efficiency for different the input sizes. The changes to the data access had impact on the performance only for larger matrices. Anyhow, the access by rows reduce the number of memory accesses for all cases.

For the SpVKP, the first optimization was to implement the algorithm using a different storage scheme, called Coordinate format (CF), in order to avoid an IF statement in the code that could cause branch prediction misses. This change did not caused any beneficial impact on the results, causing the number of accesses to increase and thus the performance to decrease.

We conclude that VKP and SpVKP are memory bound algorithms that are very sensitive to memory oriented optimizations. Changing the data access pattern and manually vectorizing the code could provide significant performance improvements.

For future work, we intend to improve the performance of VKP by exploiting the fine grain parallelism of GPUs and using the more advanced AVX instructions for manual vectorization.

Bibliografia

[1] Peter Buchholz et al. “On compact solution vectors in Kronecker-based Markovian analysis”. Em: *Performance Evaluation* 115 (2017), pp. 132–149.

[2] Shlomi Dolev, Nova Fandina e Joseph Rosen. “Holographic parallel processor for calculating Kronecker product”. Em: *Natural Computing* 14.3 (2015), pp. 433–436.

[3] Leonardo Brenner, Paulo Fernandes e Afonso Sales. “The need for and the advantages of generalized tensor algebra for Kronecker structured representations”. Em: *International Journal of Simulation: Systems, Science & Technology* 6.3-4 (2005), pp. 52–60.

[4] Marco Enrıquez e Oscar Rosas-Ortiz. “Some applications of the Kronecker product in Hubbard representation”. Em: *Journal of Physics: Conference Series* 538 (out. de 2014), p. 012007. DOI: <10.1088/1742-6596/538/1/012007>. URL: <https://doi.org/10.1088/1742-6596/538/1/012007>.

[5] Len J. Sciacca e Robin J. Evans. “Multidimensional Inverse Problems in Ultrasonic Imaging”. Em: *Multidimensional Systems: Signal Processing and Modeling Techniques*. Ed. por C.T. Leondes. Vol. 69. Control and Dynamic Systems. Academic Press, 1995, pp. 1–48. DOI: <https://doi.org/10.1016/S0090-5267(05)80037-4>. URL: <https://www.sciencedirect.com/science/article/pii/S0090526705800374>.

[6] Huamin Zhang e Feng Ding. “On the Kronecker Products and Their Applications”. Em: *Journal of Applied Mathematics* 2013 (jun. de 2013). DOI: <10.1155/2013/296185>.

[7] Brigitte Plateau. “On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms”. Em: *SIGMETRICS Perform. Eval. Rev.* 13.2 (ago. de 1985), pp. 147–154. ISSN: 0163-5999. DOI: <10.1145/317786.317819>. URL: <https://doi.org/10.1145/317786.317819>.

[8] William J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994. ISBN: 9780691036991. URL: <http://www.jstor.org/stable/j.ctv182jsw5>.

[9] Bernard Philippe, Youcef Saad e William J. Stewart. “Numerical Methods in Markov Chain Modelling”. Em: *Operations Research* 40 (1996), pp. 1156–1179.

[10] Claude Tadonki e Bernard Philippe. “Parallel Multiplication of a Vector by a Kronecker Product of Matrices (part II)”. Em: *Parallel and Distributed Computing Practices* 3.3 (2000).

[11] Anne Benoit, Brigitte Plateau e William J Stewart. “Memory-efficient Kronecker algorithms with applications to the modelling of parallel systems”. Em: *Future Generation Computer Systems* 22.7 (2006), pp. 838–847.

[12] Tugrul Dayar e M Can Orhan. “On vector-Kronecker product multiplication with rectangular factors”. Em: *SIAM Journal on Scientific Computing* 37.5 (2015), S526–S543.

[13] Roger A. Horn e Charles R. Johnson. *Topics in Matrix Analysis*. Cambridge University Press, 1991. DOI: <10.1017/CBO9780511840371>.

[14] I.S. Duff, A.M. Erisman e J.K. Reid. *Direct Methods for Sparse Matrices.* Monographs on numerical analysis. Clarendon Press, 1989. ISBN: 9780198534211. URL: <https://books.google.com.br/books?id=rlX7tbJstpIC>.

[15] Harold V. Jemderson, Friedrich Pukelsheim e Shayle R. Searle. "On the history of the kronecker product". Em: *Linear and Multilinear Algebra* 14.2 (1983), pp. 113–120.

[16] Leonardo Brenner, Paulo Fernandes e Afonso Sales. "Why you should care about Generalized Tensor Algebra". Em: 2003.

[17] Charles F Van Loan. "The ubiquitous Kronecker product". Em: *Journal of computational and applied mathematics* 123.1-2 (2000), pp. 85–100.

[18] Peter Buchholz et al. "Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models". Em: *INFORMS Journal on Computing* 12.3 (2000), pp. 203–222.

[19] Claude Tadonki. "Large scale kronecker product on supercomputers". Em: *2011 Second Workshop on Architecture and Multi-Core Applications (wamca 2011)*. IEEE. 2011, pp. 1–4.

[20] Philip J Mucci et al. "PAPI: A portable interface to hardware performance counters". Em: *Proceedings of the department of defense HPCMP users group conference*. Vol. 710. 1999.

[21] Krishnaswamy Viswanathan. *Disclosure of Hardware Prefetcher Control on Some Intel Processors.* <https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html>. 2014.