Performance Analysis and Optimization of the Vector-Kronecker Product Multiplication

Alexandre Azevedo*, Cristiana Bentes†, Maria Clicia Castro‡, Claude Tadonki§
*Computational Sciences Program, State University of Rio de Janeiro, alexaazevedo@gmail.com
†Department of Systems Engineering, State University of Rio de Janeiro, cris@eng.uerj.br
‡Department of Informatics and Computer Science, State University of Rio de Janeiro, clicia@ime.uerj.br
§CRI (Centre de Recherche en Informatique), MINES ParisTech, claude.tadonki@mines-paristech.fr

***Abstract*—The Kronecker product, also called tensor product, is a fundamental matrix algebra operation, used to model complex systems using structured descriptions. This operation needs to be computed efficiently, since it is a critical kernel for iterative algorithms. In this work, we focus on the vector-kronecker product operation, where we present an in-depth performance analysis of a sequential and a parallel algorithm previously proposed. Based on this analysis, we proposed three optimizations: changing the memory access pattern, reducing load imbalance and manually vectorizing some portions of the code with Intel SSE4.2 intrinsics. The obtained results show better cache usage and load balance, thus improving the performance, especially for larger matrices.**

I. Introduction

In large-scale scientific applications, the matrix-vector product is usually a key operation. For many applications, however, when the matrix is very large, it is better to avoid forming it explicitly before performing the multiplication. This not only saves memory space but can also reduce redundant computations.

In this work, we focus on tensor algebra and the particular component of this field that is solving a vector-kronecker product multiplication. There are a number of applications that rely on this operation, as the multidimensional modeling of Stochastic Automata Networks (SAN) [1], [2], [3], Fast Fourier Transform (FFT), Fast Poisson Solver (FPS) [4], [5], Quantum Computation (QC) [6], and Lattice Quantum Chromodynamics [7].

The kronecker product matrix is defined as a block matrix formed with a special multiplication between two matrices [8]. The problem is that given N square matrices $A^{(i)}$ of order n_i and a vector $x \in \mathbb{R}^{1\times L}$ where $L = \prod_{n=1}^{N} n_i$, the complexity of building this matrix is $(\prod_{i=1}^{N} n_i^2)$. This complexity can make the matrix construction and the space for storage prohibitive.

Therefore, in this work, we take a different approach to computing the vector-kronecker product multiplication. Instead of building the full matrix, we exploit the *normal factor* property in order to proceed with each matrix individually, as proposed in [9]. We propose here a thorough study of the sequential and parallel algorithm to solve the vector-kronecker product multiplication proposed in [9], focusing on the impact of the size and quantity of matrices, the cache memory behavior, and the load balance of the parallel algorithm. The performance analysis has arisen some optimizations to the original algorithm. We propose here an improvement in data access, a reduction of load imbalance in the parallel code, and vectorization of the most important loop.

The proposed optimizations intend to increase the efficiency of the algorithms. The load imbalance and vectorization optimizations provided the highest impact with all experimental inputs. The data access optimization provided an impact on inputs using large matrices.

The remainder of this paper is organized as follows. Section II presents the previous work on vector-kronecker product multiplication. Section III presents the definition of the kronecker product matrix, its properties, and an overview of the original vector-kronecker product multiplication algorithm. Section IV analysis the cache memory behavior and load imbalance of the original algorithm. In Section V, based on our analyses, we propose some optimizations and show the consequent performance improvements. Finally, Section VI shows our conclusions and directions for future work.

II. Related Work

The vector-kronecker product multiplication has been widely used and studied in several different applications within different areas. The work by Van Loan [10] offers a variety of models and well-known applications related to the kronecker product in different fields of research. This work confirms the importance of kronecker product. The work by Brenner *et al.* [11] focuses on the 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 work [8] is primarily based on optimizing the execution of the shuffle algorithm in order to improve data locality. The optimization also reduces the number of FLOPS, this is accomplished by focusing the computation on the nonzero values of the matrices and thus trying to avoid FLOPS that use zero rows and columns.

An important application of the tensor algebra is in modeling Continuous Time Markov Chains (CTMC). In CTMC, the kronecker product takes the role of the generator matrix of the Markov chain, holding all information regarding the system. Buchholz *et al.* [12] work presents a number of different algorithms for solving vector-kronecker multiplication. The algorithms are focused on reducing the number of operations performed and the memory usage by exploiting the sparsity of the matrices in the kronecker product. The work by Benoit *et al.* [3] proposes a new algorithm that tries to reduce the memory cost of operating a large kronecker product. The algorithm focuses on refining the size of the main vector by removing sections full of zeros. The latest work by Buchholz *et al.* [13] 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 and also increase its time efficiency.

For larger models, the processing time of the vector-kronecker product 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 [14] work presents a fully working system for calculating the kronecker product using an optical apparatus, where a set of lasers, filters, and sensors replace the common silicon processor. Tadonki and Philippe work [9] shows the fundamental mathematical reasoning and groundwork on which the majority of our work is based on. The main difficulty when 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 local inner products of much smaller portions rather than the massive result vector. The most important aspect of this work is the parallel algorithms, with two different approaches. The first one uses a redundant computation approach and avoids communication, 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's latest work [15] proposes an algorithm to allow the vector-kronecker product multiplication to be calculated with large scale hybrid supercomputers, avoiding redundant work to be performed. The focus of this work is the optimization of communication among processors. The work presents a heuristic algorithm to construct an optimal topology for processor communication, which increases the scalability of the computation of his previous work. The implementation uses a hybrid parallelization with the shared memory model on the computing nodes, allowing further improvement in the scalability.

Our work is based on [9], [15] and extends them. Since the shared memory algorithm plays an important role in reducing the execution time, our goal here is to properly understand the cache effect and employ different optimization techniques to further reduce the computational cost.

III. Vector-Kronecker Product Multiplication

Definition 1 (Kronecker Product)**.** The Kronecker product of two matrices $A \in \mathbb{R}^{n_a \times m_a}$ and $B \in \mathbb{R}^{n_b \times m_b}$, denoted by $A \otimes B$, defined as the following matrix $\in \mathbb{R}^{n_a n_b \times m_a m_b}$:

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

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

$$M \otimes (N \otimes P) = (M \otimes N) \otimes P \tag{1}$$

Factorization (normal factors)

$$A \otimes B = (A \otimes I_{na}) \times (I_{nb} \otimes B) \tag{2}$$

By using Equation (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 $\prod_{n=1}^{N} n_i \times \prod_{n=1}^{N} m_i$ [9].

As can be seen from previous formula, the kronecker product can be a considerable challenge to properly

handle because of 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 $n_i, i = 1, 2, 3, ...N$, followed by the use of Equation (2), which yield Equation (3).

$$\otimes_{i=1}^{N} A^{i} = \prod_{s=1}^{N} (I_{n1} \otimes ... \otimes I_{ns-1} \otimes A^{s} \otimes I_{ns+1} \otimes \cdot \otimes I_{nN}) \tag{3}$$

It is also important to mention that the ordinary matrix operation is commutative with normal factors.

Definition 2 (Vector-Kronecker Product Multiplication)**.** Consider N square matrices $A^{(i)}$ of order n_i and a vector $x \in \mathbb{R}^{1 \times L}$ where $L = \prod_{n=1}^{N} n_i$. The Vector-Kronecker Product Multiplication is defined as:

$$z = x(\otimes_{n=1}^{N} A^{(i)}) \tag{4}$$

The construction of the kronecker product matrix $\otimes_{n=1}^{N} A^{(i)}$ to perform the multiplication (4) would take a large amount of space-memory, where the order of this matrix is $(\prod_{n=1}^{N} n_i)^2$. Besides, the naive approach in building it would yield the same 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 yields

$$(\prod_{i=1}^{N} n_i)^2 \tag{5}$$

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

$$(\sum_{i=1}^{N} n_i) \times (\prod_{i=1}^{N} n_i). \tag{6}$$

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

$$(Nn) \times (n^N) = Nn^{N+1} \tag{7}$$

The number of memory accesses is proportional to the floating-point operations (FLOPS) complexity. However, the sustained performance 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. Table I shows some complexity values for different scenarios.

A. The Algorithm

In [9], Tadonki and Philippe proposed a sequential and parallel implementation of the vector-kronecker product multiplication algorithm. In the sequential algorithm, the main kronecker 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.

N \ n_i	8	12	16	18	20
10	8.6×10^{10}	7.4×10^{12}	1.7×10^{14}	6.4×10^{14}	2.1×10^{15}
12	6.6×10^{12}	1.3×10^{15}	5.4×10^{16}	2.5×10^{17}	9.8×10^{17}
14	4.9×10^{14}	2.1×10^{17}	1.6×10^{19}	9.4×10^{19}	4.6×10^{20}
16	3.6×10^{16}	3.5×10^{19}	4.7×10^{21}	3.5×10^{22}	2.1×10^{23}
18	2.6×10^{18}	5.7×10^{21}	1.4×10^{24}	1.3×10^{25}	9.4×10^{25}
20	1.8×10^{20}	9.2×10^{23}	3.9×10^{26}	4.6×10^{27}	4.2×10^{28}

Table I
COMPLEXITY OF DIFFERENT SCENARIOS

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 is being used is controlled by the outermost loop at line 4. In order to perform all computation with each 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 of size n_s, at 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 at lines 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

1. V $\leftarrow$ X
2. L $\leftarrow \prod_{p=1}^{N} n_p$
3. r $\leftarrow 1$
4. **for** $s \leftarrow N$ **to** *1* **do**
5.   l $\leftarrow L/n_s$
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.   **for** $j \leftarrow 1$ **to** n_s **do**
11.   **for** $t \leftarrow 1$ **to** n_s **do**
12.   scal $\leftarrow$ scal + $A^{(s)}(t, j) * U[t]$
13. $V[((k-1) * n_s + j - 1) * r + i] \leftarrow$ scal
14.   r $\leftarrow r * n_s$
15. Z $\leftarrow$ V

The shared memory parallel implementation of this algorithm, also proposed in [9], 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 communication between processors.

IV. Analyzing the Cache Behavior

Before optimizing the Vector-Kronecker Product Multiplication algorithm, we performed an in-depth study on its cache behavior. The experiments were performed on an Intel Core i7 930 with 8Gb of RAM memory running Ubuntu 16.04 LTS Linux distribution. They consist of operating a vector-kronecker multiplication with two square matrices of different order N. The implementation of Algorithm 1 was done in C, using gcc version 5.4 with the optimization flag -O3. We analyzed only the L1 behavior because the number of L2 accesses for this algorithm is negligible when compared to the L1 accesses (L2 accesses represent only 0.5% of the total memory accesses). The L1 data cache accesses and hit rate were collected using the Performance Application Programming Interface (PAPI) library [16].

Figure 1 shows the values of the L1 hit ratio for different matrix sizes, $\{2, 4, 6, ...1000\}$. 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 innermost loop of the algorithm and the hardware prefetching mechanisms.

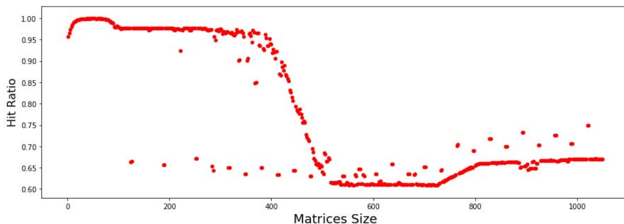

Figure 1. L1 Hit ratio for increasing matrix sizes

The L1 data cache of the Intel Core i7 930 is an 8-way set associative with 32KB size, which can store up to 8192 float elements. The cache line size is 64 bytes thus can store 16 float elements. Intel processors have two types of hardware prefetchers for the L1 cache [17]. The L1 hardware prefetchers are called Data Cache Unit (DCU). The first prefetcher, 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 an appropriate stride.

The two innermost loops of the vector-kronecker product 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 an inner product between V and A_j, where A_j is a column of matrix A. The great 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 main variables inside this loop, which are U, A, and $scal$. The values of U and $scal$ always fit in L1 for any $n \leq 4000$ (for $n = 4000$, U uses 15.6KB of the L1 cache). For values of $n \leq 90$, the whole matrix A also fits in the L1 cache (for $n = 90$, A uses 31.6KB and U uses 360 bytes). 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 vector-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 contiguous 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, that occupy around 42KB, and this would exceed the capacity of 32KB of the L1 cache. So, the access of element $A[i,j+1]$ will generate a miss, since it is not stored in L1 anymore. This explains 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. Inside the innermost loop, there is one access to U, one access to $scal$, and one access to $A[i,j]$. So, the

accesses to A represent $\frac{1}{3}$ of the L1 accesses inside the loop. Since half of the accesses will generate misses, the algorithm will generate around $\frac{1}{6}$ of faults, which gives a hit rate of around 80%, as shown in the graph.

In order to evaluate how the execution time of the vector-kronecker product multiplication 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 approximately constant the number of memory accesses, which can be done by selecting a collection of input data that has roughly the same complexity, as computed from Equation 7.

Table II shows for each matrix size and number of matrices, the size of the problem (complexity), the L1 cache hit ratio, the execution time (in seconds), and the number of cache accesses. This set of experiments had the objective of showing that for a similarly complex experiment, matrices with size larger than 330 have lower hit ratio, thus resulting in a lower execution time when compared to several matrices of smaller sizes. For this set of experiments, the matrices used for the same execution are of the same size. Considering the 3 accesses in the innermost loop (one for U, one for $scal$, and one for $A[i,j]$), 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 the matrix.

n_i	N	Complexity $\times(10^{10})$	Hit Ratio	Time(s)	Accesses $\times(10^{10})$
4	14	1.5	0.995	25.031	5.2
7	10	2.0	0.997	26.661	6.5
11	8	1.9	0.998	23.418	6.1
23	6	2.1	0.999	23.707	6.4
290	3	2.0	0.949	28.200	6.4
2000	2	1.6	0.642	131.148	6.4

Table II
CACHE BEHAVIOR AND EXECUTION TIME FOR DIFFERENT MATRIX SIZES AND NUMBER OF MATRICES.

A. Parallel Implementation

Table III shows the L1 hit ratio, time, number of accesses and the speedup for the shared memory parallel implementation using 4 threads. The hit ratios in Table III show similar behavior as the sequential version, considering that the threads are independent of each other. We can observe, however, an important drop in the speedup for the matrices sizes of 290×290 and $2000\times$ 2000. This drop is caused by a load imbalance. According to Algorithm 1, as the execution progresses, the l index will become smaller and r will become bigger. For the very last matrix, l should be 1 and the execution will be essentially sequential.

n_i	N	L1 hit ratio	time(s)	# accesses $\times(10^{10})$	Speedup
4	14	0.991	7.693	5.6	3.25
7	10	0.994	9.093	6.8	2.93
11	8	0.996	8.149	6.2	2.87
23	6	0.998	8.793	6.4	2.71
290	3	0.951	15.478	6.4	1.82
2000	2	0.568	93.319	6.4	1.4

Table III
PARALLEL IMPLEMENTATION RESULTS

V. PROPOSED OPTIMIZATIONS AND PERFORMANCE IMPROVEMENTS

Based on the previous performance analysis of the vector-kronecker product multiplication, we performed some optimizations to the code to increase its performance. This section presents the proposed optimizations and the performance improvements obtained with each of them. The experiments were undertaken on the same experimental platform like the one described in the previous section.

A. Optimizing the data access

As explained in Section IV, the innermost loop of the vector-kronecker product multiplication accesses the matrix A^i by columns. The problem of accessing A^i by columns is that for $N > 330$, there is a considerable drop in L1 hit ratio. Therefore, the first recognizable optimization made to the code is direct: transpose A and perform the inner loop of the vector-kronecker product multiplication by row.

Figure 2 shows the L1 hit ratio of the optimized code as the matrices sizes increase. For all the experiments, the transpose operation is precomputed. We can observe in this graph an almost constant high L1 hit ratio, regardless of the matrices sizes. The spike at the beginning of the graph occurs because for $N < 90$, the whole matrices and vectors fit in L1 (as explained in Section IV). The comparison of this graph with the graph of Figure 1 shows that transposing A before computing the vector-kronecker product multiplication produces better L1 performance.

Table IV 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 vector-kronecker product multiplication, for all matrices sizes. This reduction is more prominent for bigger matrices.

Input Size		Access per row			Access per column		
n_i	N	hit ratio	time(s)	# accesses $\times(10^{10})$	hit ratio	time(s)	# accesses $\times(10^{10})$
4	14	0.994	23.33	4.0	0.995	25.03	5.2
7	10	0.996	25.38	4.8	0.997	26.66	6.5
11	8	0.999	21.75	4.4	0.998	23.41	6.1
23	6	0.999	22.19	4.4	0.999	23.70	6.4
290	3	0.982	21.51	4.3	0.949	28.20	6.4
2000	2	0.857	16.91	3.2	0.642	131.14	6.4

Table IV
L1 HIT RATIO, NUMBER OF MEMORY ACCESSES AND EXECUTION TIMES FOR THE ORIGINAL AND OPTIMIZED CODE

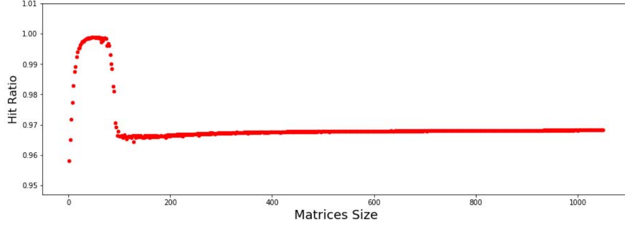

Figure 2. L1 hit ratio for increasing matrices sizes

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 $n_i = 16$. Figure 3 shows the assembly code of the innermost loop of the vector-kronecker product when $n_i = 16$ for the implementation by column. For $n_i = 16$, the compiler unrolls the innermost loop, and we can observe 16 repetitions of the operation in (8).

$$scal = scal + A[i][j] \times U[i] \quad (8)$$

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 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.

```
.L61:
    movss   (%r14,%rax), %xmm4      movaps  %xmm0, %xmm1           addss   %xmm1, %xmm0
    movq    8(%rsp), %rcx           movss   (%rdi,%rax), %xmm0     movss   (%rcx,%rax), %xmm1
    mulss   (%rdx), %xmm4           mulss   36(%rdx), %xmm0        mulss   52(%rdx), %xmm1
    movss   0(%r13,%rax), %xmm3     addss   %xmm2, %xmm1           movq    32(%rsp), %rcx
    mulss   4(%rdx), %xmm3          movss   (%r9,%rax), %xmm2      addss   %xmm0, %xmm4
    movss   (%r12,%rax), %xmm2      mulss   28(%rdx), %xmm2        movss   (%rcx,%rax), %xmm0
    mulss   8(%rdx), %xmm2          addss   %xmm1, %xmm4           mulss   56(%rdx), %xmm0
    movss   0(%rbp,%rax), %xmm1     movss   (%r8,%rax), %xmm1      addss   %xmm4, %xmm3
    mulss   12(%rdx), %xmm1         mulss   32(%rdx), %xmm1        addss   %xmm3, %xmm2
    movss   (%rbx,%rax), %xmm0      addss   %xmm4, %xmm3           addss   %xmm2, %xmm1
    mulss   16(%rdx), %xmm0         movss   (%rsi,%rax), %xmm4     addss   %xmm1, %xmm0
    addss   %xmm5, %xmm4            mulss   40(%rdx), %xmm4        movaps  %xmm0, %xmm1
    addss   %xmm4, %xmm3            addss   %xmm3, %xmm2           movss   (%r15,%rax), %xmm0
    movss   (%r11,%rax), %xmm4      movss   (%rcx,%rax), %xmm3     mulss   60(%rdx), %xmm0
    mulss   20(%rdx), %xmm4         mulss   44(%rdx), %xmm3        addss   %xmm1, %xmm0
    addss   %xmm3, %xmm2            movq    16(%rsp), %rcx         movss   %xmm0, (%rdx,%rax)
    movss   (%r10,%rax), %xmm3      addss   %xmm2, %xmm1           addq    $4, %rax
    mulss   24(%rdx), %xmm3         movss   (%rcx,%rax), %xmm2     cmpq    $64, %rax
    addss   %xmm2, %xmm1            mulss   48(%rdx), %xmm2        jne .L61
    movaps  %xmm1, %xmm2            movq    24(%rsp), %rcx
```

Figure 3. Assembly code for the innermost loops when N=16, accessing the matrices by column

On another hand, the implementation by row does not require the load of the matrix addresses to registers. Since the matrix is accessed by row the compiler just needs to increase the offset 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 $n_i = 32$, 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 4 and 5 show the assembly code for the innermost loops of the vector-kronecker product when $n_i = 32$ for column and row implementations, respectively. These figures show that for $n_i = 32$, 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 (8), with two jne instructions defining the loop limits.

Despite using fewer registers, the drawback of 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 shown 203693 memory accesses with the column implementation and 140222 memory accesses with the row implementation.

```
.L61:
    movq    (%rdi,%rax,2), %rdx
    movss   (%rdx,%rcx), %xmm0
    mulss   (%rsi,%rax), %xmm0
    addq    $4, %rax
    cmpq    $128, %rax
    addss   %xmm0, %xmm1
    jne .L61
```

Figure 4. Assembly code for 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 5. Assembly code for for N=32, accessing the matrices by row

B. Load Imbalance

As explained in Section IV-A, the parallel algorithm shown a load imbalance when the l index becomes small and r becomes big. We propose here a solution to this thread imbalance problem. Algorithm 2 shows our optimization 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 1. However, when $l < p$ the parallel region will change to an inner loop allowing all processors to be used until the end of the execution.

Algorithm 2: Optimized parallel Vector-matrix multiplication

```
1  V ← X
2  L ← ∏_{p=1}^{N} n_p
3  r ← 1
4  p ← num_threads
5  for s ← N to 1 do
6      l ← L/n_s
7      if (l ¿ p) then
8          #pragma omp parallel for
9          for k ← 1 to l do
10             for i ← 1 to r do
11                 for t ← 1 to n_s do
12                     U[t] ← V[((k−1)∗n_s+t−1)∗r+i]
13                 for j ← 1 to n_s do
14                     for t ← 1 to n_s do
15                         scal ← scal + A^(s)(t,j) ∗ U[t]
16                     V[((k−1)∗n_s+j−1)∗r+i] ← scal
17     else
18         for k ← 1 to l do
19             #pragma omp parallel for
20             for i ← 1 to r do
21                 for t ← 1 to n_s do
22                     U[t] ← V[((k−1)∗n_s+t−1)∗r+i]
23                 for j ← 1 to n_s do
24                     for t ← 1 to n_s do
25                         scal ← scal + A^(s)(t,j) ∗ U[t]
26                     V[((k−1)∗n_s+j−1)∗r+i] ← scal
27     r ← r ∗ n_s
28 Z ← V
```

Table V shows the number of memory accesses for both the unbalanced version and the optimized version of the code for each thread. The optimization has greatly reduced the imbalance in terms of memory access and increased the speedup, especially for the larger matrices. The execution times and speedups compared to the sequential version are shown in Table VI.

C. Vectorization

Since the gcc compiler with -O3 optimization flag was not able to vectorize the innermost loop of the vector-kronecker product, we implemented the operation of (8) using Intel SSE4.2 intrinsics. The innermost loop of

Input Size			Original	Optimized
n_i	N	Thread ID	# access $\times(10^{10})$	# access $\times(10^{10})$
4	14	0	1.7	1.4
4	14	1	1.3	1.4
4	14	2	1.3	1.4
4	14	3	1.3	1.4
2000	2	0	4.0	1.6
2000	2	1	0.8	1.6
2000	2	2	0.8	1.6
2000	2	3	0.8	1.6

Table V
MEMORY ACCESSES OF THE ORIGINAL AND OPTIMIZED VERSION.

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

Table VI
EXECUTION TIME AND SPEEDUP OF THE UNBALANCED VERSION AND THE OPTIMIZED VERSION OF THE CODE.

the vector-kronecker product is essentially a dot product between vector U and one column of the matrix.

Table VII 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. Both the sequential and the vectorized implementation uses the optimized version where the matrix is accessed by rows.

Input Size		Sequential	Vectorized	
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 VII
EXPERIMENTAL RESULTS OF VECTORIZATION

VI. CONCLUSIONS

Multiplying a vector by a kronecker product of matrices has been widely applied to model complex systems. In this paper, we analyzed the cache memory behavior of a sequential and a parallel version of the vector-kronecker product based on [9] and [15] and extended them. We proposed three different optimizations to the algorithm: changing the data access pattern, reducing the load imbalance, and manually vectorizing the most important loop with Intel SSE intrinsics.

Our results show performance improvements for transposing the matrices, storing them by row, and performing the inner loop of the vector-kronecker product by row, instead of by column. The improvement in the load imbalance yield performance gains for large matrices and the vectorization of the innermost loop provided speedups for all matrices sizes.

For future work, as most complex systems are represented by sparse matrices, we intend to focus on algorithms that avoid the multiplications of all null elements. Also, we intend to observe the performance gains in another computational environment with a greater number of processors.

REFERENCES

[1] P. Fernandes, B. Plateau, W. J. Stewart, Efficient descriptor-vector multiplications in stochastic automata networks, Journal of the ACM (JACM) 45 (3) (1998) 381–414.

[2] A. Benoit, P. Fernandes, B. Plateau, W. J. Stewart, On the benefits of using functional transitions and kronecker algebra, Performance Evaluation 58 (4) (2004) 367–390.

[3] A. Benoit, B. Plateau, W. J. Stewart, Memory-efficient kronecker algorithms with applications to the modelling of parallel systems, Future Generation Computer Systems 22 (7) (2006) 838–847.

[4] C. Tong, P. N. Swarztrauber, Ordered fast fourier transforms on a massively parallel hypercube multiprocessor, Journal of Parallel and Distributed Computing 12 (1) (1991) 50–59.

[5] C. Van Loan, Computational frameworks for the fast Fourier transform, SIAM, 1992.

[6] P. W. Shor, Quantum computing, Documenta Mathematica 1 (1000) (1998) 1.

[7] C. Tadonki, G. Grodidier, O. Pene, An efficient cell library for lattice quantum chromodynamics, ACM SIGARCH Computer Architecture News 38 (4) (2011) 60–65.

[8] T. Dayar, M. C. Orhan, On vector-kronecker product multiplication with rectangular factors, SIAM Journal on Scientific Computing 37 (5) (2015) S526–S543.

[9] C. Tadonki, B. Philippe, Parallel multiplication of a vector by a kronecker product of matrices (part ii), Parallel and Distributed Computing Practices 3 (3) (2000).

[10] C. F. Van Loan, The ubiquitous kronecker product, Journal of computational and applied mathematics 123 (1-2) (2000) 85–100.

[11] L. Brenner, P. Fernandes, A. Sales, The need for and the advantages of generalized tensor algebra for kronecker structured representations, International Journal of Simulation: Systems, Science & Technology 6 (3-4) (2005) 52–60.

[12] P. Buchholz, G. Ciardo, S. Donatelli, P. Kemper, Complexity of memory-efficient kronecker operations with applications to the solution of markov models, INFORMS Journal on Computing 12 (3) (2000) 203–222.

[13] P. Buchholz, T. Dayar, J. Kriege, M. C. Orhan, On compact solution vectors in kronecker-based markovian analysis, Performance Evaluation 115 (2017) 132–149.

[14] S. Dolev, N. Fandina, J. Rosen, Holographic parallel processor for calculating kronecker product, Natural Computing 14 (3) (2015) 433–436.

[15] C. Tadonki, Large scale kronecker product on supercomputers, in: 2011 Second Workshop on Architecture and Multi-Core Applications (wamca 2011), IEEE, 2011, pp. 1–4.

[16] P. J. Mucci, S. Browne, C. Deane, G. Ho, Papi: A portable interface to hardware performance counters, in: Proceedings of the department of defense HPCMP users group conference, Vol. 710, 1999.

[17] K. 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).