Faculty of Sciences and Technology

Department of Informatics Engineering

A Study on the Energy Efficiency of Matrix Transposition Algorithms

Gonçalo Alexandre Pinto Lopes

Dissertation in the context of the Masters in Informatics Engineering, Specialization in Software Engineering advised by Prof. João Paulo Fernandes and Prof. Luís Paquete and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

September 2019

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

Abstract

Energy consumption is becoming a serious concern in the context of software development. Recent works have shown that the energy consumption of an algorithm not only depends on its running time but also on its number of memory accesses. This suggests that the total energy consumed by an algorithm can be modelled as a linear combination of the energy consumed by the CPU instructions and memory accesses. In this work, we empirically analyse several algorithms for matrix transposition operation with different patterns of low-level cache access, and compare them in terms of energy consumption and running time with respect to CPU instructions and memory accesses for different matrix sizes. Moreover, we analyse the effect of parallelization on energy consumption and running time performance of different memory access patterns. Our results suggest that different memory access patterns and the number of activated cores in the parallel version have a strong influence on the energy consumption and on the cache performance of these algorithms.

Keywords

Energy consumption, Running time, CPU instructions, Memory accesses, Matrix Transposition, Memory access patterns, Cache performance, Parallelization

This page is intentionally left blank.

Resumo

O consumo de energia está a tornar-se uma preocupação séria no contexto de desenvolvimento de software. Estudos recentes mostraram que o consumo de energia de um algoritmo não depende apenas do tempo de execução, mas também do número de acessos à memória. Isso sugere que a energia total consumida por um algoritmo pode ser modelada como uma combinação linear da energia consumida pelas instruções do CPU e acessos a memória. Neste trabalho, analisamos empiricamente vários algoritmos para a operação de transposição de matrizes com diferentes padrões de acesso a memória, comparando-os em termos de consumo de energia e tempo de execução relativamente às instruções do CPU e acessos à memória para diferentes tamanhos de matrizes. Além disso, também analisamos o efeito da paralelização no consumo de energia e no desempenho do tempo de execução dos diferentes padrões de acesso a memória. Os resultados obtidos sugerem que diferentes padrões de acesso a memória e o número de cores ativados na versão paralela exercem uma forte influência no consumo de energia e no desempenho da cache desses algoritmos.

Palavras-Chave

Consumo de energia, Tempo de execução, Instruções do CPU, Acessos a memória, Transposição de Matrizes, Padrões de acesso a memória, Desempenho da cache, Paralelização

This page is intentionally left blank.

Acknowledgements

I would like to thank my family, particularly my parents and sister, for supporting, encouraging me and helping me get to where I am today. To my friends, thank you for the comfortable moments, which are part of a healthy lifestyle and coexist with the hard work, and for the late nights full of stories. Moreover, a special thanks to my reviewers, for all the fixes and improvements to make my thesis better, and to the persons closer to me that during this period provide me all the encouragement and motivation to stay focus and reach my goals.

A special thanks to the Department of Informatics Engineering of the University of Coimbra for, without doubt, one of the best five years of my life, and most importantly, providing me with an excellent quality education. My thanks also go to the ECOS lab, for the company and work environment, and for allowing me to access and use their workspace. Finally, a special thanks also to Swapnoneel Roy for providing the source code of one of the analysed algorithms.

Last but not least, an even bigger special thanks to my both advisers, Prof. João Paulo Fernandes and Prof. Luís Paquete, for guiding my work, for all the dispensed time, incentive and effort to make my thesis better, and that despite all the setbacks made this possible.

This page is intentionally left blank.

Contents

Acronyms

ALU Arithmetic Logic Unit.

BFS Breadth-first Search.

CPU Central Processing Unit.

CU Control Unit.

DDR Double Data Rate.

DFS Depth-first Search.

DRAM Dynamic Random Access Memory.

EM External Memory.

FIFO First-In, First-Out.

GPU Graphics Processing Unit.

HDD Hard Disk Drive.

I/O Input-Output.

LLC Last Level Cache.

LRU Least Recently Used.

MRU Most Recently Used.

MSR Model-Specific Register.

OpenMP Open Multi-Processing.

OS Operating System.

PEM Parallel External Memory.

PRAM Parallel Random-Access Machine.

RAM Random-Access Machine.

RAM Random-Access Memory.

RAPL Running Average Power Limit.

SDRAM Synchronous Dynamic Random-access Memory.

SIMD Single Instruction Multiple Data.

SRAM Static Random Access Memory.

TBB Intel Threading Building Blocks.

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

The increasing popularity of electronic devices and platforms leads to questions regarding their energy efficiency, which is relevant on several levels. It has impact on the utility of portable devices, e.g. battery life of mobile phones [1], on business costs, e.g. energy consumption of large data centers [2, 3], and also on social aspects, e.g. impact of the energy consumption of electronic devices on global warming [4]. As such, the topic of energy consumption is becoming a growing concern in the context of Information Technology. The software industry started thinking about energy when designing, constructing, testing and maintaining their software, as well as developing strategies to improve the energy usage of their applications [5]. According to a recent article [6], there is still little knowledge on how to reduce the energy consumption of software. According to the research performed, from 2005 to 2014, only 20 research articles included the words "power" and "energy" on their keywords. The authors also observed that this topic emerged only around 2012, as well as energy consumption becomes a more serious concern for software developers [7, 8, 9, 10]. However, due to the lack of knowledge and guidelines on how to write energy efficient software applications, these practices have not been consolidated.

In order to deal with energy consumption concerns, hardware manufacturers have been improving lower-level layers of the hardware to reduce energy consumption [11]. However, recent studies [12, 13] indicate that better results can be achieved by encouraging software developers to participate in the process [14], complementing the low-level improvements. For that reason, hardware manufacturers have been designing tools for software developers to understand energy consumption of programs. Some CPU manufacturers already provide some measurement tools that collect data from different system interfaces and calculate the energy consumption of their processors, such as PowerTOP [15], Intel Power Gadget [16] and RAPL [17, 18].

In order to improve energy efficiency, developers assume that improving the program's running time increases energy efficiency. However, it is common folklore that energy consumption depends only on the running time [19]. This assumption suggests that the energy consumed equals running time times power, that is, a faster program is also a more energy efficient program. Moreover, when an asymptotic performance is considered, constant factors, such as memory accesses, are ignored. However, there exist other factors that may play some role on the energy performance of a program in practice [8]. For instance, some works have been showing that different algorithm implementations [20, 21], different cache architectures and data management choices by the programmers [22, 23, 24], the choice of the programming language [25, 26] or the code practices and data structures used by the programmers [27, 28, 29, 30, 31, 32], may affect energy consumption of a program.

In the current thesis, we took into consideration the Energy Complexity model proposed by Roy et al. [33] concerning the energy consumption of an algorithm. The authors propose a computational complexity model for the consumption of energy of an algorithm, which models the total energy consumed as a function of both CPU instructions and memory accesses. Moreover, based on parallel memory models, such as Two-level Memory model [34], they suggest that the energy consumed by an algorithm is not just dependent of the time taken by non Input-Output (I/O) operations but also on the number of accesses to memory to read or write a row that contains the required memory word. Therefore, this model indicates that is possible to minimise energy consumption by improving cache performance. Several techniques exist that optimise cache performance using different patterns of memory accesses, such as Cache-Oblivious and Cache-Aware Algorithms [35]. Moreover, some empirical studies on cache efficiency have been performed [36, 37].

Another aspect that may affect energy consumption is parallelization. Over the years, hardware manufacturers have been improving lower-level layers of computer architectures, mainly increasing the number of physical components, particularly cores, multicore architectures [38]. These architectures enable the spread of instructions over different performing units, thread-level and task-level parallelism, increasing the performance of the applications. However, despite several studies to understand energy consumption of multicore architectures [39, 40], there is still little understanding of how the applications or algorithms explicitly designed or redesigned, to run in parallel and take advantage of the task-level parallelism, influence energy consumption [41]. Moreover, due to the memory hierarchy of these multicore architectures, several studies have been performed to understand their cache performance behaviour and optimisations [42, 43, 44, 45]. However, the energy consumption performance of different memory access patterns remains a quite unexplored topic.

In this work, our aim is to understand the relation between energy consumption of an algorithm in practice and its running-time and cache performance using different memory accesses patterns and different parallelization strategies. In particular, we describe a thorough experiment with several algorithms for the Matrix Transposition operation which arises in many real-life situations such as image processing [46, 47] and signal processing [48]. In Matrix Transposition, data elements occupy relatively close storage locations. However, for large matrices, the order in which the elements are swapped, that is, how cache writes and reads are managed, can cause strong effects on the running time and cache performance. We expect that different algorithmic strategies and implementation tricks may have a strong influence on energy performance.

The main research questions of this work are the following:

- **RQ1**: According to the Energy Complexity model proposed by Roy et al. [33], the total energy consumed by an algorithm can be modelled by the energy consumed by performing CPU instructions and memory accesses. According to this model, what is the energy consumption of the traditional Matrix Transposition algorithm?

  - We analysed the energy consumption and running time of both CPU instructions and memory accesses components on the traditional algorithm for Matrix Transposition, which we call the Naïve Algorithm. The results point that time and energy seem to be strongly correlated and show a dominance of the memory accesses over the CPU instructions on energy consumed and running time. Moreover, the large number of cache references and misses, and the dominance of the memory accesses, suggest that energy consumption can be reduced by reducing the number of instructions and/or the number of memory accesses.

- **RQ2**: According to the model proposed by Roy et al. [33] it is possible to improve energy consumption by taking advantage of how main memory is organised and accessed. They have proposed an algorithm based on parallel memory models, namely, the Blocked Transpose Algorithm. How does this algorithm behave in practice?

  - In comparison with the Naïve Algorithm, we observed an improvement in both energy and time consumption. However, despite the substantial lower time consumption, the Blocked Transpose Algorithm presented almost the same amount of energy consumed than that of the Naïve Algorithm. By analysing the energy consumed by different CPU components (DRAM and Core), we observed a lower energy consumption of DRAM and a larger energy consumption of Core. We also noticed that the latter may be related to the large number of bitwise operations that are required to acquire the data elements positions due to the memory organisation.

- **RQ3**: The results above suggest that cache usage and memory access patterns affect energy (and time). Therefore, can we improve the access to low-level caches, reducing the number of cache misses and inducing the reuse of data present in the cache levels?

  - We analysed variants of the algorithm for Matrix Transposition that should optimise the usage of the cache, which are based on Cache-Oblivious and Cache-Aware models. The results provided empirical evidence that both approaches decreased energy consumption. However, each model presented different energy and time consumption patterns for both CPU and memory accesses instructions. The Cache-Oblivious Algorithm presented better results than the Naïve Algorithm. Moreover, contrarily to what we have observed with the traditional algorithm, the time and energy consumed by the CPU instructions are higher than the time and energy consumed by the memory accesses. The Cache-aware algorithm, similar to what happens in the Naïve Algorithm, the amount of time and energy spent is lower in CPU instructions than in memory accesses.

- **RQ4**: One way of improving the running time of an algorithm for Matrix Transposition is to parallelize it. What is the effect of Multicore and Multithreading techniques in the energy consumption?

  - We implemented the parallelized versions of algorithms for matrix transposition and analysed their time and energy performance for different number of cores and/or threads. We observed that the increase on the number of threads on the same core, does not bring any energy performance improvement. Differently, we noticed a better energy performance and speed-up by increasing the number of cores. Moreover, the energy consumed by the memory accesses slightly decreased while the energy consumed by the CPU instructions decreased.

The document is structured in seven chapters. First, we start by introducing some necessary definitions, theoretical models and notations in Chapter 2. In Chapter 3 we review the literature concerning energy efficiency, cache efficiency and few related theoretical models. In Chapter 4 we review the algorithms for Matrix Transposition. Then, we present the methodology that is used in the remainder of the thesis in Chapter 5. In Chapter 6, we discuss the obtained results for each research question. Finally, we perform a general discussion and conclusions in Chapter 7. Additionally, in Chapter 7, we present some ideas for further work.

This page is intentionally left blank.

Chapter 2

Background

In this chapter, we introduce some background definitions necessary to understand the following chapters. We start by presenting in Section 2.1 two different types of analysis used to evaluate the performance of algorithms. Then, in Section 2.2 and 2.3 we present some definitions related to the computer architecture, cache and working principles of the computational models presented in Section 2.3. In Section 2.3 we discuss six computational models required to understand some theoretical concepts presented in Chapter 3 and working principles of the algorithms presented in Chapter 4. In Section 2.4 we present some cache optimisation techniques applied to the the experiments presented in Chapter 6. Finally, in Section 2.5 we review some fundamental concepts on multithreading and multiprocessing.

2.1 Analysis of algorithms

The analysis of algorithms mainly focuses on running time and memory usage under a specific computation model, usually RAM (see Section 2.3). The time complexity of an algorithm describes the amount of time needed according to the input size. Similarly, the space complexity of an algorithm describes the quantity of memory necessary according to the input size. Moreover, these analyses are focused on the worst scenario. In some cases, it is possible to describe the performance in terms of average case, assuming a certain distribution on the input data. Furthermore, time and space can be directly affected by several factors such as the hardware, operating system, processors and memory architecture. However, they are defined as constant factors and are not considered in the asymptotic analysis.

An experimental analysis consists of analysing the results of an algorithm based on some parameters, such as sample input size and some distribution of the input data. Therefore, time and space usage resources can also be evaluated. Moreover, the running time and space collected in the experimental analysis can be used to create regression models.

Regression is a method of modelling a target value based on independent predictors. Therefore it is possible to forecast and quantify the relationship between multiple variables. The regression techniques mostly differ in the number of independent variables analysed and the type of relationship between the independent or dependent variables. For example, linear regression considers a linear relationship between the predictor and response variables. Furthermore, besides the regression models, there are correlation models. These models measure the extent to which two variables tend to change together, describing both the

strength and direction of the relationship, also called a correlation coefficient. For example, the *Pearson product correlation* is used to evaluate the linear relationship between two continuous variables. However, this correlation does not measure the degree of association between two variables. Therefore, it is possible to use the Spearman rank-order correlation. *The Spearman rank-order correlation* evaluates the monotonic relationship between two continuous variables. The monotonic relationship verifies if the variables increase (or decrease) in the same direction, although, not always at the same rate.

2.2 Computer architecture

In this section, we present some definitions related to the computer architecture and cache, such as memory hierarchy, characteristics and behaviour of the CPU and CPU cache as well as multithreading and multicore architecture concepts.

Computer memory

Memory refers to any physical device capable of storing information for prompt use in a computer. Each type of memory can be either volatile or non-volatile. *Volatile memory*, such as Random-Access Memory (RAM), requires energy to maintain the stored contents periodically refreshed. Therefore, when the hardware device loses power, the memory drops its contents. *Non-volatile memory*, such as Hard Disk Drive (HDD), contrary to volatile memory, does not require energy to retain the stored information.

RAM is the hardware in a computing device where the Operating System (OS), software applications, data and machine code currently being used are stored. Since it allows data to be read and written much faster than other direct-access storage, it is the most common type of memory used for a Rapid-Access Memory. Therefore, computer CPUs are repeatedly accessing RAM. However, when RAM fills up, CPUs need to overlay the old data with new data, which slows down the computer operations.

There are two primary forms of RAM, the DRAM and Static Random Access Memory (SRAM). *DRAM* represents the typical computing RAM device and, as previously noted, its data is constantly refreshed with an electronic charge every few milliseconds, with a transistor acting as a capacitor holding the charge. *SRAM* such as DRAM, also needs constant power to retain the data. However, it does not need the systematic refresh of data content, since the transistor acts as a state switch, with one position serving and other not serving, requiring less energy to maintain. Moreover, SRAM is much more expensive to produce than DRAM. Therefore, there is a higher usage of the DRAM type.

The higher usage of DRAM and advantages over the SRAM led the manufactures to improve its structure. They created another memory-based type, Synchronous Dynamic Random-access Memory (SDRAM). The *SDRAM* is designed to synchronise itself with the CPU timing, enabling the memory to know the exact clock cycle when the request will be fulfilled. Therefore, the CPU does not need to wait between memory accesses. However, SDRAM needs to wait for the completion of the previous command to be able to perform another I/O operation. Over the years, further, precise and faster generations of SDRAM were produced such as Double Data Rate (DDR), DDR2, DDR3 and DDR4.

Computer data storage

The data storage of a computer, also known as memory, is a technology consisting of components that are used to retain digital data in a computer. Generally, computers use a memory hierarchy (see Figure 2.1), which places fast and small memory options close to the CPU and slower but larger options further away. Therefore, the lower the memory is in the hierarchy, the further away is from to the CPU, which leads to a lesser bandwidth and higher access latency.

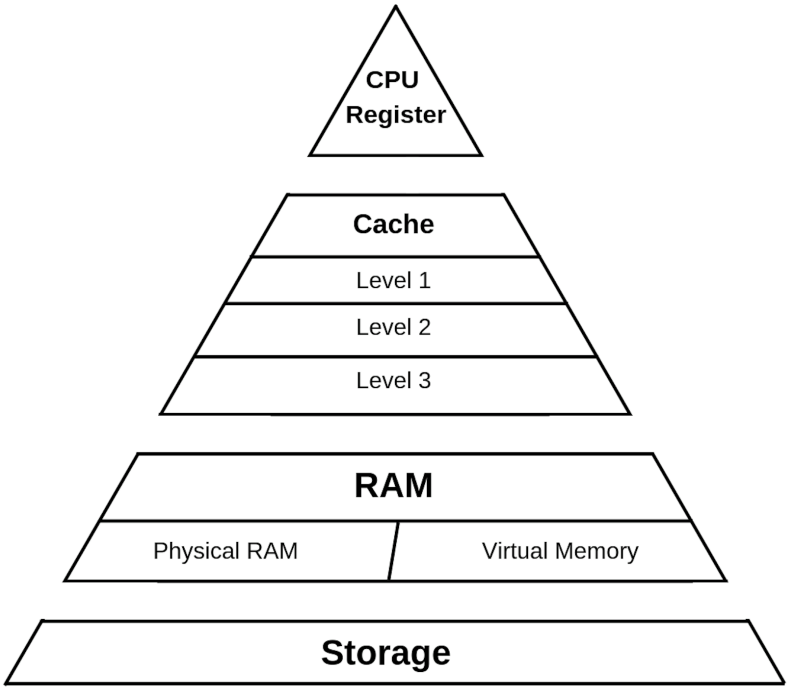


Figure 2.1: Memory Hierarchy

The memory hierarchy contains two main sub-layers: the primary and secondary storage. The primary storage, also known as main memory or fast memory, is the only that is directly accessible by the CPU. The CPU is responsible for reading the instructions stored in it and executing them. It is split into three sub-layers: CPU registers, CPU cache and RAM. A processor register is a location reference of a storage place on a processor that holds data that is being processed by the CPU and is at the top of the memory hierarchy, providing the fastest way to access data. The CPU cache will be discussed more detailed in the following Subsection 2.2. Moreover, the different types of RAM were already discussed in the previous Subsection 2.2.

The secondary storage, Storage in Figure 2.1, also known as slow memory or external memory, unlike the primary storage, is not directly accessible by the CPU. Therefore, the computer uses the input/output channels to access secondary storage data and transfers it to the CPU. However, the time taken to access data stored in the second storage is higher than accessing data stored in RAM. As a result, the CPU spends much of its time idling. Therefore, when it is necessary to access data that is accessible in slower layers, it offers an opportunity to design efficient External Memory algorithms, or also called Cache-Aware algorithms, which access the data in large sequential blocks to hide high latency, such as locality of reference.

Locality of reference is the tendency of a processor to access the same block of memory locations during a short period of time. The processor assumes that if a memory location i was accessed, the probability of accessing the $i+1$-th memory location is very high. There are several different types of locality of reference:

- Spatial locality: refers to the use of memory locations within nearly close storage locations;
- Temporal locality: refers to the reuse of memory locations during a short period of time;
- Sequential locality: refers to the arrangement and access of memory elements linearly, such as, traversing the elements in a one-dimensional array.

In computer architecture, *memory hierarchy* is a hardware optimisation that takes the benefits of using spatial and temporal locality on several levels of memory hierarchy, separating computer storage into a hierarchy based on response time. For example, a cache is an example of exploiting temporal locality. It is specially designed to keep recently referenced and nearby data, to potentially increase the performance of data access. However, the data elements in a cache may not be spatially close in the main memory, although they are all into the same cache line. Temporal locality has an important role on the lowest hierarchical memory level, keeping the referenced memory locations in the machine registers. Hence, systems that present strong locality of reference can achieve a better performance and improve the speed of the computer system through the use of techniques such as cache prefetching.

Cache prefetching is a technique used by computer processors, fetching data or instructions from a slower memory level to a faster memory level before it is necessarily needed. Therefore, because of the main memory design, accessing cache memory levels is faster than other memory levels, boosting the execution performance.

Central Processing Unit

The CPU, or main processor, is the unit that performs part of the processing inside a computer. It is constituted by two main components, Control Unit (CU) and Arithmetic Logic Unit (ALU). The CU is responsible for extracting the instructions from memory, decoding them and directing them to other parts of the system to execute them. The ALU is responsible for handling arithmetic and logical operations.

Each physical processor can have a single CPU or multiple CPUs, all connected by a common bus, the Front-side Bus, to the memory controller hub, known as the Northbridge. The Front-side Bus makes the connection between the cache and the Northbridge, while the Northbridge connects with the RAM. Therefore, the Front-side Bus transfers data to and from the CPU, at a certain speed (varies from processor to processor), and the Northbridge is responsible for the communication with the internal memory, RAM. However, some computers use two or more processors, consisting of separate physical CPUs. Therefore, each one has individual paths to the system Front-side Bus and separated caches.

On older systems, this substructure presented a particular bottleneck. The existence of only one bus to all the RAM chips, restrained the parallel accesses. However, recent RAM types, such as SDRAM, require two or more separate buses or channels, which more than doubles the available bandwidth.

Instructions pipelining The CPU fetches, decodes and executes instructions of a computer program by performing I/O operations, which send and receive data to and from the CPU. However, to keep every part of the processor busy and increase the program performance, some instructions can be divided into a series of sequential steps, performed in parallel and by different processor units. *Instruction pipelining* is a technique for performing instruction-level parallelism in a single processor, CPU. This technique attempts to divide incoming instructions into a series of sequential steps to keep every part of the processor busy. Conceptually, it consists of the following five steps:

- Instruction fetch: The instruction is fetched from memory;
- Instruction decode: The instruction is decoded to determine the operands and operation;
- Execute: Execute the operation itself;
- Memory access: Read operands from memory;
- Register the results: Write the result in memory.

Without pipelining, each instruction is processed entirely, from start to finish, before moving to the next instruction. However, modern processors can improve pipelining efficiency by assigning different instructions to multiple hardware sections. Therefore, the processor is capable of simultaneously fetch, decode and execute different instructions [49].

Central Processing Unit Cache

The CPU cache is one type of hardware memory that a computer processor core, an independent processing unit that reads and executes program executions, uses to reduce the average cost, time or energy, to access data from the main memory. It enables storing and provides access to recently or frequently used data. Typically, it is closer or within the processor core of a computer. Most CPUs have different caches where the data is organised as a hierarchy of more cache levels (see Figure 2.2). Moreover, the lowest level is Level 1 (L1) and the highest level, generally, is Level 3 (L3). However, commonly, each processor core is divided into two levels of cache, L1 and L2.

The L1 cache is a cache with very high speed of transfer, low latency, and with a size between 2 and 64 KB. Although, in some types of processors, L1 can be divided into two levels, data and instructions, L1d and L1i, respectively. The L2 cache is a cache with a medium speed of transfer, medium latency, and with a size of approximately 256 KB or 512KB. However, besides these two levels for each processor core, all cores also share a common cache, L3 with a size of 4 MB to 32MB and with a slow speed of transfer and high latency. Moreover, it is important to note that the cache access latency, normally measured in cycles, and speed of transfer, vary from cache level to cache level and from processor to processor.

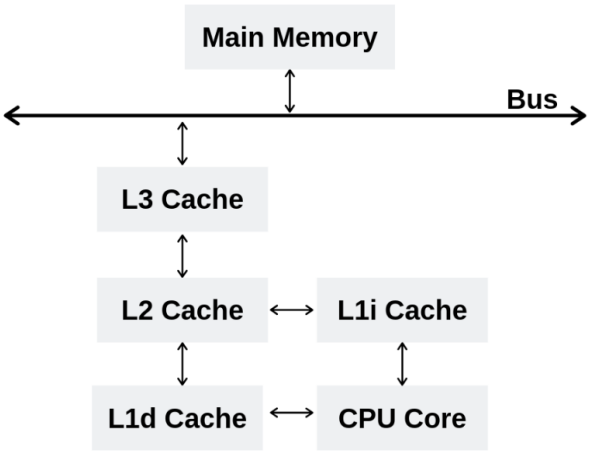


Figure 2.2: CPU cache levels of one processor core

Memory data is transferred in blocks of fixed size, called cache blocks or cache lines, between the main memory and the different cache levels or just between the different cache levels. However, when the processor needs to read or to write to a location in memory, it first searches for a corresponding cache entry in the cache before accessing the main memory.

After performing a read instruction and searching the requested memory location in the different levels of cache, two responses may occur, a cache hit or a cache miss. A *cache hit* arises when the processor found the memory location in the correspondent cache level and the processor immediately reads or writes the data in the cache line. Otherwise, if the processor did not found the memory location in the correspondent cache level, a *cache miss* occurs.

During a cache hit, the processor transfers the data to the lowest cache level, also applying a certain replacement policy and evicting some block to make room for the new entry. Moreover, due to instructions pipelining, during a cache hit or miss, the CPU can perform other instructions while waiting for the cache line to be fetched from memory. However, when the program exclusively performs memory access operations or runs out of instructions to perform, it reaches a state called *cache stall*. Therefore, cache stalls, and associated waiting time, due to cache misses can disturb the program performance.

During a cache miss, the cache needs to evict one of the existing entries and replace it by the new cache entry. The heuristic used to choose the entry to dismiss is called *replacement policy*. Cache efficiency is mainly dependent on the reuse of cached data entries. Therefore, the replacement policy must predict which cache entry is not likely to be used soon. However, each program performs different memory accesses, so there is no perfect method among the diversity of replacement policies. For example, one of the most popular replacement policies, the Least Recently Used (LRU) policy, discards the least recently accessed cache entries, while the Most Recently Used (MRU), in contrast to LRU, discards the most recently accessed cache entry.

Similar to a read instruction, a store instruction performs all the previously mentioned procedures. However, when the processor writes data to cache, the system needs to write this data to the main memory as well. The timing of this write is managed by the *write policy*. There are only two writing approaches, write-through and write-back. The *write-through* performs a synchronous write to both cache and main memory. The *write-back* initially writes only to the cache, and after this block be replaced by another cache block, it writes the data to the main memory.

During both read and store instructions, the accessed memory block is placed in a particular cache entry. This placement is decided by the *placement policy*. If the placement policy is free to choose any entry in the cache to place the main memory entry, the cache is called *fully-associative*. Moreover, the placement policy determines where each block of memory is placed and is restricted to a particular location, called *direct-mapped* cache. A direct-mapped cache is organised into multiple sets, each one constituted by a single cache line. Therefore, when a memory block is accessed, based on his memory address, it can only be placed in a single cache line of a respective set. However, recent cache placement policies, can place each memory block in more than one cache entry, particularly, to N places in the cache. Therefore, the cache is structured into N sets, each set constituted by M cache lines. This type of cache is described as N-way set associative. Moreover, a set-associative cache is a trade-off between both previously mentioned cache types, direct-mapped and fully-associative (see Figure 2.3).

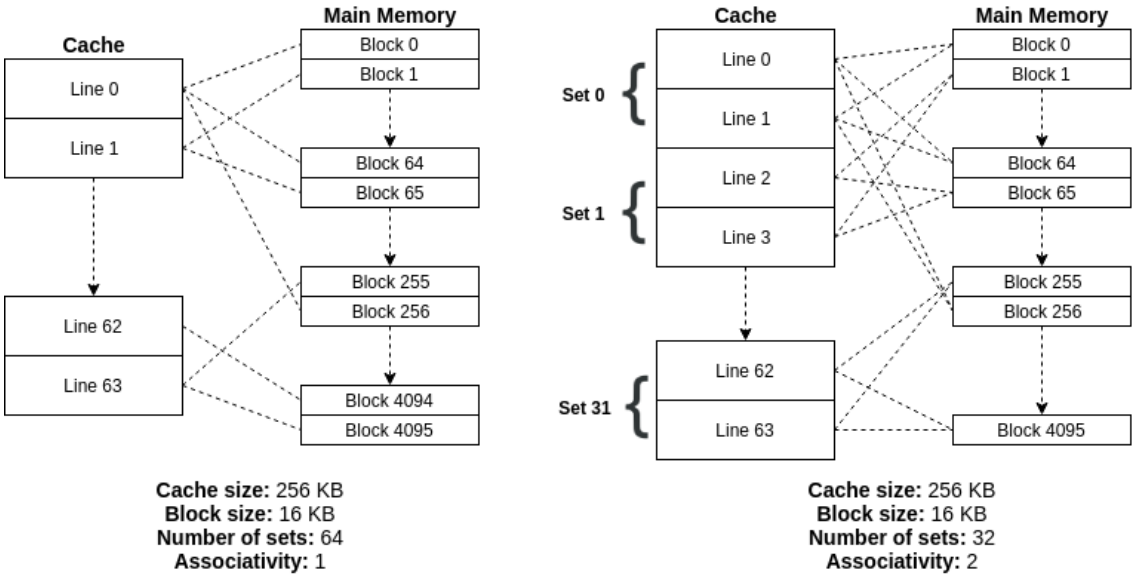


Figure 2.3: Direct-mapped cache (left) and Set Associative cache (right)

Moreover, each cache level contains its data blocks. However, the data blocks present in lower cache levels can be present, or not, in higher cache levels. This constraint is decided by the *cache inclusion policy*. If the data blocks present in the higher level cache is also present in the lower level cache, the lower level cache is said to be *inclusive* of the higher level cache. For example, considering the L2 cache inclusive of L1. If the requested block is present in the L1 cache, the data is read from L1. If the requested block is present in the L2 cache and not in the L1, the block is fetched from the L2 cache and placed in L1. However, if neither of the two caches contains the requested block, the block is fetched from memory and placed in both L1 and L2 cache.

Otherwise, if the data blocks present in the lower level cache is not present in the higher level cache, the lower level cache is said to be *exclusive* of the higher level cache. For example, considering the L2 cache exclusive of L1. If the requested block is present in the L1 cache, the data is read from L1. If the requested block is present in the L2 cache and not in the L1, the block is fetched from the L2 cache and placed in L1, and the evicted block is placed into L2. However, if neither of the two caches contains the requested block, the block is fetched from memory and placed just in the L1 cache.

Multicore architecture

A multicore architecture refers to a single physical processor, containing the core logic of two or more separate, but bundled, CPUs, called cores or virtual cores. Therefore, each core can run instructions at the same time, increasing the speed for programs that support Multithreading and Multicore parallelism (Section 2.5). Moreover, the more cores a processor has the more sets of instructions it can receive and process at the same time.

Within a processor, there are multiple levels of cache memory that contain data for the next operations of the processor. Figure 2.4, gives an example of a dual-core processor architecture.

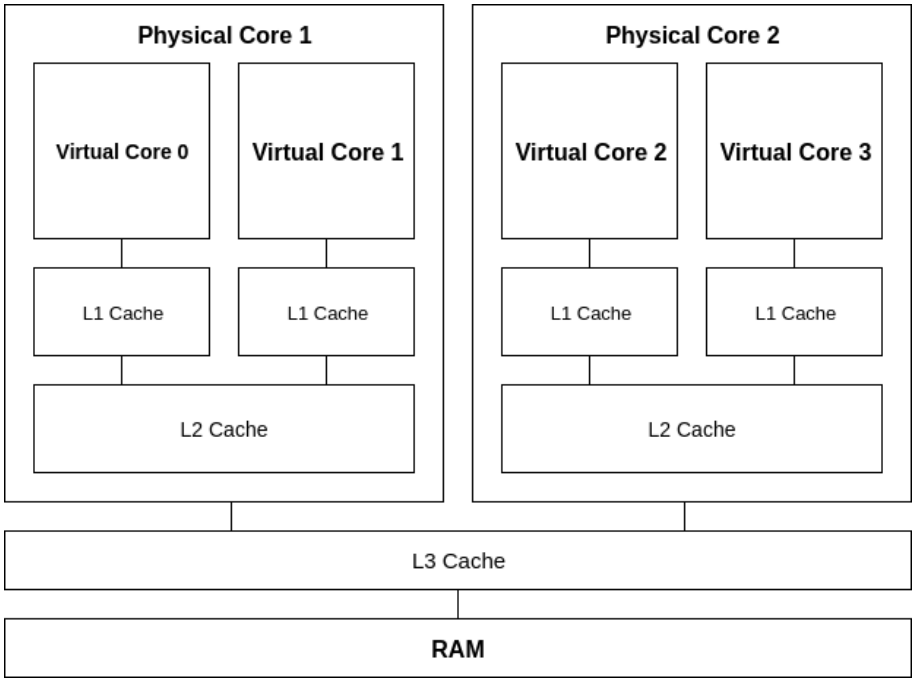


Figure 2.4: Multicore processor architecture with two physical dual-core processors

Normally, each core within a processor has its L1 cache and an L2 cache connected and shared by both L1 cache cores. However, the L3 cache is common to all processors. Moreover, the RAM is attached to the L3 cache.

In particular, when a processor needs to read or to write to a location in memory, it first searches in its cache. If the data is not present in this cache, a *distributed cache* miss occurs, and the data is then searched in the shared cache. Moreover, if it is not present in the shared cache either, then a *shared cache* miss occurs, and the data is loaded from the main memory in the shared cache and afterwards in the distributed cache. The same mechanism occurs when a core tries to write to an address that is not in the caches.

Furthermore, the cache share capability, is one advantage that can improve memory performance when using multithreading techniques. Particularly, since memory is shared, and data accesses are performed through a hierarchy of caches (see Section 2.2), from shared caches to distributed caches, taking further advantage of data locality, minimising data blocks movement.

2.3 Computational models

In this section, we discuss six computational models. First we discuss the Random-Access Machine and Parallel Random-Access Machine models. Then, we discuss the External Memory and Parallel External Memory models. Finally, we discuss the Cache-Oblivious and Cache-Aware models. These models are required to understand some concepts mentioned in Chapters 3 and 4.

Random-Access Machine

The Random-Access Machine [50], or Random-Access Machine (RAM) model, is a simple model of computation that can be used to measure the performance of sequential algorithms in a machine-independent way. This model assumes a single processor, P, and its memory consists of an unbounded sequence of registers or M memory locations. A computer program is a numbered list of statements and registers that are executed one after the other, with no concurrent operations, determined by the program counter. Under this model, there are three principal assumptions:

1. Each simple operation, such as an if or a addition operation, takes exactly one time step.
2. Loops and subroutines are the composition of single-step operations. Therefore, the time taken to run a loop depends upon the number of loop iterations.
3. Each memory access takes exactly one time step, ignoring the fact of whether or not an item is in cache. The model assumes also no shortage of memory.

For any given algorithm, the time complexity, according to the model, is assumed to be the number of time steps until it terminates and the space complexity used by an algorithm is assumed as the number of RAM memory cells.

Note, however, that the first and the third assumption are not realistic. For example, multiplying two numbers takes more time than adding two numbers on some processors. The third assumption suggests that the accessing an item in memory takes always the same time step, which is not necessarily true. Since it depends on whether data sits on cache or on the disk (see Section 2.2).

Parallel Random-Access Machine

The RAM model is a simple model of computation for an independent-way machine. However, for a parallel random-access machine with shared-memory abstract machine, a straightforward and natural generalisation of the random-access machine has been considered, the PRAM. The Parallel Random-Access Machine (PRAM) [51], or PRAM model, is a parallel-computing analogy for representing and analysing the complexity of parallel algorithms. A PRAM machine based on this model consists of multiple processors attached to a single block of shared memory, communicating and computing synchronously in parallel with the shared RAM memory.

Processors can perform various arithmetic and logical operations in parallel. Each operation can access, to read or write, any shared memory cell in one time step. Therefore,

N processors can perform independent operations on N shared memory cells in a particular unit of time. Moreover, simultaneous access to the same memory cell from different processors can occur. However, this model does not allow both read or write conflicts. Therefore, to resolve these conflicts, the model has the following strategies/constraints:

- Exclusive Read, Exclusive Write (EREW): only one processor is allowed to read or write to the same memory cell during the time step.
- Exclusive Read, Concurrent Write (ERCW): only one processor is allowed to read, but multiple processors can write to the same memory cell during the time step. However, this constraint is never considered due to the necessity of deciding the value to be written.
- Concurrent Read, Exclusive Write (CREW): multiple processors are allowed to read a memory cell, however, only one can write to the memory cell during the time step.
- Concurrent Read, Concurrent Write (CRCW): multiple processors are allowed to read a memory cell, however, only one can write to the memory cell during the time step.

Moreover, similar to the RAM model, under this model, there are three principal assumptions:

1. There is a unbounded collection of numbered processors, $P_1, P_2, \ldots, P_n$, and a unbounded collection of shared memory cells, $M_1, M_2, \ldots, M_m$, accessible from any processor.
2. There is no limit on the amount of shared memory in the system. However, the resource contention is absent.
3. The programs written under this model are generally of type Single Instruction Multiple Data (SIMD).

For any given algorithm, the parallel time complexity, according to the model, is assumed to be the number of time steps until the last P terminates its computation and the space complexity used by an algorithm is assumed as the number of shared memory cells accessed.

Note, however, this model ignores performance bottlenecks in modern architectures because assumes a single shared memory in which each processor can access any memory cell in unit time. Moreover, it ignores lower level architectural constraints, and details, such as memory access contention and overhead, and synchronisation overhead.

External Memory model

The External Memory (EM) model [52], or also known as the I/O model, assumes a two-level memory hierarchy, consisting of a fast cache with size M (M records), connected to the main memory, which is limitless and slow (see Figure 2.5). Both cache and main memory are divided into blocks of size B. Thus, the cache holds $\frac{M}{B}$ blocks. Moreover, it assumes simultaneously transfer of P physical blocks, each consisting of B contiguous records.

The model captures the fact that read and write operations are faster in cache than in the slow memory. Therefore, the concept of the external memory model allows to analyse

algorithms that process data sets that are too large to suit into the main memory of a computer at once. Since the CPU can only operate directly on the data stored in the cache, algorithms can transfer blocks between the main memory and the cache, depending on the operation. These specific algorithms must be optimised to accurately carry and access data stored in the slow memory. The cost of an algorithm, based on this model, is the number of blocks transfer required, considering operations over cache without cost. This model is one of the bases for the development of Cache-Aware algorithms, which are explained in Chapter 4.

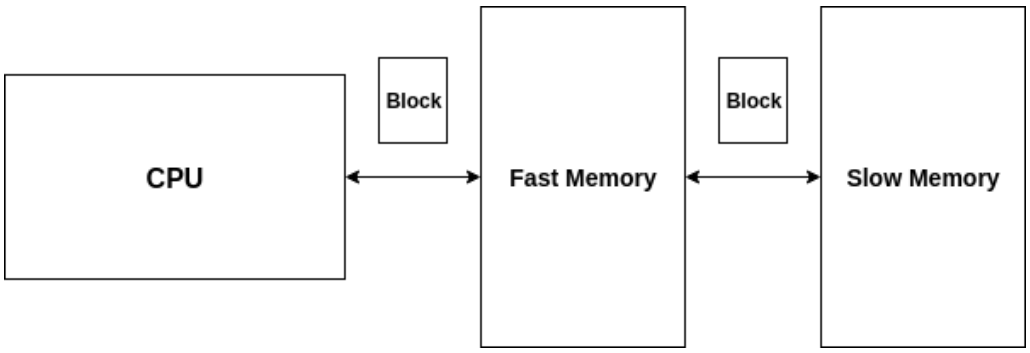


Figure 2.5: A representation of the EM model

Two-level Memory model

Furthermore, a realistic two-level memory model [34] with parallel blocks transfer was introduced. Similar to the previous model, it consists of internal memory capable of storing M records, P disks, each one capable of simultaneously transferring one block of size B. The model captures two basic ways of parallelism. In the first type of parallelism, blocks are transferred concurrently in contiguous blocks of size B, taking the same amount of time to access and transfer one block as it does one record. Then, in the second type of parallelism, P blocks can be transferred in a single I/O operation. However, to perform a realistic operation, the P blocks must be associated with tracks from P different disks. Therefore, only one track per disk can be accessed.

Parallel External Memory model

The Parallel External Memory (PEM) model [53] is an external-memory abstract machine, derived from the combination of the External Memory model and the Parallel Random-Access Machine model. Therefore, it can be seen as the parallel-computing analogy to the single-processor EM model and as an analogy to the PRAM. This model consists of a number of processors, P, each one with a private cache of size M partitioned in blocks of size B, and a shared main memory of size N partitioned in blocks of size B. Furthermore, processors can only perform access data contained in their cache, although, the data can be transferred from the main memory to the processor cache in blocks of size B (see Figure 2.6).

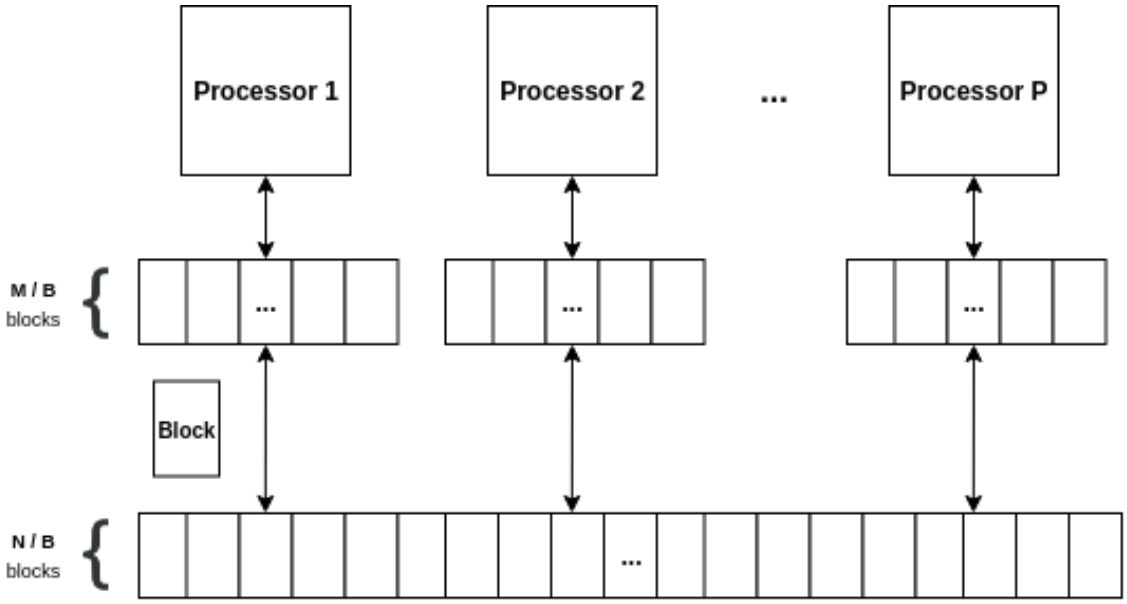


Figure 2.6: A representation of the Parallel EM model

Similar to the PRAM model, processors can access, to read or write, any shared memory cell in one time step, although, if different processors access the same memory cell, read or write conflicts occur. However, this model does not allow both read or write conflicts. Therefore, to resolve these conflicts, the model has the following strategies/constraints:

- Exclusive Read, Exclusive Write (EREW): only one processor is allowed to read or write to the same memory cell during the time step.
- Exclusive Read, Concurrent Write (ERCW): only one processor is allowed to read, but multiple processors can write to the same memory cell during the time step. However, this constraint is never considered due to the necessity of deciding the value to be written.
- Concurrent Read, Exclusive Write (CREW): multiple processors are allowed to read a memory cell, however, only one can write to the memory cell during the time step.
- Concurrent Read, Concurrent Write (CRCW): multiple processors are allowed to read a memory cell, however, only one can write to the memory cell during the time step.

Moreover, similar to the EM model, the complexity measure of an algorithm is determined by the number of reads and writes to memory.

Cache-Oblivious and Cache-Aware models

The concept of a Cache-Oblivious algorithm, sometimes called Cache-Oblivious model [54] and subsequently refined [35], is to design an algorithm that has a better cache performance without having the explicit knowledge of the size of the cache or the length of the cache lines as an explicit parameter.

The algorithm is designed to work without modifications in any machine without knowing the cache structure or memory architecture, which consists of a generalisation of the EM

model to a multilevel memory model. Typically, Cache-Oblivious algorithms use a recursive divide and conquer approach that repeatedly divides the data set, creating sub-problems, until it eventually becomes cache resident, that is, reaching to a sub-problem that fits into the cache, regardless of the cache size.

Cache-Oblivious algorithms are usually analysed using an idealised model of the cache. A real cache has characteristics that involve one of the three different policies available for placement of a memory block in the cache, so-called cache replacement policies:

- Direct mapping: The cache needs to be organised into multiple sets and each block can only occupy a single cache line.
- Full associativity: The cache is organised into a single cache set with multiple cache lines and each memory block can occupy any of the cache lines.
- Set associative: Trade-off between mapping and full associativity.

This model consists of a two-level memory hierarchy consisting of a cache of M items and a large main memory. We assume that the faster level cache is the cache of M items and the slower is the main memory. The cache is partitioned into cache blocks of consecutive B items that are transferred between cache and main memory, denoted as *memory transfer*. The CPU processor can only refer to data that is accessible in the cache. If the processor accesses data that is in its cache, a cache hit occurs, otherwise, a cache miss occurs, and the block where this data belongs is moved to cache, replacing other. The idea of this model resides in replacing the cache block whose next access is furthest in the future and, thus, it exploits cache data arrangement avoiding cache misses and consequently blocks replacements that can be required a few instructions later.

The model is built upon some assumptions, described as follows:

- Tall cache assumption: The cache holds M objects where $M = \Omega(B^2)$, an asymptotic lower bound that says M is larger than some constant times B^2.
- Optimal replacement: When a cache misses occurs and the cache is full, a block needs to be replaced and in most hardware this is implemented as First-In, First-Out (FIFO) or LRU. The model assumes that the cache line chosen for replacement is the one that will be accessed furthest in the future.
- Full associative: Each block can be loaded from the slower level of the memory into any part of the faster level.
- Automatic replacement: When a block needs to be brought to faster level of the memory, it is automatically done by the OS without prevention of the algorithm design.

A Cache-Oblivious algorithm is designed without knowing the cache structure or memory architecture parameters sizes. However, we can design algorithms taking into account the knowledge of the memory architecture, such as the size of the faster level cache size and the cache blocks size, a Cache-Aware algorithm.

The Cache-Aware algorithms [52], also known as External Memory algorithms, is an idealised model of computation of the External Memory model. This model captures the memory hierarchy and has the knowledge of the memory hierarchy and architecture, such as cache levels and cache line sizes, unlike previous discussed models. For this reason, the

model is called as the Cache-Aware model. Particularly, the *Cache-Aware model* simulates an abstract machine similar to the RAM machine model, but with a cache in addition to main memory. The model consists of the same structure as the Cache-Oblivious model, where a processor with internal memory or cache of M items is connected to the unbounded external memory where both are divided into blocks containing B items. The process of memory transfer is also identical. Moreover, the complexity measure of an algorithm is determined by the number of reads and writes to memory.

2.4 Cache optimisation techniques

In this section, we present some cache optimisation techniques to arrange and access data in computer memory, in particular, data alignment, matrix vectorization and the loop tilling technique.

Data structure alignment

Data structure alignment refers to the way data is arranged and accessed in computer memory. Data alignment, data structure padding and packing are three separated but related issues and together known as data structure alignment. The *data alignment* means to put the data in a memory address that is equal to some multiple of the word size. However, for example, when dealing with data structures, the compiler may insert additional unsigned data members, extra bytes, between the end of the last data structure and the start of the next data structure. This insertion of extra bytes of memory to align the data is named *data structure padding*. Padding data does not contribute to the functionality of the program, but it grants data exclusivity to the entire cache line. However, by changing the ordering of the structure members, it is possible to achieve the quantity of padding necessary to align the structure and sometimes even reduce the memory required. This reordering of the structure members is also called *packing*.

The data that is not aligned to the cache can lead to performance degradation [55]. For example, when a program needs to read contiguous data from memory, the processor continuously transfers data to the cache lines. However, if the data is misaligned, the program cannot take advantage of the data present in the cache lines. Therefore, it is beneficial to allocate memory aligned to cache lines.

There are several ways to ensure that an object is aligned to the cache lines (see [56, 57, 58]). For example, in the C programming language, we can perform these two techniques:

- the object can be allocated with a specific alignment requirement, using dynamic allocation calls such as *malloc*, with auxiliary attributes, or *posix_memalign*;
- the required alignment can be modified by the compiler by using a type attribute such as *_attribute((aligned(#alignment)))*, in the structure declaration, where *#alignment* means the number of bytes.

Matrix vectorization

Commonly, in computing, a matrix is represented as an array of two dimensions. However, when dealing with dense matrices, this representation can influence algorithm performance.

Despite the extra memory required to allocate a two-dimension dynamic array, because of the pointers to the set of one-dimension arrays, the memory will not be contiguous or aligned. As mentioned above, memory alignment represents an important technique when performing multiple and contiguous access operations over a matrix. Therefore, to achieve a better performance and spatial locality, it is possible to perform a linear transformation, especially used in linear algebra and matrix theory, namely vectorization. *Vectorization* transforms an $M \times N$ matrix to a $M \times N$ column vector.

As observed in Figure 2.7, the one-dimensional array is contiguous in memory. However, the two-dimensional array, besides the extra pointer to follow and consequently more memory allocated, is not contiguous in memory. Therefore, loses the cache locality.

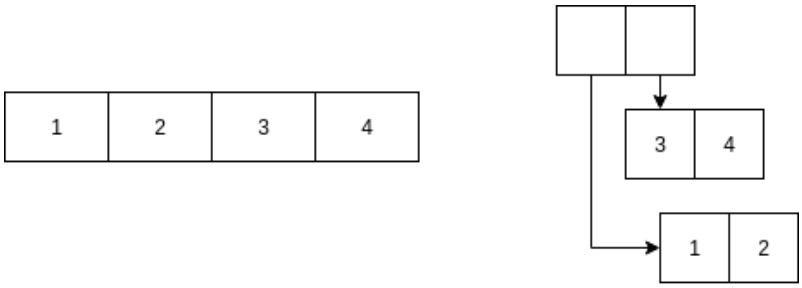


Figure 2.7: One-dimensional and two-dimensional matrix representation

Loop tilling

Loop tilling, known as loop blocking or loop unrolling, is an optimisation technique that applies a set of loop transformations that exploits spatial and temporal locality of data accesses in nested loops, dividing the problem into smaller problems, with the purpose of achieving a better memory locality performance and reducing the overhead of nested loops. Moreover, this technique is mainly used to treat large amounts of data, specially matrices, and to decrease memory access latency or cache bandwidth in some linear algebra algorithms [59, 60]. Furthermore, this technique is documented in the newest compiler and architecture texts [61].

Each loop transformation, usually on *for loops*, rewrites a single loop into two loops: one iterating inside each block and other iterating over the blocks. Therefore, this partition leads to the reuse of the cache and to fit accessed array elements into cache size, eliminating cache size requirements. Moreover, the loops order and block size have an important role in improving cache performance.

For example, Algorithm 1 returns the elements sum of an array M with N size. This algorithm divides the array into blocks of sixteen elements to increase the spatial locality.

Algorithm 1 Loop tilling - Algorithm example

```
1: procedure SumArray
2:     sum ← 0
3:     for i ← 1 to N by 16 do
4:         for j ← i to min(i + 16, N) do
5:             sum += M[j]
       return sum
```

2.5 Multithreading and multiprocessing

Hardware manufacturers have been improving lower-level layers of the hardware to develop computer architectures for faster processing and mainly, to increase instruction-level parallelism. Moreover, several generations of processors have been built with higher clock frequencies to improve the performance of applications. However, due to the physical architecture limitations, in particular, power consumption and heat dissipation, a stagnation point has been reached [38, 62]. Therefore, they turned their focus on the increase of the number of physical components, in particular, cores, given the rise to multicore architectures (see Section 2.2). These architectures enable the spread of instructions over different performing units, allowing thread-level and task-level parallelism.

There are several ways to parallelise the program execution and, consequently, accelerate it. Parallel computing is a type of computation in which multiple processes are being executed simultaneously. It uses specific computation abilities to process different elements simultaneously, such as multithreading and multiprocessing. Parallelism is accomplished partitioning the problem into independent parts so that each process executes a specific part of the program simultaneously with the others, bringing a significant increase in performance and speed up. Moreover, there are two main principles: instructions decomposition and data decomposition. *Instructions decomposition* means that different threads perform different works, while *data decomposition* splits the data into multiple blocks to multiple processors. However, possible gains are limited by the software that can run in parallel simultaneously, especially on multiple cores, which is an effect described by Amdahl's law. The Amdahl's law [63] is often used to predict the theoretical speed up on multicore architecture and to calculate how much a computation can be sped up by running part of it in parallel. Moreover, it states that in parallelization, if P is the part of the task that can be made parallel, and $1 - P$ is the proportion that remains serial, then the maximum speedup that can be achieved using N number of processors is equal to $\frac{1}{(1-P)+\frac{P}{N}}$.

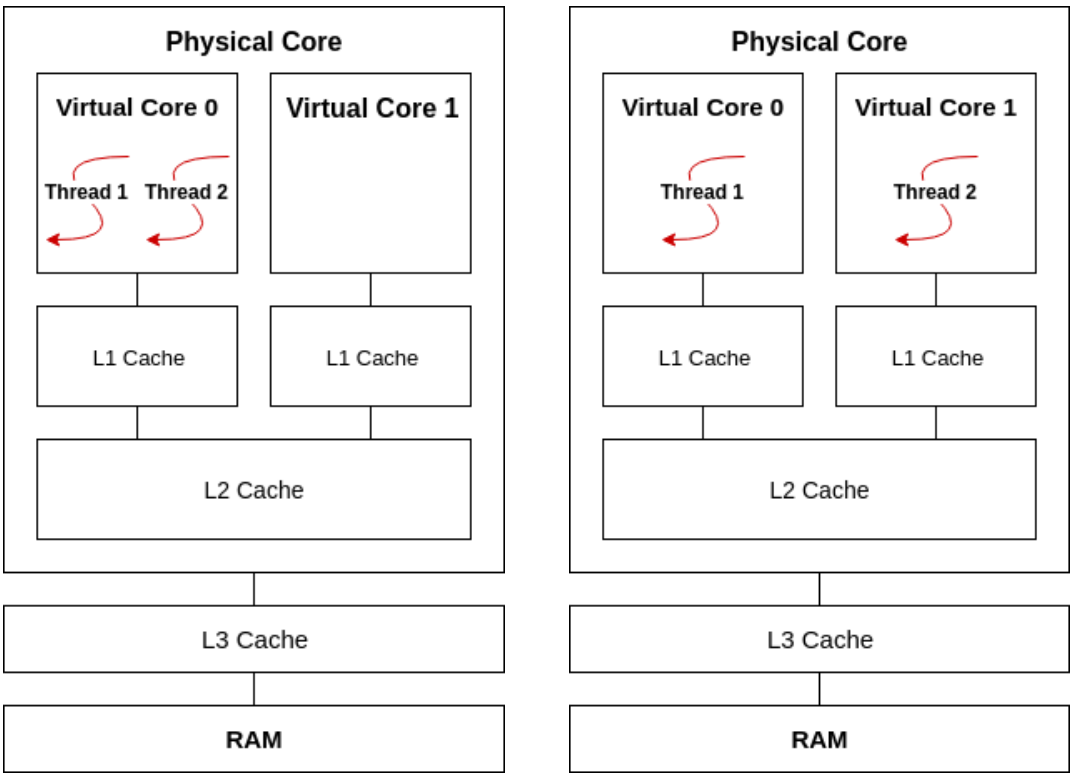
Multithreading is a programming and execution model and a CPU ability that allows multiple threads to exist within one process, which is mainly found in multitasking OS (Figure 2.8a). A thread is a sequence of instructions that can be controlled independently of a schedule. However, threads that belong to the same process share system resources and data. Therefore, instead of creating separate processes for each task request, it is more efficient to create threads of a process. Each operating system has it is own way to implement threads and processes. A process can be divided into multiple threads, executing concurrently and sharing multiple resources such as memory. However, processes do not share resources. The usage of multithreading technique brings some advantages:

- Responsiveness: it allows an application to remain responsive to input, while in a single-thread program, if the main execution thread blocks on a long-running task, the entire application may slow down.
- Faster execution and lower resource consumption: the program operates faster and consumes less resources.
- Parallelization: it allows an application to split data and tasks into parallel sub-tasks.
- Better system resource utilisation: for example, the application can reuse data present in the shared cache memory.

These advantages can improve significantly the overall performance of the application.

However, performance is also dependent on several factors, such as the application type and threads synchronisation.

Multiprocessing is the use of two or more CPUs and a system ability to allocate tasks for each core, within a single computer system (Figure 2.8b). At the operating system level, multiprocessing means parallel execution of multiple processes using more than one processor.



(a) Multithreading (b) Multiprocessing

Figure 2.8: Difference between executing a program using two threads and one virtual core, multithreading, and using two threads and two virtual cores, the combination of multithreading and multiprocessing

As observed in the previous Figure 2.8 and explained in Section 2.2, different virtual cores in the same physical core, share the same two-level cache. This can improve application performance since the communication between threads becomes faster and they can share the same code and data. However, if different threads use different memory areas, the cache will be filled up with data blocks that will not be the most possibly used. Moreover, there will be cache contentions if different threads write to the same memory areas and due to the cache placement policies (see Section 2.2), if more than one thread writes to the same cache line, it will invalidate others processors accesses and cause large delays.

This page is intentionally left blank.

Chapter 3

State of the Art

In this chapter, we review empirical and theoretical studies about energy efficiency, cache efficiency, and multithreading and multicore efficiency. Note that definitions and theoretical models presented in Chapter 2, are essential to understand some relevant concepts found in this chapter.

This chapter is divided into 4 sections. We start by reviewing the Energy Complexity model in Section 3.1. In Section 3.2, we discuss empirical studies on energy efficiency. In Section 3.3, we discuss empirical studies on cache efficiency. Finally, in Section 3.4, we discuss empirical studies on multithreading and multicore efficiency.

3.1 Theoretical studies on Energy Efficiency

In Subsection 3.1.1, we review the Energy Complexity model and an empirical study on the model.

3.1.1 Energy Complexity Model

The Energy Complexity Model was proposed in 2013 by Swapnoneel Roy et al. [33]. This model is based on design of modern processors and memory because of its architecture which is organised in banks, rows and columns. It is also similar to the Parallel EM model of Aggarwal and Vitter [34] [64], which presents a simple energy model represented by the sum of the time complexity of the algorithm and the number of I/O accesses made by the algorithm. The energy complexity of this model is constrained in terms of time complexity, in the RAM model (see Section 2.3), and its I/O complexity, in the EM model (see Section 2.3).

The memory architecture of this model, or memory model, is divided into a set of parallel P banks, each having its own cache. Each bank is also constituted by blocks, each one formed by B items and $\frac{B}{s}$ strides, where s is the number of items in each stride. Moreover, each stride behaves as a cache line.

Different from other models, in this model, each cache only holds one block. Therefore, an algorithm can manipulate the items in all caches whenever required, which can be called a "big cache" since it has $P \times B$ items available. Since an algorithm has at its disposal P banks, the number of I/O accesses made by the algorithm is the number of parallel I/O made to the P parallel banks. However, blocks needed in parallel are written to different

disks, which is ensured by a constraint in the algorithm design. This division of the memory architecture into a set of parallel P banks could be also P-Way, as named by the authors. Figure 3.1 illustrates the architecture of this model for $P = 4$ (4-Way).

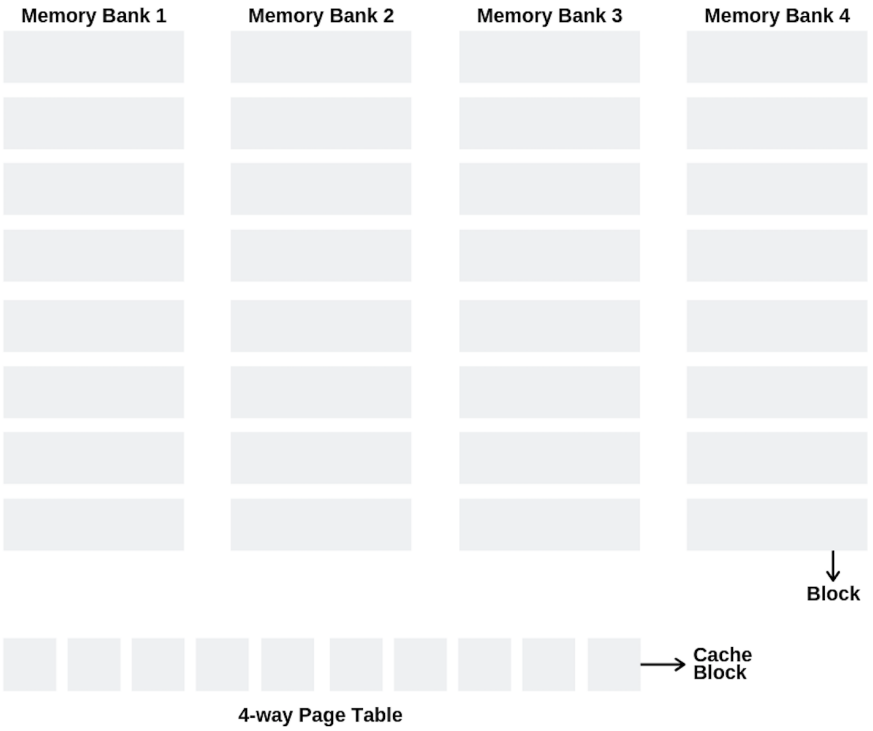


Figure 3.1: The Energy Complexity model - Memory layout for parallelism with $P = 4$

In spite of some similarities with the Cache-Oblivious model (see Section 2.3), this model differs in some ways. First, it has an associativity of one, i.e., every bank has exactly one cache block. Moreover, each block in slow memory can be mapped to one slot in fast memory, whereas cache-oblivious model assumes "ideal cache" with high associativity. Second, it does not have a *tall cache assumption* (see Section 2.3). Finally, its main aim is to minimise a linear combination of work complexity and the number of parallel I/Os whereas the cache-oblivious model tries to minimise both with no concern with parallelism.

Roy et al. show that this model captures the energy consumed by a modern server for executing an algorithm, focusing exclusively on server power, which includes the power drawn by the processors and the server memory. Therefore, in order to achieve those values the authors presented a model for the processor energy and another for the memory energy. After combining these two and after some simplifications, the authors propose the following expression for total energy consumed for an Algorithm A,

$$E(A) = W(A) + ACT(A) \times \frac{P \times B}{k} \tag{3.1}$$

where E(A) is a function that returns the total energy consumed, W(A) reports the total time taken by the non I/O operations and ACT(A) is the number of activations to read or write a row that contains the memory word needed. The k value is the average number of I/O access made in parallel by the algorithm.

In this abstract model, energy consumption is very closely tied with the memory layout.

Therefore, it emphasises that P can be a few orders smaller than B, the opposite from the *tall cache assumption*, in the Cache-Oblivious model (see Section 2.3), where $P \geq B$. Considering the previously explained memory model, they aligned their energy model with existing parallel I/O models [34] (see Section 2.3). Therefore, they have defined $W(A, x)$ as the work performed by the algorithm A, without accounting for the time to transfer blocks from/to main memory, with an arbitrary input x. Moreover, they defined $T(A, x)$ as the sequence of batches. Therefore, they described the energy consumed by A on x as the sum of the work that an algorithm performs on non-I/O instructions plus the latency times the number of accessed batches/strides:

$$E(A,x) = W(A,x) + L \times T(A,x), \tag{3.2}$$

where L is a latency parameter with $L = \Theta(P \times B)$.

Empirical studies on the Energy Complexity model

Swapnoneel Roy et al. also performed an empirical study to validate their model [33]. They defined an operating point with a fixed work complexity, i.e., varying the allocation of the data across the banks in a way that the algorithm would have varying degrees of parallelism, $P \in \{1, 2, 4, 8\}$, and a fixed number of memory accesses. Their energy modelling experiment was conducted using three benchmarks, with varying values for P. The energy optimal algorithm proposed in [33], requires data to set out in memory with a certain degree of parallelism. Therefore, they proposed a generic way to ensure memory parallelism for a given input access pattern or vector.

The Energy Complexity model assumes that memory is divided into a set of parallel P banks and each bank, besides the own cache, is constituted by blocks, each one formed by B items. Therefore, to ensure memory parallelism for a given vector, it requires two pre-processing steps. The first step sets the P banks and the second step defines the page table according to the value of P. For the first step, after receiving the input vector of size N, the algorithm creates a vector V with N elements and creates a logical mapping which ensures access to the vector in P-way parallel. However, only power of two values are valid to simplify the blocks accesses through bitwise operations. First, contiguous memory blocks of size B are created for each bank. Then a mapping function converts V into a matrix M of dimensions $\frac{N}{B} \times \frac{B}{s}$, where B is the size of each block and s is the stride size. Each block in M consists of $\frac{B}{s}$ strides, each of size s. To fill the matrix M, a function assigns strides to $P \times B$ blocks. For the second step, a page table vector T of size $\frac{N}{B}$ is defined with the ordering of the blocks in order to know the location of each stride. The ordering is based on the way the blocks are accessed. Therefore, if P equals one, the algorithm ensures that consecutive blocks are in the same bank, otherwise, blocks are assigned to banks in a round robin order.

They considered several benchmarks. The first benchmark problem consists of a program that writes data into an integer vector with a specific access pattern. The results for this benchmark showed that the higher the value of P, the lower the energy consumed. However, we can notice that the largest difference occurs between the 1-Way and the 2-Way, being that the difference between the others decreases as the value of P gets larger.

The second benchmark problem consists in copying one large integer vector into another. For the second benchmark, the results were quite similar to the first benchmark, showing a benefit in using parallelism. As in the first benchmark, the largest energy consumption difference occurs between the 1-Way and the 2-Way, although, with less discrepancy than

that of in the first benchmark. The difference between the energy consumed by the other values of P it is not very significative, although, the difference between the values of P increase according to the vector size.

The authors considered the transpose of an integer matrix as the third benchmark problem. For this problem they proposed the Blocked Transpose Algorithm (see Section 4.2). Unlike the two other benchmarks, the differences between the values of P do not show that much discrepancy. However, we can notice that the larger is the problem size, the larger is the difference between the different values of parallelism and that the biggest difference occurs between the 1-Way and the 2-Way.

The authors performed another empirical study in order to validate the key concepts of the energy model [65]. They performed experiments over seven benchmarks related with vector operations, matrix transpose, sorting and graphs algorithms using the same methodology as in the previous work [33]. The experiments over the sorting benchmark problems used three known sorting algorithms, *selection sort*, *quicksort* and *mergesort*. The experiment results show that the number of parallel banks, P, used by an algorithm has indeed an important role, particularly the larger is the value of P the larger is the energy efficiency. Moreover, they did similar experiments with modified versions of *Breadth-first Search (BFS)*, *Depth-first Search (DFS)* and *Dijkstra* algorithms. The results show that the gains in going from 1-Way to 8-Way was in line with the previous experiments. However, they noticed that parallelizing the layout does not lead to energy savings if the auxiliary data structures, such as *neighbour list*, do not fit into cache.

As a conclusion, the model suggests that energy consumed by an algorithm can be partitioned in two components, memory accesses and CPU instructions. Moreover, the empirical results suggest that the memory organisation can also influence energy consumption. Therefore, in our work, we will expand this analysis exploring the relation between memory accesses and CPU instructions with energy and time, and analyse the impact of different memory access patterns and data organisation.

3.2 Experimental studies on Energy Efficiency

Several empirical studies have been made aiming at understanding the impact of software on energy consumption from different perspectives. In Subsection 3.2.1, we review literature focused on the energy efficiency of cache architecture and data management. Then, in Subsection 3.2.2, we review literature focused on the energy efficiency of some algorithms implementations.

3.2.1 Cache architecture and data management

The empirical study by Soontae Kim et al. [22] focused on partitioning the cache resources architecturally for energy efficiency. They examined ways of splitting the cache into smaller units, designated *subcaches*, to reduce per-access energy costs and eventually improve the locality behaviour as well. Moreover, they proposed a *subcache* architecture to improve the cache performance and memory system energy efficiency. They claimed that dynamic energy consumption in the cache can be lowered by reducing the number of accesses to the cache. Therefore, they performed two optimisations for cache energy reduction, dynamic page remapping and *subcache* prediction. The obtained results at the energy level showed that by using *subcaches* configurations and varying the size of the direct-mapped cache it is possible to reduce on average the energy consumption in the memory system by 60% up-to

83%. As a conclusion, while the performance of the memory system is mostly influenced by the number of cache misses, both cache misses and accesses influence energy consumption. Therefore, the optimisation at the architecture level, such as *subcache* architectures, are crucial for reducing cache energy costs.

Another study by Michele Co et al. [23] analysed the trace cache energy efficiency. Trace cache is a specialised cache that stores the dynamic stream of instructions, called trace, to increase the fetch bandwidth by storing traces of instructions already fetched. Therefore, their experiments consisted in evaluating whether concatenating basic blocks, a straight-line code sequence with no branches, translates to energy efficiency. Moreover, they compared trace caches and instructions caches in both overall performance, such as time and memory, and energy efficiency. Although trace cache achieved better results than instructions cache, it was the branch-prediction that more strongly influenced overall performance and energy. Moreover, results showed that a similar performance can by achieved applying sequential trace caches or instruction cache-based engines.

Liu et al. [24], focused on the energy impact of data management choices by programmers, as data organisation or data access patterns and the interaction between hardware-level energy management and application-level management, over the programming language Java. Therefore, five programmers choices were analysed: *data access pattern*, *data organisation*, *data representation*, *data precision* and *data I/O strategies*. For the data access patterns, they measured the energy consumption when accessing a large array under sequential or random access, considering both read and write operations. As expected, the random accesses consumed much more energy than sequential. Moreover, the energy and performance achieved by read and write instructions were not proportional, which could be explained by the overhead of each hardware instruction. However, the energy consumption remained stable at the DRAM level. For the data representation strategies, they measured the impact of representing a sequence of integers using a primitive array and an ArrayList. The results showed that, as expected, that the method invocation used to access indexes using an ArrayList was significantly more expensive than using a primitive array. For the data organisation, they considered two programs, the first used a large array of objects with five fields, and the second used five primitive arrays, performing both read and write operations. They concluded that, although Object-oriented paradigm provides some benefits such as modularity and maintainability, it does not benefit energy consumption. For the data precision choices, they analysed the energy consumption of the matrix multiplication operation using different primitive types. The obtained results showed that the matrix of **double** data type consumed 1.45 more energy than the **int** type and 4.95 more energy than **short**.

The second analysis of this empirical study was concerned with the interaction between hardware-level energy management and application-level management. Although scaling down the CPU frequency effectively saves power, it may increase the running time of a program. Therefore, they performed an analysis varying the operation supply voltage of the CPU and the operating frequency. They concluded that downscaling the CPU not only results in a performance loss but also in an increase in energy consumption.

The previous empirical studies showed that cache architecture, different types of structures and primitive types, and different CPU frequency can have different energy consumption behaviours. However, these studies ignored other factors, such as the correlation between the number of cache misses and the primitive types, and the relation between memory accesses and energy consumption. In our work, we will analyse the relation between memory accesses, energy consumption and running time. Moreover, to understand the impact of different memory access patterns on cache performance, we analyse the number of cache

references and misses.

3.2.2 Algorithms implementations

The experimental study by Rashid et al. [21] consisted in comparing the impact of some sorting algorithms with different computational complexity in terms of energy consumption. They implemented the algorithms in three different programming languages: ARM assembly, C and Java. For each language, algorithm and data set size, they collected the number of instructions per cycle, percentage of cache misses, percentage of branch misses, energy consumed and running time. They obtained different levels of energy consumption for both different algorithms and languages. In particular, the Counting sort exhibited better performance, followed by the Quicksort. The most energy efficient language was ARM assembly. As a conclusion, a large part of the energy consumed by the algorithm was determined by the time performance. However, some factors were not considered, such as memory accesses. In our work, we will explore this relation between memory accesses, time and energy.

3.3 Empirical studies on Cache Efficiency

Tsifakis et al. [37] focused on the analysis of the Matrix Transposition algorithms: Cache ignorant, Blocked and Cache-Oblivious. For each algorithm, they performed an experiment, measuring hardware performance counters, and a simulation. They noticed that the Cache-Oblivious algorithm presented some unexpected results. The cache miss profile presented poor results when the matrix row dimension was an exact multiple of the cache size, such as 4096 and 8192. Therefore, they decided to analyse the associativity of the cache. The results showed that increasing the cache line associativity achieved better results, although its implementation does not pay off the difference. For the real experiment, they used two machines with two different systems. The first system has a 16 KB direct-mapped data cache and the second with a 64 KB 4-way set associative data cache. In the first system, they observed that the obtained results were similar to the simulated, although slightly higher, especially in both 4096 and 8192 matrix dimensions. However, the second system presented better results. These results showed that the use of a data cache with associativity could achieve better results, especially in the problematic matrix dimensions sizes previously referenced. As a conclusion, they confirmed that the Cache-Oblivious Algorithm has a significant impact on performance. However, the algorithm's performance is dependent on several factors, such as cache size, matrix dimension, cache line size, cache associativity and cache policies. In our work, we will extend this empirical study and analyse the impact of this memory access pattern on energy consumption and running time.

3.4 Empirical studies on Multithreading and Multicore Efficiency

In Subsection 3.4.1, we review empirical studies on the energy efficiency of multicore architectures and multithreading processes. In Subsection 3.4.2, we review empirical studies on the cache efficiency of multicore and multithreading architectures and algorithms.

3.4.1 Empirical studies on Energy Efficiency

The empirical study by Chengling Tseng et al. [41] evaluates the power consumption of multithreaded processes on multicore machines. They used the Jacobi algorithm as a benchmark to investigate the effect of the degree of parallelism on different frequency configurations. They observed that an increase on frequency corresponds to a decrease on the overall energy consumption. However, speeding up the process drained more energy from the voltage source, which inhibits the overhead when the frequency is lower. Moreover, they noticed that some energy savings were possible to obtain due to the context switch, such as saving local variables in the stack or temporary locations. Therefore, the entire system can suffer in terms of time, leading to a total energy loss. As a conclusion, their analysis showed that the frequency and power consumption have a linear relationship and that the most energy efficient strategy is to "finish fast and shut down".

Other empirical studies focused on cache energy reduction techniques such as turning off parts of the cache in order to reduce static energy [66] or into low-power state-preserving mode [67] or into low-power state-destroying mode [68].

According to the empirical research conducted, there is no empirical study on the analysis of the energy efficiency of memory accesses and CPU instructions of a multicore or multithreading algorithm. In our work, we will explore this relation between memory accesses and CPU instructions with energy and time over multicore and multithreading techniques.

3.4.2 Empirical studies on Cache Efficiency

An empirical study by Jacquelin et al. [44] analysed the number of cache misses during the computation of Matrix Multiplication operation. Moreover, they tried to minimise the number of cache misses as well as the predicted data access time of the algorithm. The authors used three different algorithms as benchmarks, the first focused on more efficient memory allocation, and the other two on distributing data blocks among cores. They also implemented different versions of the first algorithm to minimise the number of shared cache misses, distributed cache misses and data access time with different data replacement policies. The results showed that minimising the number of shared cache misses or the number of distributed cache misses on the first algorithm achieved better performance. Furthermore, they observed that minimising the data access time offers the best performance, with a trade-off between shared and distributed cache misses and a predominance of distributed misses.

Tzul [43] performed an empirical study identifying the parts of several algorithms that can be parallelized. The analyse focused on three algorithms, Matrix Multiplication, and the parallel version of two graph algorithms, BFS and Floyd-Warshall. The two main matrix multiplication algorithms were the traditional sequential algorithm and its parallel version. The results for the first two matrix multiplication algorithms suggested that keeping and reusing data close to the processor increases the cache performance. For example, dividing the matrices into smaller sub-matrices improves the performance by keeping data close to the cache. However, with parallelism, the cache performance can degrade and influence the execution time. The parallel BFS algorithm needs a synchronisation after each level to stop the process of updating the next level in the graph while other processors are still working on the previous level. This synchronisation added overhead on the algorithm. The parallel version of the Floyd-Warshall algorithm is a transformation of the iterative algorithm to a recursive algorithm. The idea of the algorithm is to divide the problem into smaller problems and take advantage of the data present in the cache. However, due to the

dependencies in the algorithm, the ordering of the recursive calls is essential. Moreover, despite the recursive algorithm presenting better results, some recursive sub-tasks can not be executed in parallel. Therefore, the recursive algorithm just takes advantage of the cache.

Both empirical studies analysed the cache performance in different algorithms and different paradigms. However, they did not explore the combination of three factors: energy consumption, running time and cache performance. Our work will explore this relation between memory accesses and CPU instructions with energy and time on multicore algorithms, and analyse the impact of multicore and multithreading techniques in energy consumption.

Chapter 4

Algorithms for Matrix Transposition

We aim to understand the relation between the energy consumption of an algorithm, its running-time and cache performance considering different memory access patterns. The Matrix Transposition operation was chosen as our benchmark because it is an operation mainly dependent on memory accesses and the cache performance, and that can be resolved using different memory accesses patterns.

Matrix transposition is a fundamental operation in linear algebra and used in many real-life situations, e.g. [46, 47, 48]. This operation consists on reading and writting an element from memory in a different location. In general, two-dimensional arrays, or matrices, can be represented using two different layout formats: row-major or column-major. In the former format, adjacent elements within each row are, contiguous in memory, therefore, traversing a row-major matrix along rows is much faster than along columns. However, performing a matrix transposition for large matrices sizes can have a tremendous impact on cache performance.

The order in which matrix elements are swapped has a strong impact in the performance, especially in memory. This problem uses data elements within relatively close storage locations, which can lead us to a lack of spatial locality. For instances, when accessing the matrix elements in row-wise, memory is accessed continuously in each row. For example, when the element `[1,1]` of the matrix is accessed, it is fetched to the cache line and to the lowest cache level. Therefore, it almost costs nothing to access the others elements in the same block of data as `[1,1]`, such as `[1,2]` through `[1,16]`, supposing that our cache line carries blocks of data containing sixteen elements. However, each element is written into a column where adjacent elements are separated in memory by a stride equal to the length of the matrix row causing cache misses at first cache level. Consequently, the way the matrix transposition is performed causes the algorithm to either read from the cache line or write new elements to the cache line.

This chapter introduces several algorithms for Matrix Transposition and is divided into four sections. In Section 4.1, we present the Naïve Algorithm. In Section 4.2, we present the Blocked Transpose Algorithm proposed by Roy et al. [33]. Finally, in Section 4.3 and 4.4, we present the Cache-Oblivious Algorithm and Cache-Aware Algorithm (see Section 2.3) for this problem.

4.1 Naïve Algorithm

The traditional algorithm for Matrix Transposition, called Naïve Algorithm in this document, is, theoretically, the most inefficient algorithm in terms of running time and memory usage. Therefore, it was used as a reference for comparison. This algorithm traverses the matrix in row-major order and another in column-major. Therefore, for large matrices, it, usually, gets a cache miss on every step of the column-wise traversal.

The following algorithm presents the algorithm's pseudo-code, where Out is the output matrix, In is the input matrix and N is the number of elements per column.

Algorithm 2 Naïve Algorithm

```
1: procedure TRANSPOSE
2:     for i ← 1 to N+1 do
3:         for j ← 1 to N+1 do
4:             Out[j][i] ← In[i][j]
```

4.2 Blocked Transpose Algorithm

The Blocked Transpose Algorithm was introduced by Swapnoneel Roy et al. [33] to increase the energy efficiency, according to the model described in Section 3.1.1. This algorithm takes advantage of the cache organisation in P banks to achieve a better performance. As explained in the Subsection 3.1.1, the model assumes that memory is divided into a set of parallel P banks and each bank and it is constituted by blocks, each one formed by B items. Moreover, this algorithm requires the two pre-processing steps as presented in Section 3.1.1. Given a vector of size $N \times N$, representing the matrix, these pre-processing steps create three vectors. The first vector, V, contains the $N \times N$ elements of the matrix. The second vector, M, works as a mapping vector to access the correspondent memory blocks indices. Finally, the third vector, T, sets a page table vector which defines the ordering of the blocks and the location of each stride in memory.

Algorithm 3 illustrates the pseudo-code of this algorithm. It uses vectors M and V, to divide M into $\frac{N \times N}{P \times P}$ submatrices of size $P \times P$. Then, for each generated sub-matrix, the algorithm transposes it naively in place, using arithmetic operations to compute the correct indices. These arithmetic operations consist of shifts, subtractions and additions to compute which block, which stride in the block and which offset within the stride the vector position belongs. After transposing all submatrices in place, each sub-matrix is swapped with the sub-matrix in M that occupies its place, row-by-row, using again arithmetic operations.

Algorithm 3 Blocked Transpose Algorithm

```
1: procedure TRANSPOSE
2:     Divide M into N²/P² submatrices of size P × P denoted as PM
3:     for each P ∈ PM do
4:         Transpose P naively in place (using arithmetic operations)
5:     for each P ∈ PM do
6:         Swap P with the corresponding sub-matrix in M row-by-row
```

4.3 Cache-Oblivious Algorithm

The recursive Cache-Oblivious Algorithm introduced by Frigo [35] and uses the model described in Section 2.3. As mentioned, the model replaces the cache block whose next access is furthest in the future in order to minimise the number of cache misses and achieve better memory performance. The start of this algorithm considers the full matrix, then splits the current matrix into two submatrices and recursively divides the actual matrix into two submatrices until it reaches the base case sub-matrix size, a single element.

Figure 4.1 illustrates how the algorithm transverses the matrix: the algorithm recursively divides the actual matrix until it reaches the sub-matrix with the number 1. Then, it accesses the sub-matrix with the number 2, followed by the sub-matrix with the number 3 and finally the sub-matrix with the number 4. It continues accessing others submatrices and it performs the transposition in the same order as explained.

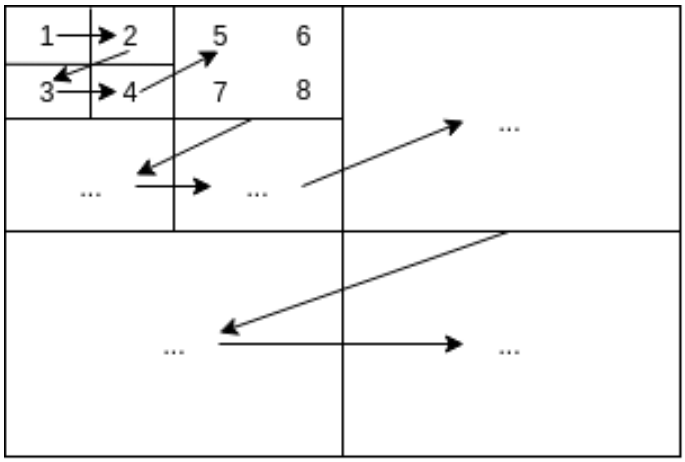


Figure 4.1: An illustration of Cache-Oblivious Algorithm

The following algorithm presents the algorithm's pseudo-code, where *Out* is the output matrix, *In* is the input matrix, N is the number of elements per column, and *imid* and *jmid* are auxiliary variables.

Algorithm 4 Cache-Oblivious Algorithm

```
1:  procedure TRANSPOSE(r_begin, r_end, c_begin, c_end, base_case)
2:      n_rows ← r_end − r_begin
3:      n_cols ← c_end − c_begin
4:      if n_rows ≤ 1 and n_cols ≤ 1 then
5:          Out[j][i] ← In[i][j]
6:      else if n_rows ≥ n_cols then
7:          r_mid ← n_rows/2
8:          TRANSPOSE(r_being, r_being + r_mid, c_begin, c_end)
9:          TRANSPOSE(r_being + r_mid, r_being, c_begin, c_end)
10:     else
11:         c_mid ← n_cols/2
12:         TRANSPOSE(r_being, r_being, c_begin, c_being + c_mid)
13:         TRANSPOSE(r_being, r_being, c_begin + c_mid, c_end)
```

Therefore, when considering the performance of this algorithm it is also important to

consider base cases larger to amortise the overhead of the recursive subroutine calls when considering a base case of 1×1. The following algorithm 5 illustrates the pseudo-code of the algorithm, where *Out* is the output matrix, *In* is the input matrix, N is the number of elements per column, *imid* and *jmid* are auxiliary variables and *base* is the base case sub-matrix size.

Algorithm 5 Cache-Oblivious Algorithm

```
1:  procedure TRANSPOSE(r_begin, r_end, c_begin, c_end, base_case)
2:      n_rows ← r_end − r_begin
3:      n_cols ← c_end − c_begin
4:      if n_rows ≤ base_case and n_cols ≤ base_case then
5:          for i ← 1 to n_rows do
6:              for j ← 1 to n_cols do
7:                  Out[j][i] ← In[i][j]
8:      else if n_rows ≥ n_cols then
9:          r_mid ← n_rows/2
10:         TRANSPOSE(r_being, r_being + r_mid, c_begin, c_end)
11:         TRANSPOSE(r_being + r_mid, r_being, c_begin, c_end)
12:     else
13:         c_mid ← n_cols/2
14:         TRANSPOSE(r_being, r_being, c_begin, c_being + c_mid)
15:         TRANSPOSE(r_being, r_being, c_begin + c_mid, c_end)
```

4.3.1 Cache-Oblivious Parallel Algorithm

The Cache-Oblivious Algorithm is a recursive algorithm, that recursively splits the current matrix into two submatrices until it reaches the base case. Since the recursive function receives as arguments the initial row, last row, initial column, and last column of the actual sub-matrix, it is possible to split the matrix into a given number of submatrices, each of which to be processed by a thread. After assigning each sub-matrix to its correspondent thread, the current matrix is split into two submatrices and, recursively, divides the actual matrix into two submatrices until it reaches the base case, as in Algorithm 4 and 5.

The matrix can be equally partitioned by columns or by rows. If partitioned by columns, given the number of threads, the step between rows is equal to N, where N is the number of rows and columns, while the step between columns in equal to division of N by the number of threads. Note that if N is not a multiple of the number of threads, the last sub-matrix will be smaller. This case is treated by adding one to the step between columns. Moreover, to perform a partition by rows, the step between columns is equal to N, while the step between rows in equal to division of N by the number of threads, having the previous note into account.

4.4 Cache-Aware Algorithm

The Cache-Aware Algorithm explores the model described in the Section 2.3. It is an algorithm that uses the knowledge of the memory architecture to achieve a better performance in terms of cache misses. Therefore, it uses the cache line size or other cache sizes of the processor, L, and divides it by the size of the matrix type. The algorithm is similar to the Naïve Algorithm but it rearranges the data in order to transfer it to the cache in

blocks, similarly to the recursive Cache-Oblivious Algorithm (Section 4.3), to minimise the number of cache misses. Moreover, the L value will be used to create submatrices of size $L \times L$.

For example, in Figure 4.2, the algorithm will divide the matrix into submatrices of size $L \times L$. Then, similar to the Naïve Algorithm, it accesses the elements by rows and columns. The arrows indicate the sequence of blocks accesses by the algorithm.

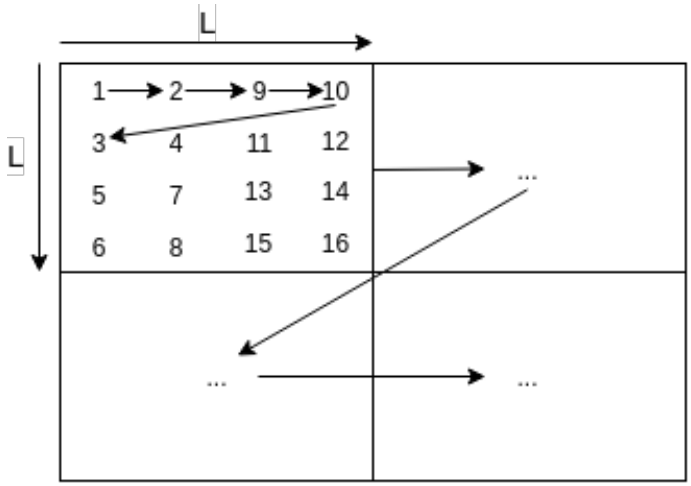


Figure 4.2: An illustration of Cache-Aware Algorithm

Algorithm 6 presents the pseudo-code of the Cache-Aware Algorithm, where Out is the output matrix, In is the input matrix, N is the number of elements per column and L is the number of columns of the sub-matrix.

Algorithm 6 Cache-Aware Algorithm

```
1: procedure TRANSPOSE
2:     for r ← 1 to N by L do
3:         for c ← 1 to N by L do
4:             rlimit ← min(r + L + 1, N)
5:             climit ← min(c + L + 1, N)
6:             for i ← r to rlimit do
7:                 for j ← c to climit do
8:                     Out[j][i] ← In[i][j]
```

4.4.1 Cache-Aware Parallel Algorithm

In order to parallelize the Cache-Aware Algorithm it is necessary to introduce the initial and the last row as well as the initial and the last column (see Algorithm 7). Therefore, similar to the Cache-Oblivious Parallel Algorithm, the function receives as arguments the initial and last row, and initial and last column of the actual sub-matrix. It is possible to split the matrix into a given number of submatrices, each of which to be processed by a thread.

Algorithm 7 Cache-Aware Parallel Algorithm

```
1: procedure TRANSPOSE(r_begin, r_end, c_begin, c_end)
2:     for r ← r_begin to r_end by L do
3:         for c ← c_being to c_end by L do
4:             rlimit ← min(r + L + 1, N)
5:             climit ← min(c + L + 1, N)
6:             for i ← r to rlimit do
7:                 for j ← c to climit do
8:                     Out[j][i] ← In[i][j]
```

Similarly to what has already been explained in the previous Section 4.3.1, the matrix can be equally partitioned by columns or by rows.

4.5 Discussion

As previous mentioned, the Naïve Algorithm will be used as reference for comparison since it is, theoretically, the most inefficient algorithm in our experiments.

The Blocked Transpose Algorithm takes advantage of the memory and cache organisation (see Section 3.1.1) to achieve a better performance. Differently from the Naïve In-place Algorithm, it uses memory locations within nearly close storage locations and the consequent reduced number of cache misses, in different memory levels. Therefore, it is expected that this algorithm performs better than the Naïve In-place Algorithm, both in terms of running time and energy consumption.

The Cache-Oblivious Algorithm takes advantage of the memory access pattern to achieve better performance. Moreover, it uses memory locations within nearly close storage locations and consequently reduced number of cache misses, in different memory levels, and the memory access pattern which allows higher reuse of both rows and columns accessed elements. However, the numerous recursive subroutine calls can penalise the algorithm performance, particularly in the running time, although it is expected that this algorithm performs better than the Naïve Algorithm, both in running time as in energy.

The Cache-Aware Algorithm mainly takes advantage of the contiguous memory accesses to achieve better performance. Similar to the Cache-Oblivious Algorithm, it uses memory locations within nearly close storage locations and consequently reduced number of cache misses, in different memory levels. Another important aspect is that unlike the Cache-Oblivious Algorithm, there is no recursive height and consequent negative impact on the running time. Moreover, since the Cache-Oblivious Algorithm performs the transposition of the submatrices not contiguously in columns, as the Cache-Aware Algorithm, both present different patterns of memory accesses. Although they present similar algorithms structures, they may present different results in terms of energy consumption, run-time and the number of cache misses.

Comparing both Cache-Oblivious and Cache-Aware Algorithms, an algorithm that processes its data by divide-and-conquer often has better cache performance than one that uses iteration. Especially when the sub-problems are small enough to fit in the cache, the number of cache misses are mainly when the sub-problem is loaded for the first time. Therefore, due to the good temporal locality achieved by the algorithm, it is expected that the Cache-Oblivious Algorithm presents a better performance on energy and time consumed and cache performance than the Cache-Aware Algorithm.

Moreover, for both parallel versions of the Cache-Oblivious and Cache-Aware algorithms, it is expected that the paralelized version of this algorithm becomes faster than the non-paralel version. However, relatively to the energy, there is no certainty of what will be the impact with the usage of simultaneous virtual and physical cores. Moreover, due to the memory hierarchy of modern architectures (Section 2.2), particularly due to the shared cache between cores, it is not clear whether the number of cache misses will degrade or improve.

This page is intentionally left blank.

Chapter 5

Methodology and Experimental Setup

In this chapter, we describe the measurement tools chosen for our experiments to analyse the performance of the algorithms described in the previous chapter in terms of time, energy and memory usage, the environment where these experiments were conducted and the experimental procedure adopted.

The experiments were conducted on a computer with a 6th Generation Intel Core CPU, based on the Skylake architecture (x86-64), which has 8 virtual cores, 4 physical cores, running at 2.6GHz, and using Ubuntu 18.04.2 LTS. Furthermore, the computer was equipped with 16 GB of RAM and 3 cache levels, L1, L2 and L3. The L1 cache is divided into instructions and data, L1I and L1D, each one with a size of 32KiB and an 8-way associative placement policy. The L2 cache has a size of 256KiB with a 4-way associative placement policy and a exclusive cache inclusion policy. Finally, the L3 cache has a size of 6144KiB with a 12-way associative placement policy and a inclusive cache inclusion policy. Moreover, every cache levels has a write-back policy and a LRU replacement policy.

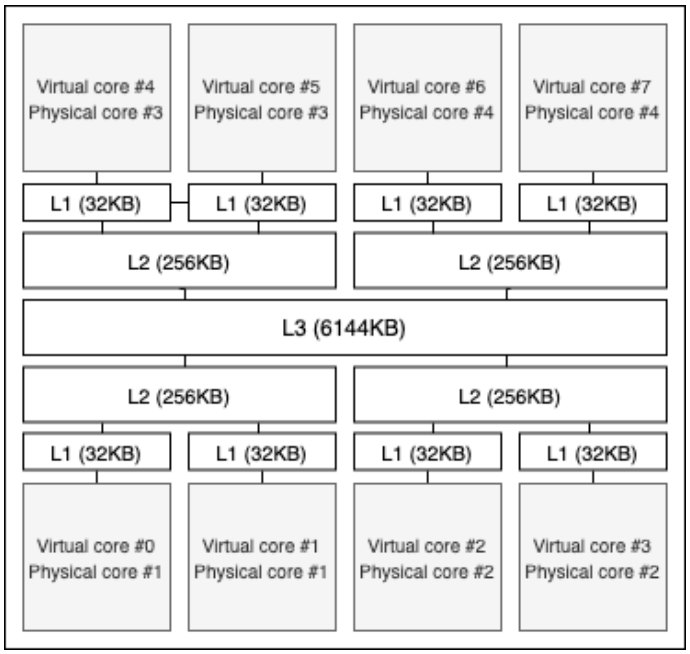


Figure 5.1: Skylake architecture - Memory hierarchy

The algorithms were implemented in C++, in particular, using the C++11 standard, and compiled with *clang++* without and optimisation flags. C++ was chosen over C due to the usage of threads. The C++ Standard Template Library (STL) is only used for the generation and management of the threads in the multithreading and multicore experiments. The remainder of the code, in particular the setup of the matrices and the algorithms for the transposition, is implemented using only the C standard library. Moreover, since *clang++* was been used for intermediate code generation, *clang++* was chosen over G++ to maintain coherency in the generated machine code.

We chose data type `int` to represent each element of the matrix. As for the matrix size, we considered squared matrices of size $N \times N$ where $N \in \{1024, 2048, 3072, \ldots, 40960\}$. Before specifying the set sizes, we performed experiments defining four variables: the minimum number of columns of the matrix, the jump matrix size value, the maximum number of columns of the matrix and matrix type, square or non-square. However, due to the limitations sizes and type of one of the algorithms, we restricted our matrix set sizes and type. Moreover, we evaluated the growth of each generated set with the other sets. Therefore, since the growth showed to be similar on all experiments, we chose the presented set. Furthermore, the Matrix Transposition operation can be performed with In-place algorithms, performing the transposition inside the same matrix. However, to perform it without high complexity algorithms, the matrix needs to be squared. Therefore, since we want to embrace both matrix types, square and not square, to generalise the results, we analysed algorithms implementations that perform the transposition from one input matrix to an output matrix.

The source code for each algorithm share same *main*. Moreover, except of the Blocked Transpose Algorithm (Section 4.2), the algorithms perform the transposition from one input matrix to an output matrix. To achieve a better cache performance (see Section 2.4), we implemented two one-dimensional arrays with $N \times N$ elements with alignment of 64 bytes. This alignment value was chosen after exploring and experimenting multiple values, such as 4096 (cache page size) and 64 (cache line size). Note that due to the cache organisation (Section 2.2), the best alignment values are always multiples of the cache line size.

The Matrix Transposition operation is composed of four steps: matrix initialisation, operations on the matrix, matrix transposition, and finish the application or perform more operations on the matrix. Our experiments focus on matrix initialisation and matrix transposition. For the matrix initialisation, we used `memset` operation to initialise the matrix and to fill the cache levels, since it ensures contiguous memory initialisation.

We used *perf* to collect the performance metrics for each algorithm such as time, energy and memory usage values. Moreover, we analysed the performance of the algorithms without memory accesses instructions. Therefore, to collect the performance metrics of the algorithms without the memory access instructions, we used LLVM and *clang++* to generate a new executable and gather the results. For the multicore and multithreading experiments, we used all eight available virtual cores and a maximum of 1024 threads by virtual core.

Each algorithm was ran fifteen times for each instance to account for slight fluctuations in memory usage performance counters, time and energy values. Note that, there are different types of cache misses, i.e. instructions and data. However, in our experiments we only analyse the data cache misses. To understand more about the collected performance events and the derived values, i.e. number of cache references at the first level cache, we refer to Appendix A. Furthermore, to maintain accuracy and reduce disturbances in the collected values, the program executions were made on a light window manager environment (in

particular, i3) with the screen and networking connections ("flight mode") turned off, and with minimal number of background processes.

In order to reduce the noise and interference of other internal processes, we isolated one physical core, two virtual cores, to run the experiments. Moreover, to prevent processes migrations between CPUs, we confined the program execution in just one of the isolated virtual cores, except the multicore programs, using `taskset` to set the CPU affinity. The command line used to perform this action is below, where C is the core number.

```
taskset -c C perf stat -o output.txt -cpu=C -e "events" ./program
```

Moreover, to ensure that all cache levels are empty, we perform a kernel command to clear page and buffer cache.

```
sudo sh -c "echo 3 > /proc/sys/vm/drop_caches"
```

In the next paragraphs, we discuss the chosen measurements tools for our experiments. We start by investigating measurements tools for measuring energy, time and memory usage performance counters. Then, we discuss simulation tools to validate the obtained results in order of memory usage performance counters values. At last, we discuss some multithreading and multicore tools.

Performance counter measurement tools There are generally two ways to measure energy: using the computer *internal sensors* or using hardware connected to the computer as *external sensors*. *External sensors* usually perform measurements at some predetermined time interval (e.g. every second) and measure the entire energy consumed. Therefore, since the level of granularity we want is too thin, at the instruction level, we have opted to use *internal sensors*. There are a few energy consumption internal sensors, such as PowerTOP [15], Intel Power Gadget [16] and RAPL [17, 18]. However, the first two only just measure the instantaneous or actual energy consumption, while RAPL can measure the energy consumed between intervals or during program execution. Therefore, in the context of our experiments and because we will use an Intel architecture, we decided to use RAPL.

Running Average Power Limit (RAPL) interfaces consist of non-architectural Model-Specific Register (MSR), i.e. control registers that are used for debugging, program execution tracing and computer performance monitoring) and it was implemented by Intel to work in SandyBridge architectures and newer [17, 18]. RAPL does not directly measure the consumed energy by each processor. Instead, it uses a modelling approach based on 100 different micro-architectural event counters. These event counters are after used to model the dynamic energy consumption.

Each RAPL domain supports a set of capabilities. One of them is the *Energy Status*, which provides energy consumption information about two main domains:

- Total package, consisting of the two following domains:
  - PP0 (Core Devices): components of the processors involved in instructions execution.
  - PP1 (Uncore Devices): devices close to the CPU but not part of the core, such as the Graphics Processing Unit (GPU) and other sub-components such as the L3 cache, the integrated memory controller, etc.
- DRAM: a type of RAM memory and the main data component of the processors.

The accuracy and validation of RAPL have been analysed in [17], where the authors show that it is capable of providing accurate energy estimates at a fine-grained level, reaching an average error rate of only 1.12%. A deeper understanding of this interface can be consulted in [69].

On Linux the RAPL energy measurements can be accessed commonly in three different ways: reading files in the *inter-rapl* directory, using the *perf* tool or reading from the MSR. This tool is a standard profiling and performance analyzing tool for the Linux kernel that provides a framework for collecting and analyzing hardware and software performance counters data, including energy and cache memory usage data. It is one of the most commonly used performance counter tools on Linux along with *OProfile* [70], it has a simpler user interface and allows us to access the energy consumption values. Therefore, since we can collect all the necessary data to our analysis using *perf*, we decided to use it as a benchmarking tool.

The *perf* tool is maintained as part of the Linux kernel tree and it is used to access the performance event subsystem in the kernel. The hardware and software performance counters, also called events, available vary based on the specific performance monitoring hardware and processor. To allow gathering these counters data as a normal user we need to change the kernel variable that controls the use of the performance events system by unprivileged users. A deeper understanding of this variable or others can be consulted in [71].

To collect a given set of performance events, we used the *perf stat* command. The *stat* command, given a set of performance events and a command to execute, presents a summary of performance events chosen and saves it into a file. For our experiments, we wanted to collect all the available data about energy and cache memory usage. Therefore, the available events in *perf* (Appendix A for a better comprehension) allows us to gather the energy values for different CPU components, such as RAM, GPU, Cores and Package (energy consumed by the GPU and Cores), memory usage performance counters, such as the number of cache misses and stall cycles for the different levels of the memory hierarchy, and other performance counters such as the number of instructions and cycles. For more information about the performance counters, see [72, 73, 74].

The following command line was used to extract the value of each available event counter used in our experiments for the respective program, where `"events"` represents the events names to gather (Appendix A) and `output.txt` the output file with the events values.

```
perf stat -o output.txt -a -e "events" ./program
```

Moreover, *perf* allows to specify the CPUs to gather the event counters, with flag `-a` to collect from all CPUs and `-cpu=C` from a specific CPU.

Finally, to measure time we experimented and analysed several functions such as `clock`, `getrusage`, `clock_gettime` and `chrono::high_resolution_clock`. After some investigation and analysis, and to maintain accuracy and consistency along with the experiments, we excluded `clock` since we want the elapsed real time of the program, which includes I/O instructions time, and this function can not present an accurate and precise time value including I/O instructions. Moreover, the other functions, `getrusage`, `clock_gettime` and `chrono::high_resolution_clock` presented similar results. However, due to the fact that *perf* uses the same approach as `getrusage` [75], we decided to use also *perf*.

Intermediate machine code generation tool There are different tools to generate intermediate source code representation. However, due to the familiarity and experience

with the library, as well as its popularity, we decided to use the LLVM. LLVM is a collection of libraries and a back-end compiler designed around a language-independent intermediate representation (LLVM IR). Moreover, it can be used to construct, optimise and generate intermediate or binary machine code. To generate the intermediate code representation, we use *clang* as front-end compiler, which is a well known compiler that is fully compliant with C++11 and uses LLVM as its back-end.

The following command line generates the intermediate source code representation into a LLVM IR file, which is a human readable LLVM bitcode representation.

```
clang++-7 program.cpp -S -emit-llvm -o program.ll
```

After identifying and removing the intermediate memory accesses instructions, we compiled the LLVM IR file into assembly language using the `llc` tool from the LLVM library. Then, to create the executable on the generated assembly code, we use the *clang* linker. The following command line shows these two steps.

```
llc-7 program.ll && clang++-7 program.s -o program_without_memory_accesses
```

Simulation tool Since *perf* collects data based on real-time events, by sampling, we explored other frameworks that allow simulating the memory usage performance for a comparison between the simulated and the real-time counters. After some investigation, we concluded that one of the most recognizable and used framework was Valgrind. *Valgrind* is a multipurpose code profiling and memory debugging tool for dynamic analysis [76]. It relies on a technique called dynamic binary instrumentation, where the binary code of a program is re-compiled dynamically as the program is running. This enables the transformation of the binary code to include different analysis tools and methods and consequently heavily reduces the performance. This framework allows us to run our program in an isolated environment with a set of tools each of which performing some kind of debugging or profiling. Since we want to profile the memory usage of our program, the chosen tool was *Cachegrind*, which is a cache and branch-prediction profiler. Similar to the events collected using *perf*, the available simulation events were:

- Instructions (*Ir*): total of instructions executed;
- Data cache reads (*Dr*): total of memory reads;
- Data cache writes (*Dw*): total of memory writes;
- L1 instruction read misses (*Ilmr*): total of cache instruction read misses in L1;
- L1 data cache write misses (*D1mw*): total of cache write misses in L1;
- L1 data cache read misses (*D1mw*): total of cache read misses in L1;
- Last Level Cache (LLC) instruction read misses (*ILmr*): total of cache instruction read misses in LLC;
- LLC data cache write misses (*DLmw*): total of cache write misses in LLC;
- LLC data cache misses (*DLmr*): total of cache read misses in LLC;

For more information about Valgrind and Cachegrind see their documentation in [77]. The following command line gathers the profiling information of each available event for the respective program.

```
valgrind -tool=cachegrind -cache-sim=yes -branch-sim=yes ./program
```

Multithreading and Multicore tool We analysed three multithreading and multicore frameworks: Open Multi-Processing (OpenMP), Intel Threading Building Blocks (TBB) and C++11 Threads. OpenMP is an application program interface that consists of several compiler directives, runtime library routines and environment variables, specified in C, C++ and Fortran [78]. It stands at a high-level abstraction, hiding and implementing by itself the instructions needed to synchronisation, work management and communication between threads. Therefore, a programmer only needs to specify the parallel regions of the application code. TBB is a C++ parallel programming library developed by Intel [79], offering a variety of constructors, ranging from simple parallel loops to flow graphs. It includes a very highly sophisticated runtime scheduler responsible for managing the thread pool and mapping tasks to worker threads. Moreover, similar to OpenMP hides the details and instructions for performance and scalability. Finally, C++11 Threads is a new thread library introduced by C++11, which includes utilities for creating, managing threads and for synchronisation. However, contrary to the previous frameworks, the programmer is responsible for the workload partitioning, threads mapping, worker management and synchronisation.

Comparing the three frameworks, it is notorious that the native threads programming model introduces more complexity within code than OpenMP or TBB. Therefore, it is more challenging to create with the scaling of the number of threads and achieve a good scalable performance. Moreover, the TBB and OpenMP are designed for performance, scalability and very useful when doing intensive work [80], due to the automatically creation and management of the thread pool, such as in thread synchronisation or scheduling. However, OpenMP presents a better performance when the algorithms are dominated by contiguous memory accesses. Therefore, since our benchmark problem is mainly working with memory accesses and OpenMP is better suited to data-parallel problems, we decided to choose OpenMP over TBB.

To validate the OpenMP framework to our experiments, we performed some preliminary experiments using the same methodology as previously explained. The results showed when dealing with multithreading parallelism within recursion functions and algorithms mainly dependent on recursion, it exhibits a high number of threads created, sometimes exceeding the number of elements to be processed. Therefore, these threads overhead make the algorithm run much slower than the non-parallel version of the algorithm and the higher number of instructions induced by the libraries ruins the shared memory optimisation of some algorithms. Moreover, when dealing with multicore parallelism within recursion functions, we can notice a slight speedup. However, this speedup is small when compared with the speedup using C++11 Threads. Therefore, since one of the analysed algorithms is a divide-and-conquer algorithm (see Section 4.3), mainly dependent on recursion calls, and our benchmark problem does not require high complexity code development to achieve parallelism in all algorithms, we decided to use C++11 Threads. C++11 Threads library utilities allow to create, manage and synchronise threads. In our experiments, to perform these operations we used three standard libraries classes, `std::vector` to create the pool of threads, `std::thread` constructor to create a thread given the arguments to the thread sub-problem and `std::thread::join` to initiate and block the current thread until the thread identified finishes its execution. Moreover, since to divide the problem among the initiated threads, we calculate the size of each sub-problem given the number of cores, threads or threads per core to execute. Finally, to bind threads to a specific CPU we used `taskset`.

In the following Chapter 6, we analyse the results of the various algorithms (see Chapter 4) that solve the matrix transposition problem using different memory access patterns. Our aim is to understand what is the impact of the application of these algorithms in terms of

energy, time and memory usage. Therefore, to compare these algorithms, we analysed their performance in terms of time, energy and memory usage. In these experiments, energy, time and cache memory usage will be the only aspects of the execution to compare, since the quality of the solutions for all the algorithms will be the same and were checked before they were performed.

This page is intentionally left blank.

Chapter 6

Experimental Analysis

This chapter describes an in-depth experimental analysis conducted with the goal of answering the four research questions that were described in Section 1. In this experimental analysis, we take into account the Energy Complexity Model described in Section 3.1.1 and we use the experimental methodology described in Chapter 5. We recall that the Energy Complexity Model suggests that the total energy consumed by an algorithm, A, can be modelled as a linear combination of the energy consumed by the CPU instructions or non-memory accesses, A_{CPU}, and memory accesses A_{Memory}, i.e.

$$E(A) = E(A_{CPU}) + E(A_{Memory}) \tag{6.1}$$

This chapter is divided into four sections, each of which corresponds to a particular research question.

6.1 Research Question 1

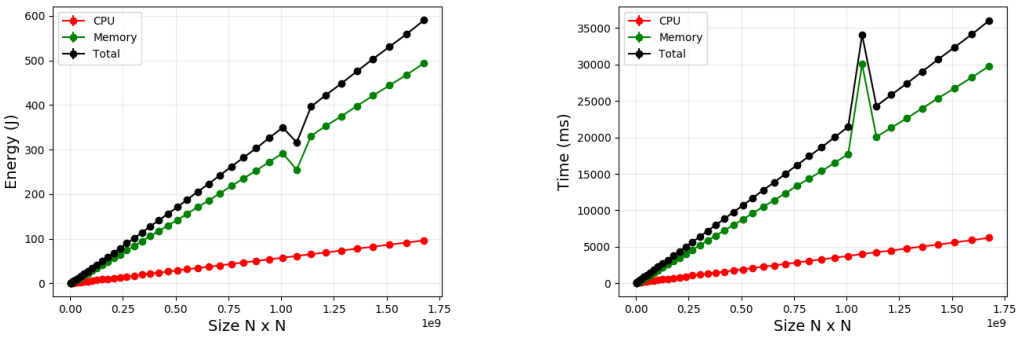
RQ1 *According to the Energy Complexity model proposed by Roy et al. [33], the total energy consumed by an algorithm can be modelled by the energy consumed by performing CPU instructions and memory accesses. According to this model, what is the energy consumption of the traditional Matrix Transposition algorithm?*

The traditional algorithm for Matrix Transposition, called Naïve Algorithm in this document (see Section 4.1), is, theoretically, the most inefficient algorithm in terms of running time and memory usage that we considered in our experiments.

In order to answer the previous research question, we measured the total amounts of running time and energy consumed by CPU instructions and memory accesses of an algorithm and divided them into two components: the CPU instructions component and the memory accesses component. To accomplish this, we performed two distinct measurements. The first measurement covered the overall energy and time consumption, including both CPU instructions and memory accesses. The second measurement covered the energy and time consumption of only the CPU instructions. Since it is impossible to measure exclusively the time and energy consumed by the memory accesses, we analysed the LLVM machine code generated by the compiler and removed the memory accesses operations, load and store, referring to the matrix transposition instructions. Therefore, by removing these instructions, we are able to characterise the performance of the algorithm without accessing and

storing on memory. The time and energy consumed by the memory accesses correspond to the subtraction of the values collected in the first measurement with the second measurement. Furthermore, it is expected that the running time and energy consumed by the algorithm are strongly (positively) correlated, since both the number of CPU instructions and memory accesses affect time.

Figure 6.1 shows the performance of the Naïve Algorithm in terms of Energy (a) and Time (b) with respect to CPU instructions (green line) and memory access (red line) and total (black line).



(a) Energy (b) Time

Figure 6.1: Performance of Naïve Algorithm in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses

Both plots of Figure 6.1 suggest that the number of memory accesses has a strong influence on both time and energy. Moreover, from Figure 6.1b, we observe a peak and in Figure 6.1a a valley on the memory part at $N = 32768$. We performed a linear regression (without the outlier at $N = 32768$), where N^2 is the number of matrix elements, and obtained the following models:

$$E_{CPU} = 0.04367 + 5.718e^{-08}N^2 \ (R^2 = 1) \tag{6.2a}$$

$$T_{CPU} = 80.4 + 364.6e^{-08}N^2 \ (R^2 = 1) \tag{6.2b}$$

$$E_{Memory} = -3.586 + 29.39e^{-08}N^2 \ (R^2 = 0.9998) \tag{6.2c}$$

$$T_{Memory} = -114.7 + 1771e^{-08}N^2 \ (R^2 = 0.9999) \tag{6.2d}$$

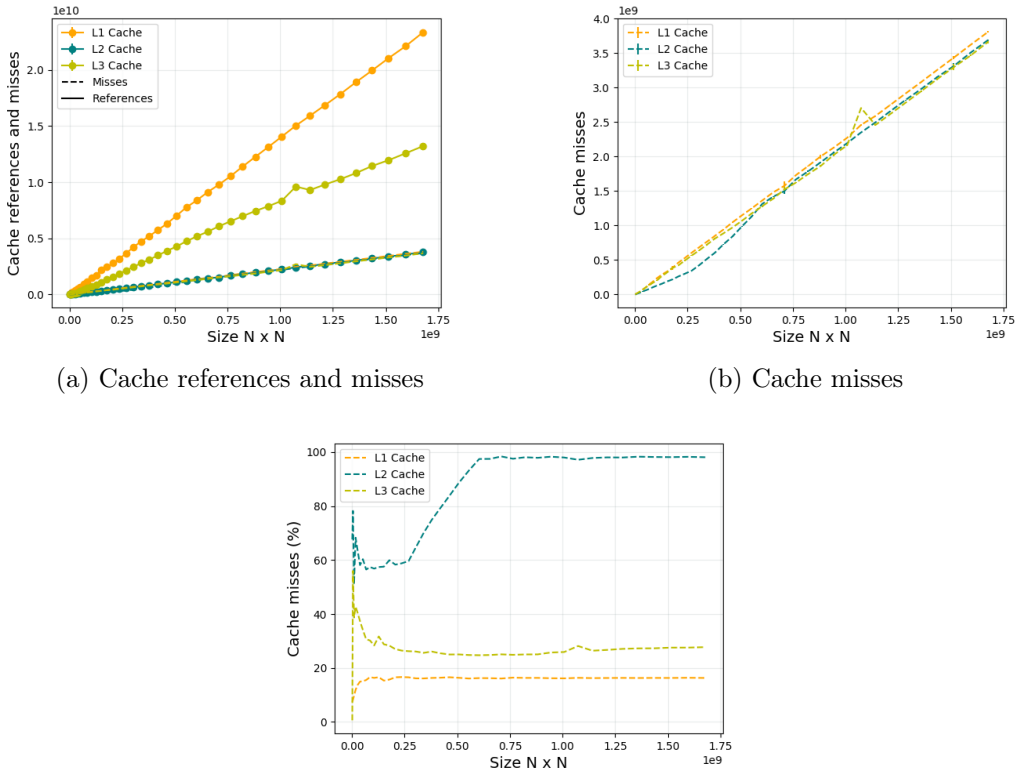
The above models are divided into two factors, energy and time. The energy models in expressions 6.2a and 6.2c, correspond to the CPU instructions and memory accesses, respectively. The time models in expressions 6.2b and 6.2d, correspond to the CPU instructions and memory accesses, respectively. Since R^2 values are close to 1, we conclude that there is a linear relationship between the independent variable, N^2, and the dependent variables, E_{CPU}, E_{Memory}, T_{CPU} and T_{Memory}.

The resource usage ratio of the memory accesses component to the CPU instructions component corresponds to the fraction E_{Memory}/E_{CPU} for energy, and T_{Memory}/T_{CPU} for time. To achieve these value we used the equation models values, ignoring the constant variable since it is insignificant for the experimented values of N. Therefore, from the

previous models, the resource usage ratio of the memory access component to the CPU instructions component shows that the memory accesses component is approximately 5 times greater than the CPU instructions in terms of both energy and time. From this, we conclude that memory accesses have a strong weight on the time and energy performance of the Naïve Algorithm.

The dominance of the memory accesses, and the peak on time and the valley on energy, suggest a worse performance, potentially because of cache misses (see Section 2.2). In the following, we analyse the number of cache references and misses of the Naïve Algorithm at each cache level.

Figure 6.2 shows the number of cache references and misses (a), the number of cache misses (b) and cache misses percentage (c) at L1 (orange), L2 (blue-green) and L3 (dark yellow) cache. Note that the number of cache misses in Figure 6.2a are overlapped with the number of cache references at L2 cache.



(a) Cache references and misses

(b) Cache misses

(c) Cache misses percentage

Figure 6.2: Naïve Algorithm - Cache references and misses (a), cache misses (b) and cache misses percentage (c) at different cache levels

Figure 6.2 shows a large number of cache references and misses. Although we expected this algorithm to have bad cache performance, it presents a low percentage of cache misses, from 7% to 16% at the L1 cache. At the L2 cache level, both lines of references and misses almost overlap, presenting a percentage of cache misses from 60% to almost 100%. Finally, contrarily to the other levels, the L3 cache presents a descending percentage of cache misses from 55% to 28% with the increase of N.

We notice another peak at $N = 32768$ on the L3 cache references and misses. The peak in Figure 6.2, the energy decrease in Figure 6.1a and the time peak in Figure 6.1b at

$N = 32768$, suggest that something is increasing the computation time while reducing the energy consumption for this particular value of N. Since during a cache miss state the processor can remain idle until the data is fetched, causing computation delays, we decided to analyse the number of stalled cycles, or idle cycles, for each cache level.

Figure 6.3 shows the number of cache misses stall cycles (see Section 2.2) at different cache levels, L1 (orange), L2 (blue-green) and L3 (dark yellow). It shows that the value of stalled cycles is substantially higher at $N = 32768$ in all cache levels.

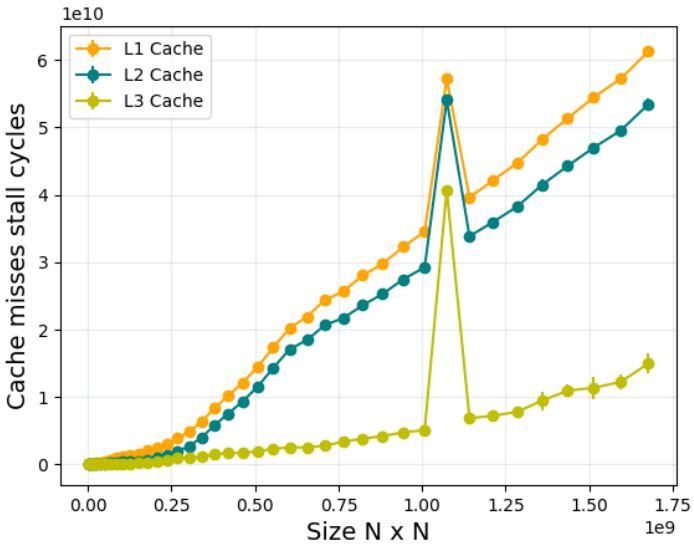
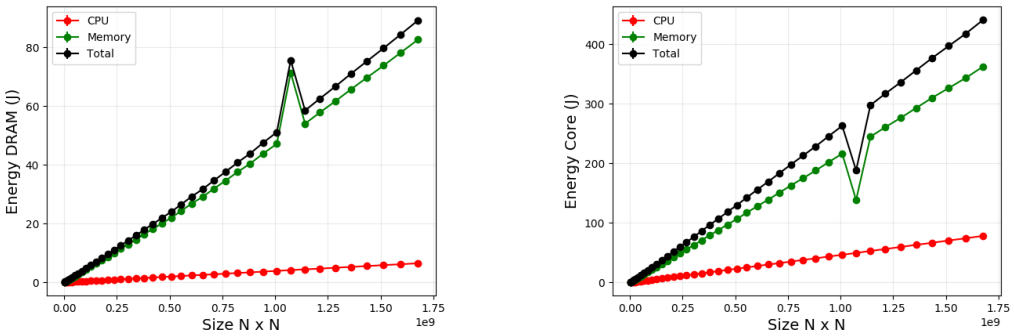


Figure 6.3: Naïve Algorithm - Cache misses stall cycles at different cache levels

Previous results in Figure 6.2a showed that there is a significant difference between the energy consumed by the CPU instructions and the memory accesses. Each processor consists of different components with specific work specifications, e.g. the Core is responsible for the instructions execution while the DRAM for the memory accesses. Therefore, we analysed energy consumed by two different CPU components, Core and DRAM.

Figure 6.4 shows the energy consumed at the DRAM (a) and at the Core (b), by CPU instructions (red) and by memory accesses (green) as well as the total (black).



(a) Energy DRAM

(b) Energy Core

Figure 6.4: Naïve Algorithm - Energy consumed by the DRAM (a) and Core (b) components

As expected, different components have different energy consumption behaviours. In Figure 6.4b we can see that the energy consumption of the Core component presents a fall at $N = 32768$, while in Figure 6.4a an increase is present on the energy of the DRAM component. Moreover, in Figure 6.1a there is a valley instead of a peak because the decrease in energy on the Core component is greater than the increase in energy of the DRAM component. Furthermore, the high value of the energy consumed by the DRAM and the overall running time by the algorithm can be a particular behaviour at $N = 32768$.

Algorithms that access multidimensional arrays sequentially can exhibit poor performance due to poor cache-line utilisation, especially when N is a power of two, due to cache-line conflicts [81]. For example, the cache placement policy can influence the cache behaviour and conflicts in these special cases (see Section 2.2). In our experiments and in the matrix transposition problem, this can cause several delays because of the distance between the memory address accessed, discontiguous jumps of memory by N, also called of cache contentions. However, the value of N seems to be dependent of the computer architecture and the CPU cache, and its restrictions, such as cache policies (see Section 2.2). Moreover, to validate this explanation we performed this experiment in computers with different architectures and noticed that this behaviour happened in other power of two values, e.g. $N = 16384$ and $N = 40960$.

Previous results in Figure 6.1 point that time and energy seem to be strongly correlated and show a dominance of the memory accesses over the CPU instructions on energy consumed and running time. Moreover, Figure 6.2a shows that there is a large number of cache references and misses and Figure 6.4a shows an high energy consumption of the DRAM component. Therefore, the results suggest that energy consumption can be reduced by reducing the number of instructions and/or the number of memory accesses. However, this can not be performed significantly on matrix transposition. One possibility, which will be explored in the following sections, is to exploit memory architecture in a clever way, as suggested by Roy et al. [33].

6.2 Research Question 2

RQ2 *According to the model proposed by Roy et al. [33] it is possible to improve energy consumption by taking advantage of how main memory is organised and accessed. They have proposed an algorithm based on parallel memory models, namely, the Blocked Transpose Algorithm (see Section 4.2). How does this algorithm behave in practice?*

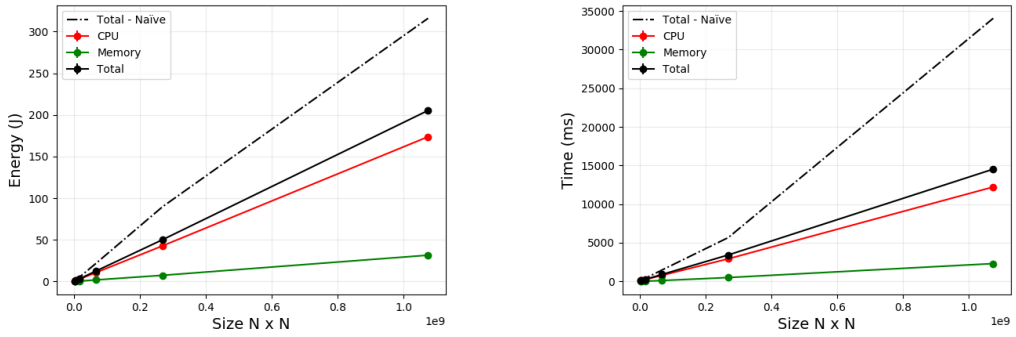
In order to validate this model, the authors developed an algorithm based on the presented memory model, the Blocked Transpose Algorithm (see Section 4.2). This algorithm divides a matrix A (of size $N \times N$) into sub-matrices of size $s \times s$ and uses multiple bitwise operations to calculate the corresponding indices of each matrix element according to the memory model structure. To simplify the procedure, Roy et al. implemented their algorithm for the case where N is power of two, as well as the algorithm dependent variables. Therefore, for this algorithm, we have considered the values of $N \in \{1024, 2048, 4096, 8192, 16384, 32768\}$.

The memory architecture of this model is divided into a set of parallel P banks, each having its own cache. Each bank is also constituted by blocks, each one formed by B items and $\frac{B}{s}$ strides, where s is the number of items in each stride. Moreover, each stride behaves as a cache line. We recall that this algorithm also requires the definition of the number of banks, P, the number of items in each block, B, and the number of items in each stride, s (see Section 3.1.1).

To define each value we tested all combinations that were defined for each parameter with $P \in \{1, 2, 4, 8\}$, $B \in \{16, 32, \ldots, \frac{N}{8}, \frac{N}{4}, \frac{N}{2}\}$ and $s \in \{8, 16, \ldots, \frac{B}{8}, \frac{B}{4}, \frac{B}{2}\}$. Therefore, for each value of P between 1 and 8, we ran experiments on the algorithm considering the value of N between 1024 and 32768. Then, for each value of N, we also considered values of B between 16 and $\frac{N}{2}$. Finally, for each value of B, we considered values of s between 8 and $\frac{B}{2}$. Some preliminary experiments suggested that, independently of the s and P values, the algorithm performance in terms of energy consumed and running time was normally higher when the value of B was not half of N. Therefore, we defined B to be half of N and tested various values of s and P. After further analysis of the results, we concluded that the ideal value of s for all P values and N sizes was with $s = 16$ for the particular machine's cache line size in which the algorithm was tested. We also observed that better performance was obtained increasing the P values. Therefore, for our experiments we defined the value of P as 8, B as $\frac{N}{2}$ and s as 16.

Similar to the previous section, we measured both running time and energy consumed by CPU instructions and memory accesses of the Blocked Transpose Algorithm. Moreover, the increase in the number of CPU instructions and memory accesses affects time and energy consumed in a linear increasing way. Therefore, it is expected that running time and energy consumed by the algorithm are strongly (positively) correlated.

Figure 6.5 shows the performance of Blocked Transpose Algorithm in terms of Energy (a) and Time (b) with respect to CPU instructions (green line) and memory access (red line) and total (black line). In addition, dashed lines present the total energy consumed and running time by the Naïve Algorithm.



(a) Energy (b) Time

Figure 6.5: Performance of Blocked Transpose Algorithm with respect to Energy (a) and Time (b) in terms of CPU instructions and memory accesses

Both plots of Figure 6.5 suggest that this algorithm is able to consume less time and energy than Naive algorithm. Moreover, the CPU instructions have a stronger influence on time and energy, contrarily to the case of the Naïve algorithm. Furthermore, we performed a linear regression, where N^2 is the number of matrix elements, and obtained the following models:

$$E_{CPU} = 0.003316 + 16.17e^{-08}N^2 \ (R^2 = 1) \tag{6.3a}$$

$$T_{CPU} = 33.43 + 1132e^{-08}N^2 \ (R^2 = 0.9997) \tag{6.3b}$$

$$E_{Memory} = -0.1351 + 2.941e^{-08}N^2 \ (R^2 = 0.9997) \tag{6.3c}$$

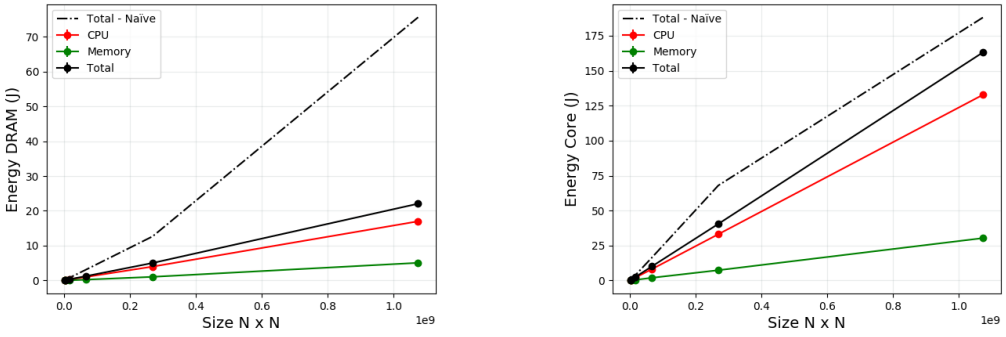
$$T_{Memory} = -24.90 + 215.8e^{-08}N^2 \ (R^2 = 0.9987) \tag{6.3d}$$

The above models are divided into two factors, energy and time. The energy models in expressions 6.3a and 6.3c, correspond to the CPU instructions and memory accesses, respectively. The time models in expressions 6.3b and 6.3d, correspond to the CPU instructions and memory accesses, respectively. Since R^2 values are close to 1, we conclude that there is a linear relationship between the independent variable, N^2, and the dependent variables, E_{CPU}, E_{Memory}, T_{CPU} and T_{Memory}.

From the previous models, the resource usage ratio of the CPU instructions component to the memory accesses component increases for both energy and time and it shows that CPU instructions component is approximately 5.3 greater than the memory accesses component in terms of both energy and time. In comparison with the Naïve Algorithm models (see Equations 6.2), it increases in such a way that the CPU instructions component is now greater that the memory accesses component, whereas the opposite was true for the Naïve algorithm.

By comparing Figures 6.5a and 6.5b, we can observe that the resource usage ratio of the CPU instructions to the memory accesses in terms of energy consumed by the Blocked Transpose Algorithm and the Naïve Algorithm is lower than the resource usage ratio of the CPU instructions to the memory accesses in terms of running time. Therefore, to understand this low discrepancy on energy consumed, we analysed the energy consumed by the DRAM and Core components.

Figure 6.6 shows the energy consumed at the DRAM (a) and at the Core (b), by CPU instructions (red) and by memory accesses (green) as well as the total (black).



(a) Energy DRAM (b) Energy Core

Figure 6.6: Blocked Transpose Algorithm - Energy consumed by the DRAM (a) and Core (b) components

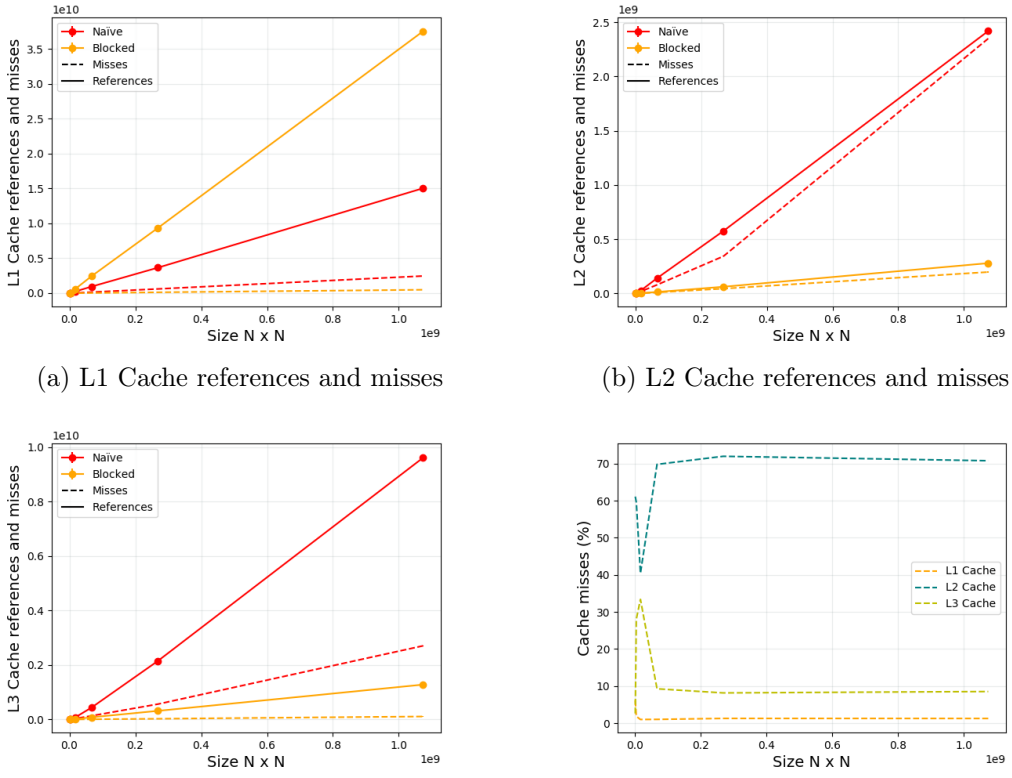
Figure 6.6 indicates that the total amount of energy spent in the Core component by Blacked Transpose algorithm is close to the value obtained with the Naïve Algorithm.

However, there is a large difference between the two in the DRAM component.

The Blocked Transpose Algorithm was design to optimise the number of memory accesses under a specific memory organisation (see Section 3.1.1). Therefore, it was expected that the algorithm presented a lower energy consumption by the DRAM component than the Naïve Algorithm. However, by analysing the energy consumed by the Core component, and despite the difference in the amount of running time, we noticed that the algorithm had a higher energy consumption. The source code of the algorithm handles a large number of bitwise operations to acquire the matrix elements positions according to the memory model structure. Therefore, since the Core component is responsible for the performance of these instructions, this could be a possible explanation for the obtained results.

Contrarily to what as observed in Section 6.1, both Figures 6.5a and 6.5b do not present the same behaviour of $N = 32768$. Therefore, this suggests us that the algorithm inhibit the cache-line conflicts and improved the cache performance. Therefore, to understand the cache performance we analysed the number of cache references and misses.

The plots in Figure 6.7 show the number of cache references and misses at different cache levels for both algorithms, in particular, at L1 cache (a), at L2 cache (b) and at L3 cache (c). Moreover, it shows the percentage of cache misses of the Blocked Transpose Algorithm at different cache levels (d), L1 (orange), L2 (blue-green) and L3 (dark yellow).



(a) L1 Cache references and misses

(b) L2 Cache references and misses

(c) L3 Cache references and misses

(d) Cache misses percentage

Figure 6.7: Cache references and misses, and cache misses percentage at different cache levels for the Blocked Transpose Algorithm and the Naïve Algorithm

According to Figure 6.7, and in comparison with Naïve Algorithm, the Blocked Transpose Algorithm presents a high number of cache references on the L1 cache, but a lower number of cache misses with a percentage of cache misses around 2%. Moreover, on the other

two cache levels, it presents lower values of cache references and misses. However, the L2 cache displays a percentage of cache misses from 60% to 70%, while the L3 cache displays a percentage around 9%.

The larger number of cache references and the lowest percentage of cache misses at the L1 cache indicates a better memory usage by the algorithm at this level. Since the L1 cache is quite explored, one of the L2 and L3 caches, generally the L2, may reveal a worse usage. For example, when a cache miss occurs at the L1 and L2 caches, the data is fetched from the L3 cache or from RAM. Because of the cache policies, such as cache inclusion policy, and the majority of blocks transfer between the L2 and L3 caches, the L2 cache may work mainly as a passage of the these two cache levels. Moreover, since the L1 cache presents a good performance, the other cache levels are not so referenced. Therefore, we can conclude that the algorithm achieved a cache efficient performance, although it is very difficult to achieve a better memory usage in every cache levels, specially in highest cache levels.

As a conclusion, the Blocked Transpose Algorithm results indicate that the way the algorithm takes advantage of memory organisation can have an impact on the energy and time consumption. Moreover, the cache performance results suggests that improving the cache usage and memory accesses patterns can also improve energy and time efficiency.

6.3 Research Question 3

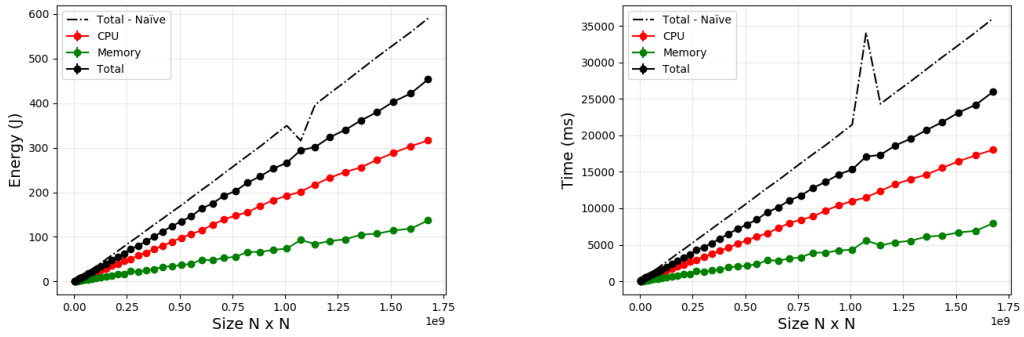
RQ3 *The results in Section 6.2 suggest that cache usage and memory access patterns affect energy (and time). Therefore, can we improve the access to low-level caches, reducing the number of cache misses and inducing the reuse of data present in the cache levels?*

In order to reduce the number of cache misses and reuse the data present in the cache levels on different memory access patterns, we analysed two different algorithms: Cache-Oblivious Algorithm and Cache-Aware Algorithm (see Section 4.3 and 4.4). Moreover, in Section 6.3.3, we shortly summarise the results on both algorithms.

6.3.1 Cache-Oblivious Algorithm

The Cache-Oblivious Algorithm performs the matrix transposition using a divide-and-conquer approach, dividing the matrix recursively until reaching a base case of 1×1. Therefore, contrarily to the Blocked Transpose Algorithm, it uses a different memory access pattern. Similarly to the two previous sections, we measured both running time and energy consumed by CPU instructions and memory accesses of the algorithm.

Figure 6.8 shows the performance of Cache-Oblivious Algorithm in terms of Energy (a) and Time (b) with respect to CPU instructions (green line), memory access (red line) and total (black line). In addition, dashed lines present the total energy consumed and running time by the Naïve Algorithm.



(a) Energy (b) Time

Figure 6.8: Performance of Cache-Oblivious Algorithm in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses

Both plots of Figure 6.8 suggest that the number of CPU instructions has a strong influence on both time and energy and that the increase in the number of CPU instructions and memory accesses affects time and energy consumed in a linear increasing way. Moreover, we performed a linear regression, where N^2 is the number of matrix elements, and obtained the following models:

$$E_{CPU} = 0.6628 + 18.99e^{-08}N^2 \ (R^2 = 0.9998) \tag{6.4a}$$

$$T_{CPU} = 111.6 + 1077e^{-08}N^2 \ (R^2 = 0.9998) \tag{6.4b}$$

$$E_{Memory} = -0.5376 + 7.684e^{-08}N^2 \ (R^2 = 0.9948) \tag{6.4c}$$

$$T_{Memory} = -21.79 + 449.6e^{-08}N^2 \ (R^2 = 0.9941) \tag{6.4d}$$

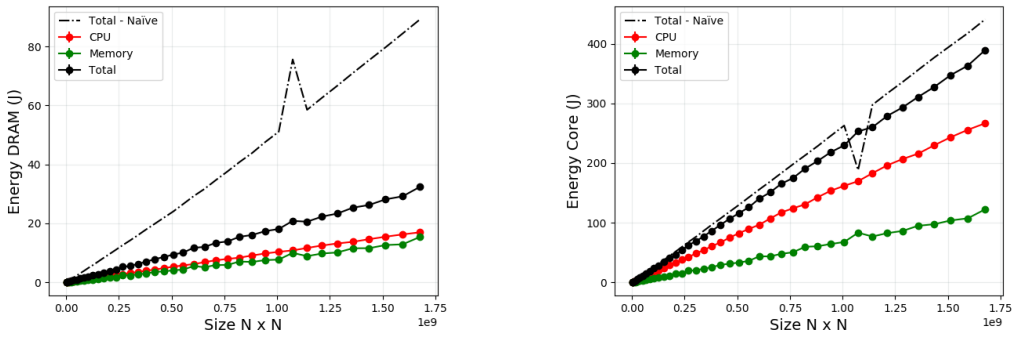
The above models are divided into two factors, energy and time. The energy models in expressions 6.4a and 6.4c, correspond to the CPU instructions and memory accesses, respectively. The time models in expressions 6.4b and 6.4d, correspond to the CPU instructions and memory accesses, respectively. Since R^2 values are close to 1, we conclude that there is a linear relationship between the independent variable, N^2, and the dependent variables, E_{CPU}, E_{Memory}, T_{CPU} and T_{Memory}.

From the previous models, the resource usage ratio of the CPU instructions component to the memory accesses component increases for both energy and time and it shows that CPU instructions component is approximately 2.5 greater than the memory accesses component in terms of both energy and time. From this, we conclude that CPU instructions have a strong weight on the time and energy performance of the Cache-Oblivious Algorithm. In comparison with the Naïve Algorithm models (see Equations 6.2), it increases in such a way that the CPU instructions component is now greater than the memory accesses component, whereas the opposite was true for the Naïve algorithm. Moreover, the resource usage ratio decreased to about half on both energy and time. In comparison with the Blocked Transpose Algorithm models (see Equations 6.3), the dominant component remained the memory accesses component but the resource usage ratio between on both energy and time decrease to less than half.

In comparison with the Naïve Algorithm (see Figure 6.1), the Cache-Oblivious Algorithm

presents better results. However, Figure 6.8a showed that there is a significant difference between the energy consumed by the CPU instructions and the memory accesses. Furthermore, since this algorithm is recursive, the large number of recursive calls can also have an impact on energy consumption. Therefore, we analysed energy consumed by two different CPU components, Core and DRAM.

Figure 6.9 shows the energy consumed at the DRAM (a) and at the Core (b), by CPU instructions (red) and by memory accesses (green) as well as the total (black).



(a) Energy DRAM

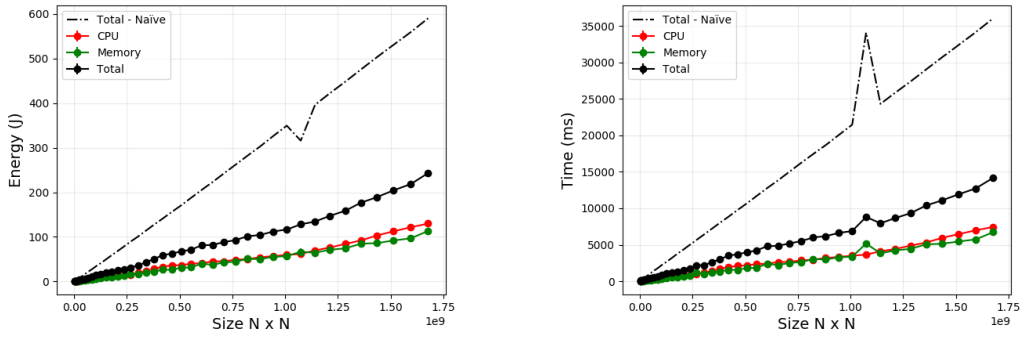
(b) Energy Core

Figure 6.9: Cache-Oblivious Algorithm - Energy consumed by the DRAM (a) and Core (b) components

From Figure 6.9a, we can observe that the DRAM total energy consumed by the Naïve Algorithm is much higher than the energy consumed by the Cache-Oblivious Algorithm. However, from Figure 6.9b, we can observe that the Core total energy consumed by the Naïve Algorithm is almost equal to the energy consumed by the Cache-Oblivious Algorithm. Moreover, in Section 6.1 we observed an unusual behaviour related with cache conflicts in a particular N value. In both Figures 6.8a and 6.8b we can also notice this behaviour at $N = 32768$, although with a just tiny peak in the number of memory accesses.

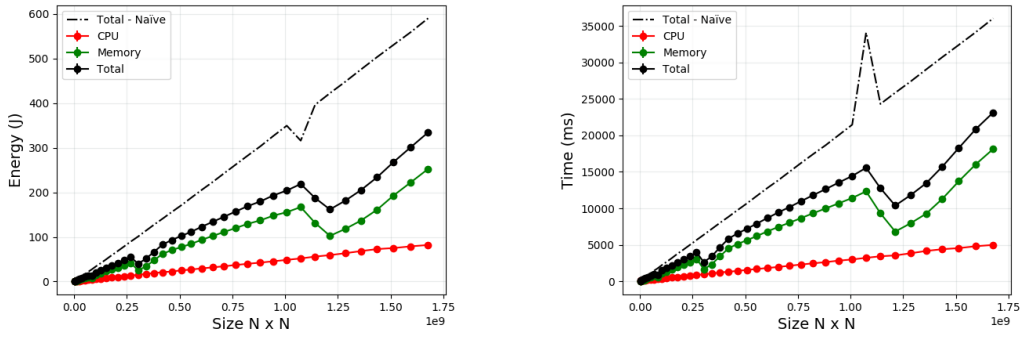
In principle, the Cache-Oblivious Algorithm continues dividing until the base case of 1×1. However, to reduce the impact of a large number of recursive subroutine calls, the size of the base case block can be larger. Therefore, we ran experiments to understand the impact of larger base cases, such as 4×4, 16×16, 64×64 and 256×256.

Figure 6.10, 6.11, 6.12 and 6.13, show the performance of Cache-Oblivious Algorithm in terms of Energy (a) and Time (b) with respect to CPU instructions (green line) and memory access (red line) and total (black line), for base cases 4×4, 16×16, 64×64 and 256×256, respectively. Moreover, for each base case we performed a linear regression, where N^2 is the number of matrix elements.



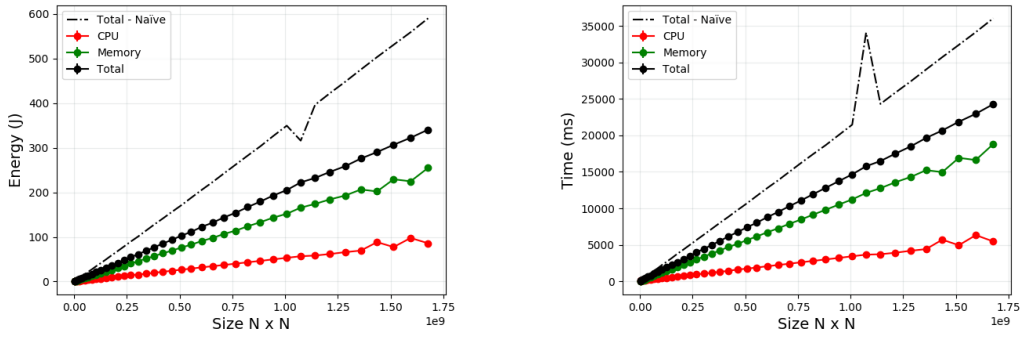
(a) Energy (b) Time

Figure 6.10: Performance of Cache-Oblivious Algorithm with a base case of 4×4 in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



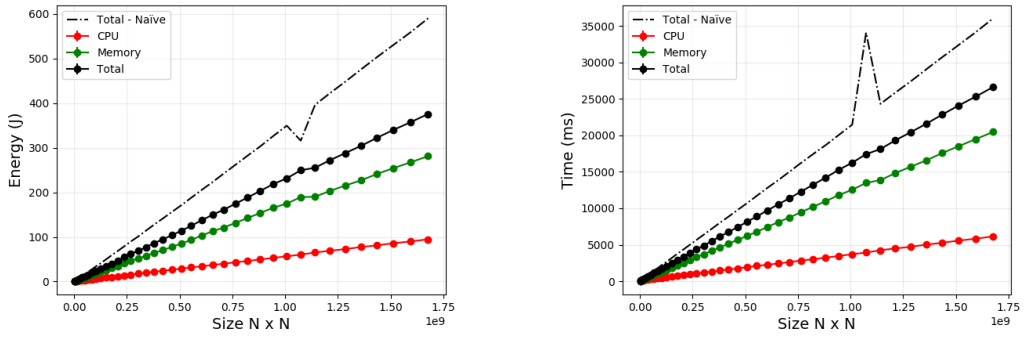
(a) Energy (b) Time

Figure 6.11: Performance of Cache-Oblivious Algorithm with a base case of 16×16 in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

Figure 6.12: Performance of Cache-Oblivious Algorithm with a base case of 64×64 in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

Figure 6.13: Performance of Cache-Oblivious Algorithm with a base case of 256×256 in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses

Linear regression models of Cache-Oblivious Algorithm with a base case of 4×4:

$$E_{CPU} = 1.077 + 6.957e^{-08}N^2 \ (R^2 = 0.9792) \tag{6.5a}$$

$$T_{CPU} = 19.93 + 396.6e^{-08}N^2 \ (R^2 = 0.9813) \tag{6.5b}$$

$$E_{Memory} = -0.7895 + 6.129e^{-08}N^2 \ (R^2 = 0.9938) \tag{6.5c}$$

$$T_{Memory} = -43.95 + 370.3e^{-08}N^2 \ (R^2 = 0.9821) \tag{6.5d}$$

Linear regression models of Cache-Oblivious Algorithm with a base case of 16×16:

$$E_{CPU} = 0.04309 + 4.913e^{-08}N^2 \ (R^2 = 0.9994) \tag{6.6a}$$

$$T_{CPU} = 80.95 + 294.6e^{-08}N^2 \ (R^2 = 0.9995) \tag{6.6b}$$

$$E_{Memory} = 4.52 + 12.64e^{-08}N^2 \ (R^2 = 0.9196) \tag{6.6c}$$

$$T_{Memory} = 383 + 896.1e^{-08}N^2 \ (R^2 = 0.899) \tag{6.6d}$$

Linear regression models of Cache-Oblivious Algorithm with a base case of 64×64:

$$E_{CPU} = -0.3831 + 5.390e^{-08}N^2 \ (R^2 = 0.988) \tag{6.7a}$$

$$T_{CPU} = 48.46 + 340.4e^{-08}N^2 \ (R^2 = 0.9854) \tag{6.7b}$$

$$E_{Memory} = 0.3753 + 14.89e^{-08}N^2 \ (R^2 = 0.9981) \tag{6.7c}$$

$$T_{Memory} = 39.44 + 1098e^{-08}N^2 \ (R^2 = 0.899) \tag{6.7d}$$

Linear regression models of Cache-Oblivious Algorithm with a base case of 256×256:

$$E_{CPU} = 0.2072 + 5.634e^{-08}N^2 \ (R^2 = 0.9999) \tag{6.8a}$$

$$T_{CPU} = 86.03 + 360.9e^{-08}N^2 \ (R^2 = 0.9999) \tag{6.8b}$$

$$E_{Memory} = 0.5586 + 16.87e^{-08}N^2 \ (R^2 = 0.9993) \tag{6.8c}$$

$$T_{Memory} = 13.85 + 1226e^{-08}N^2 \ (R^2 = 0.9998) \tag{6.8d}$$

Figure 6.10 presents a similar performance of both memory accesses and CPU instructions component. Therefore, there is no notorious dominance of one of the components over the other. Moreover, for increasing base case, we observe that the memory accesses component has a larger influence on time and energy than CPU instructions component.

Analysing the models presented above, since R^2 values are close to 1, we conclude that there is a linear relationship between the independent variable, N^2, and the dependent variables, E_{CPU}, E_{Memory}, T_{CPU} and T_{Memory}.

From the previous models, with the increase of the base case, the resource usage ratio of the CPU instructions component to the memory accesses component increases for both energy and time. Moreover, there is also a shift in the dominant component at the base case 16×16. Considering the base case of 4×4, the CPU instructions component is approximately 1.13 times greater on energy and 1.07 times greater on time than the memory accesses component. Then, considering the base case of 256×256, the memory accesses component is approximately 2.99 times greater on energy and 3.39 times greater on time than the CPU instructions component. The other resource usage ratios are presented in Table 6.1. From this, we conclude that memory accesses have a strong weight on the time and energy performance of the Cache-Oblivious Algorithm with the increase of the base case.

Base case	Dominant component	$\frac{E_{DominantComponent}}{E_{SubmissiveComponent}}$	$\frac{T_{DominantComponent}}{T_{SubmissiveComponent}}$
1×1	CPU instructions	2.5	2.4
4×4	CPU instructions	1.13	1.07
16×16	Memory accesses	2.57	3.04
64×64	Memory accesses	2.76	3.23
256×256	Memory accesses	2.99	3.39

Table 6.1: Cache-Oblivious Algorithm - Resource usage ratio of the dominant component to the submissive component

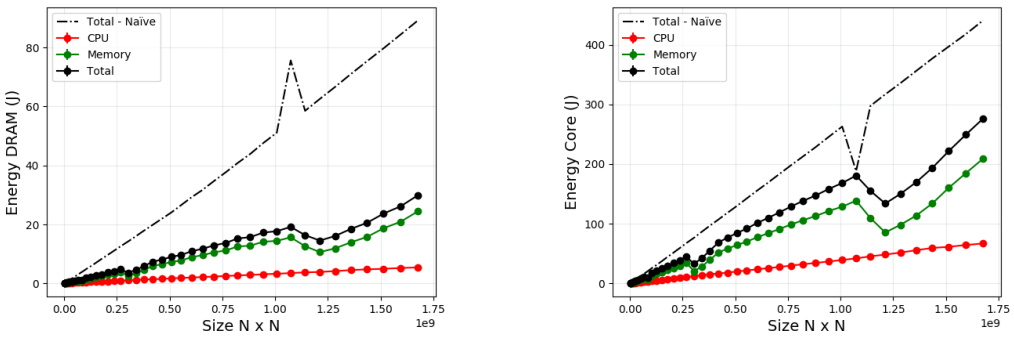
Comparing the base cases models with the Naïve Algorithm models (see Equations 6.2), the dominant component was not the same as the base cases superior to 4×4. Furthermore, the resource usage ratio between the components on both energy and time are inferior. Then, comparing the base cases models with the Blocked Transpose Algorithm models (see Equations 6.3), the dominant component remained the same until the base case of 4×4 but the resource usage ratio between the components on both energy and time are inferior. Finally, comparing the base cases models with the Cache-Oblivious Algorithm with a base case of 1×1 (see Equations 6.4), the dominant component remained the same until the base case of 4×4 and the resource usage ratio between the components on both energy and time increased.

Moreover, in Figure 6.10, with a base case of 4×4, both CPU instructions and memory accesses almost partially overlap. Comparing with Figures 6.8, we can notice that in both energy consumed and running time, the memory accesses remained practically similar, while the CPU instructions decreased substantially. Figure 6.11 presents the experimental results with a base case of 16×16. Differently from Figure 6.10, increasing the base case, lines start getting further apart with a dominance of the memory accesses, similar to the Naïve Algorithm. Finally, Figures 6.12 and 6.13 present experimental results of the two larger base cases, 64×64 and 256×256, respectively. Both base cases presented a higher dominance by the memory accesses component and the highest values of energy consumed and running time, although the highest base case presented slightly larger values.

We conclude that the Cache-Oblivious Algorithm behaves similarly to the Naïve Algorithm as the base case increases. However, despite the similarity with the Naïve Algorithm, even with the larger base case, the cache conflict observed in the Naïve Algorithm (see Section 6.1), seems to be attenuated by the memory access pattern of the algorithm. Moreover, the lowest consumption of energy and time were obtained with base case 4×4, even better than with the base case of 1×1.

On another note, in Figure 6.11, it is noticeable an unusual behaviour from $N = 32768$ to 40960. The value of 32768 already inflicted some disturbances in the performance of the Naïve Algorithm (see Section 6.1), however the effect did not spread over the following values of N. Therefore, we decided to further investigate this issue by analysing the Core and DRAM components.

Figure 6.14 shows the energy consumed at the DRAM (left plot) and at the Core (right plot), by CPU instructions (red) and by memory accesses (green) as well as the total (black).



(a) Energy DRAM

(b) Energy Core

Figure 6.14: Cache-Oblivious Algorithm with a base case of 16×16 - Energy consumed by the DRAM (a) and Core (b) components

According to both previous figures and Figure 6.11, only the memory accesses component presents this behaviour. These results suggest that something that relates memory usage or cache with both energy consumed and running time was in the origin of this behaviour. Therefore, we decided to analyse the number of stalled cycles, or idle cycles, for each level with a base case of 16×16.

Figure 6.15 shows the number of cache misses stall cycles (see Section 2.2) of the Cache-Oblivious Algorithm with a base case of 16×16 at different cache levels, L1 (orange), L2 (blue-green) and L3 (dark yellow). It shows oscillating values, although there is no

notorious correlation between the memory accesses displayed behaviour and the cache misses stalls.

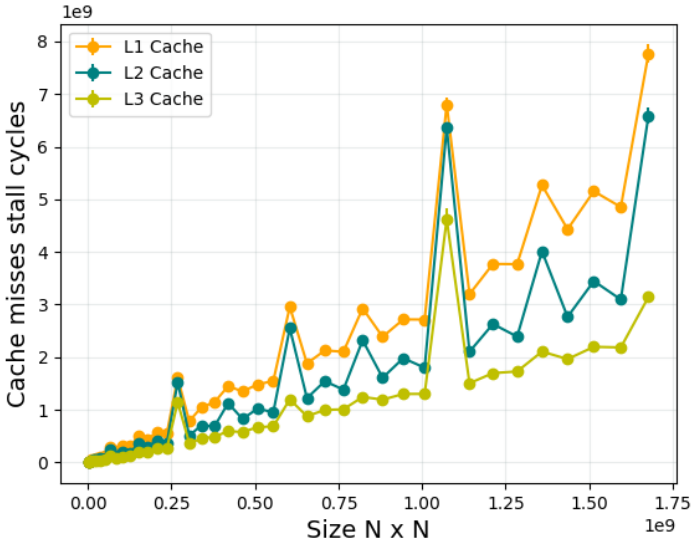
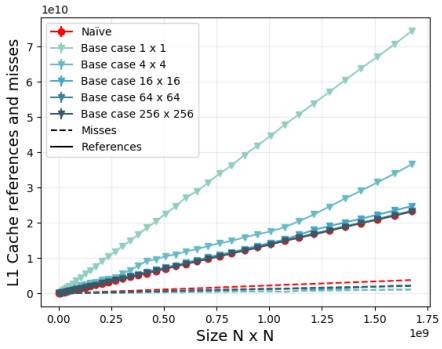


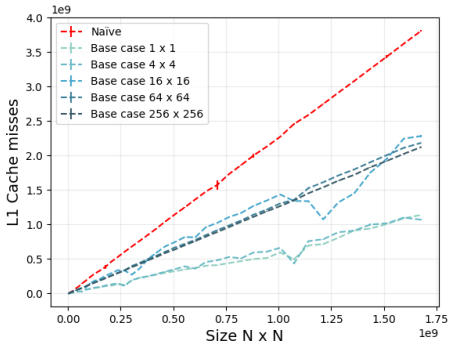
Figure 6.15: Cache-Oblivious Algorithm with a base case of 16×16 - Cache misses stall cycles

In order to understand the behaviour of the cache performance with larger base cases and the unusual behaviour of the memory accesses observed in Figure 6.11 at the base case of 16×16, we analysed the number of cache references and cache misses for each base case in each cache level.

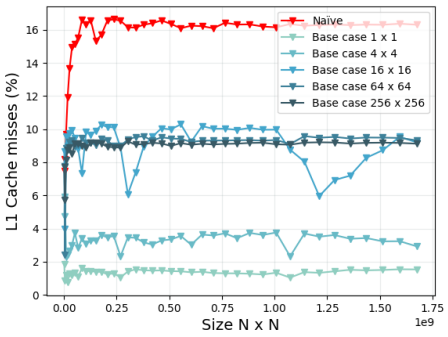
Figure 6.16 shows the number of cache references and misses (a), the number of cache misses (b) and cache misses percentage (c) at L1 cache.



(a) References and misses



(b) Misses



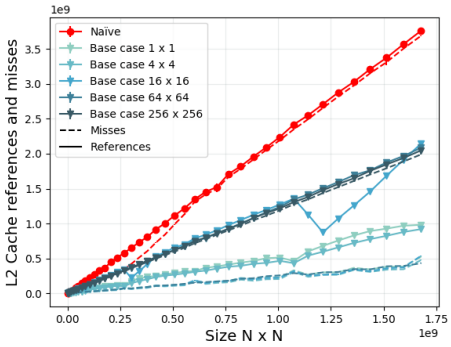
(c) Misses percentage

Figure 6.16: Cache-Oblivious Algorithm - L1 Cache references and misses (a), cache misses (b) and cache misses percentage (c)

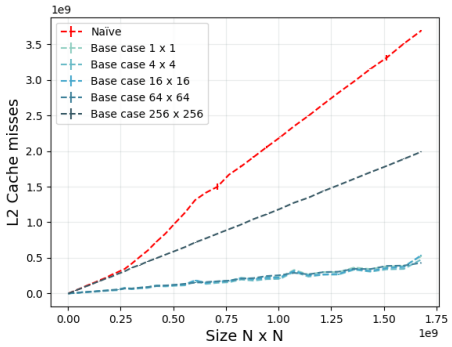
In comparison with Naïve Algorithm and with the other base cases, at the L1 cache, the two lowest base cases have the largest number of cache references and the lowest number of cache misses. Moreover, with larger base cases, the number of references converge to the number of references performed by the Naïve Algorithm, similar to what happens with energy consumed and running time (see Figures 6.12 and 6.13). However, the number of cache misses is not as high as the number of cache misses in the Naïve Algorithm.

The high number of cache references and the low number of cache misses at the L1 cache indicates a better memory usage by the algorithm at this level. Particularly, the partitioning of the algorithm until the base case of one element causes both columns and rows to be reused at lower level caches. With the increase of the base case, the reuse of cached data targets more the rows, causing more cache misses when accessing columns. For example, while the percentage of cache misses with the base case of 1×1 was around 1.5%, with larger base cases increases to around 9%.

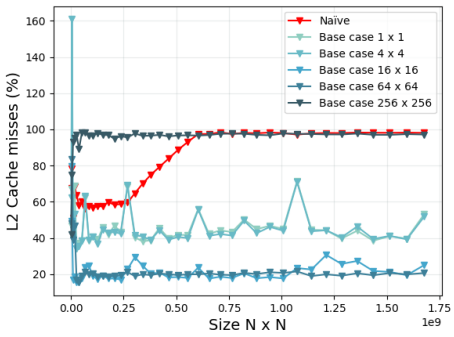
Figure 6.17 shows the number of cache references and misses (a), the number of cache misses (b) and cache misses percentage (c) at L2 cache.



(a) References and misses



(b) Misses

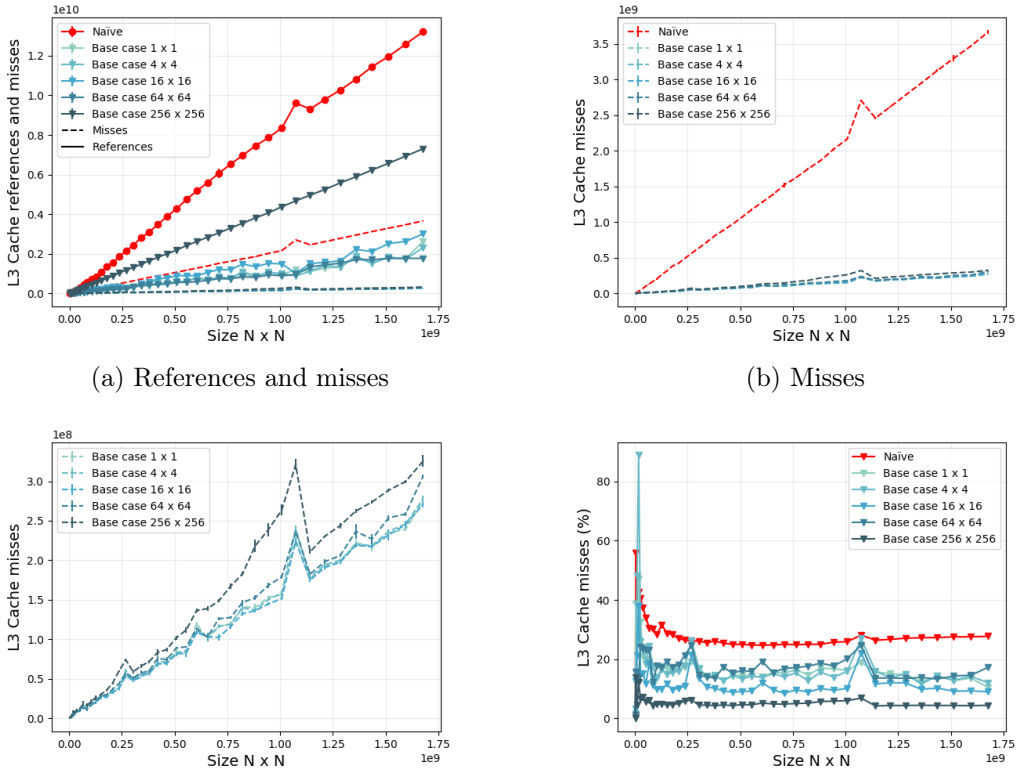


(c) Misses percentages

Figure 6.17: Cache-Oblivious Algorithm - L2 Cache references and misses (a), and misses (b)

As explained in Section 6.2, since the L1 cache is quite explored, one of the L2 or L3 caches, generally the L2, may reveal a worse usage. At the L2 cache, the percentage of cache misses in the two lowest base cases was around 40% to 45%, the largest base case was almost 98% and the other base cases presented a percentage of around 20%.

Figure 6.18 shows the number of cache references and misses (a), the number of cache misses (b) and cache misses percentage (c) at L3 cache. Figure 6.18c shows the same information from Figure 6.18b, but without the results obtained by the Naive algorithm and zooming in for smaller values of cache misses.



(a) References and misses

(b) Misses

(c) Misses without Naïve Algorithm

(d) Misses percentage

Figure 6.18: Cache-Oblivious Algorithm - L3 Cache references and misses (a), cache misses (b) and (c), and cache misses percentage (d)

Figures 6.18a and 6.18b show a similar number of cache references and misses at the L3 cache, with the exception of the largest base base which presents a higher number of cache references and misses. Moreover, the largest base case presented a percentage around 6% while the other base cases presented a percentage of around 10% to 17%.

On another note, the number of cache misses at L1 cache in Figure 6.16b with base case of 16×16 and the number of cache references at L2 cache in Figure 6.17a, present a similar behaviour to the observed behaviour on energy and time with base case of 16×16 of the memory accesses component in Figure 6.11. Therefore, the cache performance results suggest that it is possible to relate, in this case, the number of cache misses in the L1 cache and the number of cache references in the L2 cache with the energy and time consumption behaviour.

As a conclusion, the Cache-Oblivious Algorithm presented better results than the Naïve Algorithm and contrarily to what we have observed with the Naïve Algorithm, the running time and energy consumed are higher in the CPU instructions component than in the memory accesses component. Moreover, the previous results showed that we can achieve different behaviours in both components by varying the base case size.

6.3.2 Cache-Aware Algorithm

The second analysed algorithm was the Cache-Aware Algorithm (see Section 4.4). This algorithm is similar to the Naïve Algorithm. However, it performs the matrix transposition by dividing the matrix in blocks of specific sizes, ideally, cache sizes.

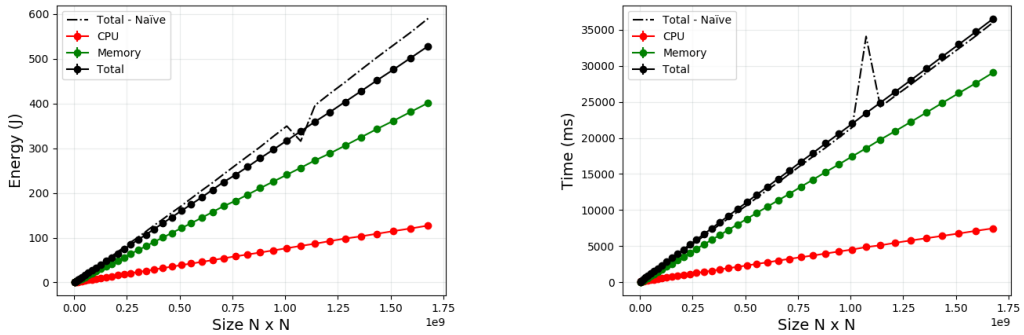
The machine where the experiments were performed had the following cache sizes:

- a L1 cache capable of storing $\frac{32768}{4} = 8192$ elements ($\sim 91 \times 91$ sub-matrix) of 4 bytes.
- a L2 cache capable of storing $\frac{262144}{4} = 65536$ elements (256×256 sub-matrix) of 4 bytes.
- a L3 cache capable of storing $\frac{6291456}{4} = 1572864$ elements ($\sim 1255 \times 1255$ sub-matrix) of 4 bytes.
- a cache line capable of storing $\frac{64}{4} = 16$ elements (4×4 sub-matrix) of 4 bytes.

In order to understand what is the impact of different submatrices sizes and which one have better performance, we ran the algorithm using blocks with sizes of $B \times B$, where B had the values of 1255, 256, 91, 64, 16 and 4. The values of 1255, 256, 91 and 4 were chosen according to the size of each cache, while the values of 16 and 64 were chosen to understand the algorithm's performance between the values 4 and 91. Since the performance results for the values of 1255, 256 and 91 were similar, we only reported results for $B = 256$.

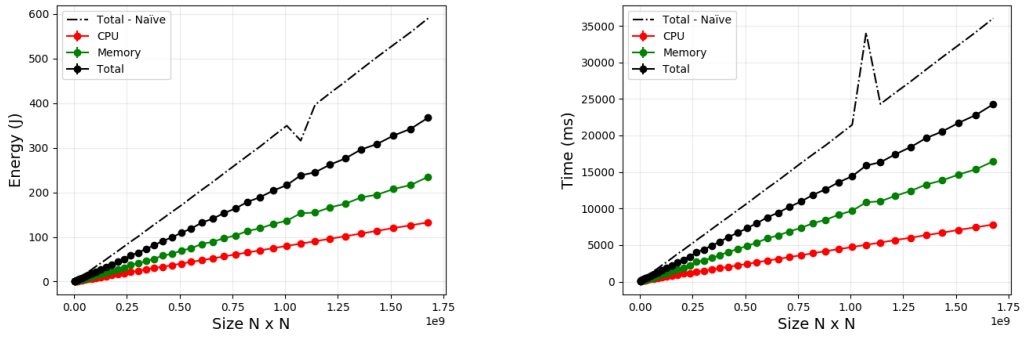
Similar to the two previous Section 6.1 and 6.2, we measure both running time and energy consumed by CPU instructions and memory accesses of the algorithm. Moreover, the increase in the number of CPU instructions and memory accesses affects time and energy consumed in a linear increasing way. Therefore, it is expected that the time spent by the algorithm and energy consumed are strongly (positively) correlated.

Figure 6.19, 6.20, 6.21 and 6.22, show the performance of Cache-Aware Algorithm in terms of Energy (a) and Time (b) with respect to CPU instructions (green line) and memory access (red line) and total (black line), for block sizes 256×256, 64×64, 16×16 and 4×4, respectively. In addition, dashed lines present the total energy consumed and running time by the Naïve Algorithm. Moreover, for each base case we performed a linear regression, where N^2 is the number of matrix elements.



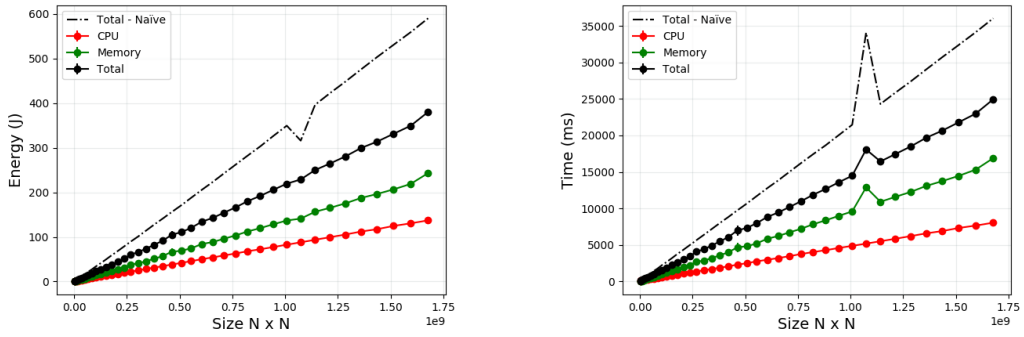
(a) Energy (b) Time

Figure 6.19: Performance of Cache-Aware Algorithm with a block size of 256×256 in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



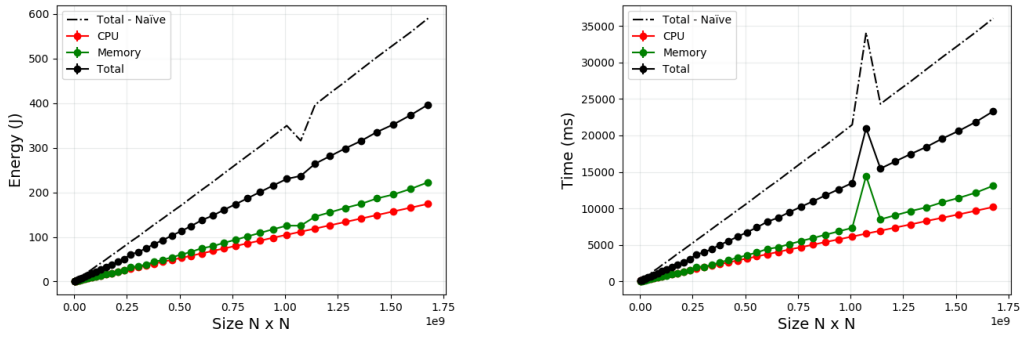
(a) Energy (b) Time

Figure 6.20: Performance of Cache-Aware Algorithm with a block size of 64×64 in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

Figure 6.21: Performance of Cache-Aware Algorithm with a block size of 16×16 in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

Figure 6.22: Performance of Cache-Aware Algorithm with a block size of 4×4 in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses

Linear regression models of Cache-Aware Algorithm with a block size of 256×256:

$$E_{CPU} = 0.2266 + 7.566e^{-08}N^2 \ (R^2 = 1) \tag{6.9a}$$

$$T_{CPU} = 97.90 + 440.6e^{-08}N^2 \ (R^2 = 1) \tag{6.9b}$$

$$E_{Memory} = -0.2028 + 23.91e^{-08}N^2 \ (R^2 = 1) \tag{6.9c}$$

$$T_{Memory} = -18.47 + 1732e^{-08}N^2 \ (R^2 = 1) \tag{6.9d}$$

Linear regression models of Cache-Aware Algorithm with a block size of 64×64:

$$E_{CPU} = 0.2137 + 7.896e^{-08}N^2 \ (R^2 = 1) \tag{6.10a}$$

$$T_{CPU} = 98.58 + 459.9e^{-08}N^2 \ (R^2 = 1) \tag{6.10b}$$

$$E_{Memory} = -0.3946 + 13.75e^{-08}N^2 \ (R^2 = 1) \tag{6.10c}$$

$$T_{Memory} = -25.54 + 972.5e^{-08}N^2 \ (R^2 = 1) \tag{6.10d}$$

Linear regression models of Cache-Aware Algorithm with a block size of 16×16:

$$E_{CPU} = 0.2464 + 8.184e^{-08}N^2 \ (R^2 = 1) \tag{6.11a}$$

$$T_{CPU} = 95.74 + 476.1e^{-08}N^2 \ (R^2 = 1) \tag{6.11b}$$

$$E_{Memory} = -0.7576 + 13.84e^{-08}N^2 \ (R^2 = 0.999) \tag{6.11c}$$

$$T_{Memory} = -37.25 + 966.2e^{-08}N^2 \ (R^2 = 0.9992) \tag{6.11d}$$

Linear regression models of Cache-Aware Algorithm with a block size of 4×4:

$$E_{CPU} = 0.4255 + 10.36e^{-08}N^2 \ (R^2 = 1) \tag{6.12a}$$

$$T_{CPU} = 98.29 + 600.9e^{-08}N^2 \ (R^2 = 1) \tag{6.12b}$$

$$E_{Memory} = -3.435 + 13.05e^{-08}N^2 \ (R^2 = 1) \tag{6.12c}$$

$$T_{Memory} = -167.6 + 762.2e^{-08}N^2 \ (R^2 = 0.9987) \tag{6.12d}$$

The previous figures suggest that the memory accesses component has a strong influence on the both time and energy. Furthermore, analysing the models presented above, since R^2 values are practically 1 or close to 1, we conclude that there is a linear relationship between the independent variable, N^2, and the dependent variables, E_{CPU}, E_{Memory}, T_{CPU} and T_{Memory}.

From the previous models, with the decrease of the block size, the resource usage ratio of the memory accesses component to the CPU instructions component decreases for both energy and time. Considering the block size of 256×256, the memory accesses component is approximately 3.1 times greater on energy and 3.9 times greater on time than the CPU

instructions component. Considering the block size of 64×64, the memory accesses component is approximately 1.7 times greater on energy and 2.11 times greater on time than the CPU instructions component. Finally, considering the block size of 4×4, the memory accesses component is approximately 1.26 times greater on energy and 1.27 times greater on time than the CPU instructions component. The other resource usage ratios are presented in Table 6.2. From this, we conclude that memory accesses have a less stronger weight on the time and energy performance of the Cache-Aware Algorithm with the decrease of the block size.

Base case	Dominant component	$\frac{E_{DominantComponent}}{E_{SubmissiveComponent}}$	$\frac{T_{DominantComponent}}{T_{SubmissiveComponent}}$
256×256	Memory accesses	3.1	3.9
16×16	Memory accesses	1.7	2.11
64×64	Memory accesses	1.69	2.02
4×4	Memory accesses	1.26	1.27

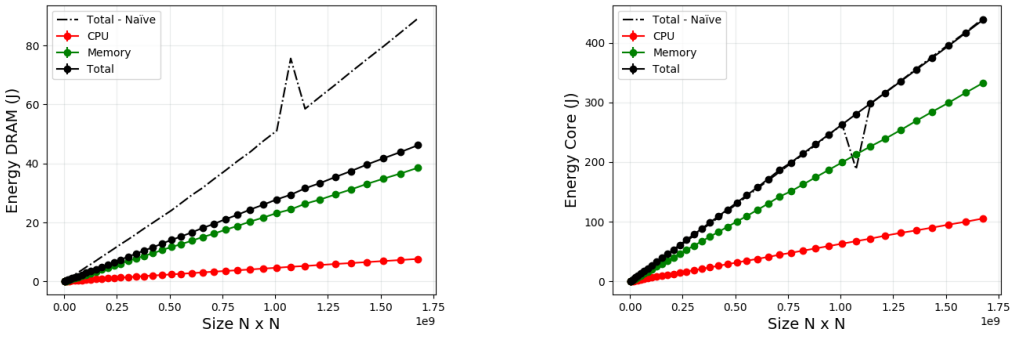
Table 6.2: Cache-Aware Algorithm - Resource usage ratio of the dominant component to the submissive component

Comparing the block size models with the Naïve Algorithm models (see Equations 6.2), the dominant component remained the same and the resource usage ratio between the components on both energy and time are largely inferior. Then, comparing the block size models with the Blocked Transpose Algorithm models (see Equations 6.3), the dominant component changed and the resource usage ratio between the components on both energy and time are largely inferior. Finally, comparing the block size models with the Cache-Oblivious Algorithm models (see Equations 6.4, 6.5, 6.7, 6.8), the dominant component changed and the resource usage ratio between the components on both energy and time are about half.

Figure 6.19 shows that with a block size of 256×256, we can achieve similar results to the Naïve Algorithm results. In Figures 6.20 and 6.21, with a block size of 64×64 and 16×16, respectively, we can notice that memory accesses converge close to the CPU instructions. Finally, Figure 6.22 shows close values of energy and the running time by the CPU instructions and memory accesses with a block size of 4×4. Moreover, with the decrease of the block size, a slight peak at $N = 32768$ increases. This peak must be related to the previously explained case in Section 6.1.

Figure 6.19, as mentioned, shows a similar performance with the Naïve Algorithm. However, while the running time of the Cache-Aware Algorithm with a block size of 256×256 and the Naïve Algorithm present similar results, the energy consumed by the Cache-Aware Algorithm displays slightly lower results. This observation suggests that something influenced the energy consumption. Therefore, we analysed the energy consumed on the Core and DRAM components.

Figure 6.23 shows the energy consumed by DRAM (a) and at Core (b), by CPU instructions (red) and by memory accesses (green) as well as the total (black).

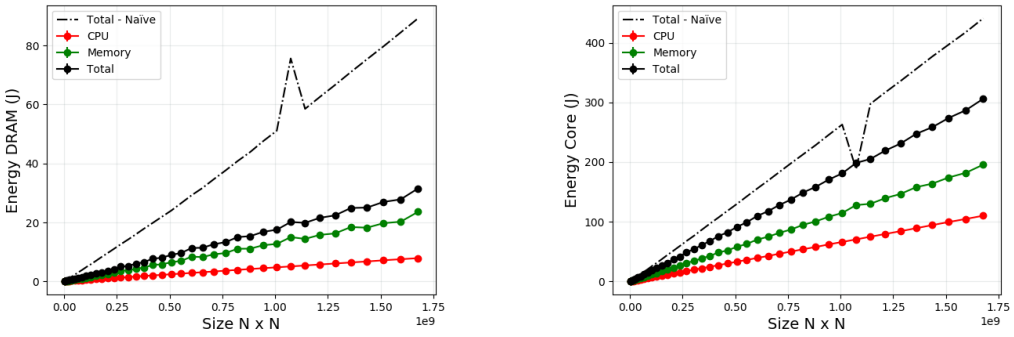


(a) Energy DRAM (b) Energy Core

Figure 6.23: Cache-Aware Algorithm with a block size of 256×256 - Energy consumed by the DRAM (a) and Core (b) components

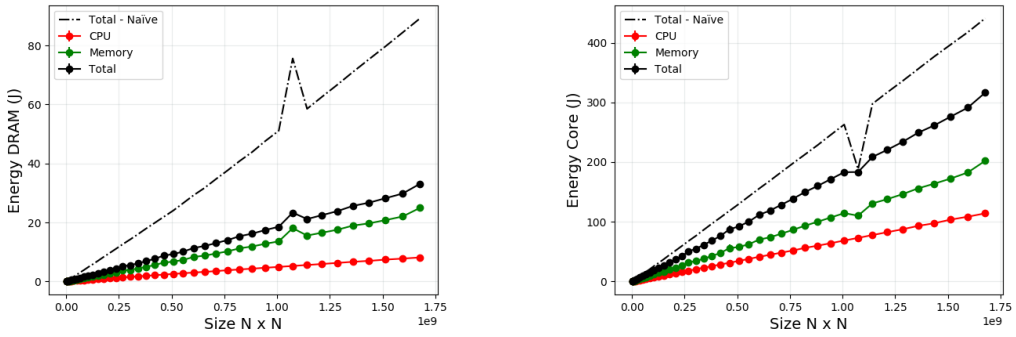
Figure 6.23b shows exactly the same amount of Core energy consumed by the Naïve Algorithm but without the valley when $N = 32768$. Analysing the energy consumed by the DRAM in Figure 6.23a, we can notice that the total energy decreased to half. To understand what was the effect of other block sizes, we analysed their energy consumed on Core and DRAM components.

Figure 6.24, 6.25 and 6.26, show the energy consumed by DRAM (a) and at Core (b), by CPU instructions (red) and by memory accesses (green) as well as the total (black), for block sizes 64×64, 16×16 and 4×4, respectively.



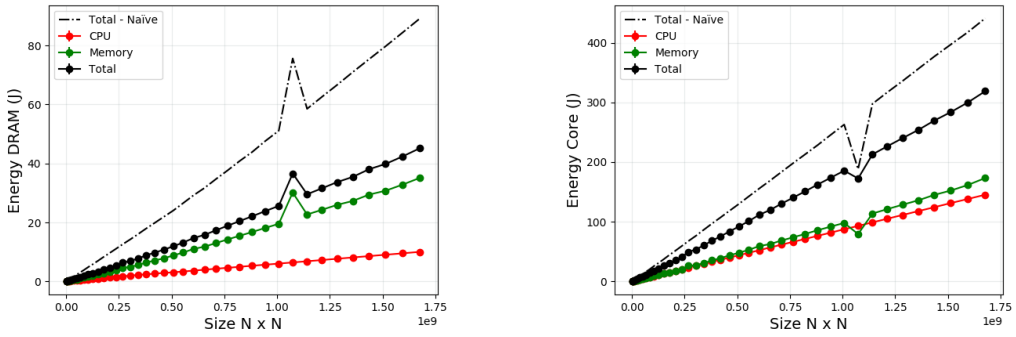
(a) Energy DRAM (b) Energy Core

Figure 6.24: Cache-Aware Algorithm with a block size of 64×64 - Energy consumed by the DRAM (a) and Core (b) components



(a) Energy DRAM (b) Energy Core

Figure 6.25: Cache-Aware Algorithm with a block size of 16×16 - Energy consumed by the DRAM (a) and Core (b) components



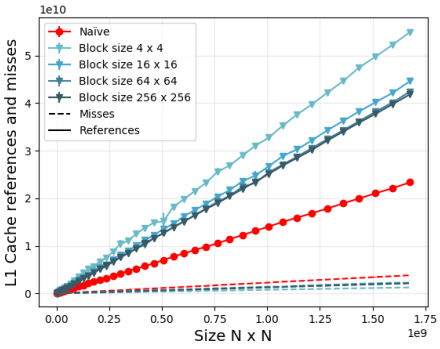
(a) Energy DRAM (b) Energy Core

Figure 6.26: Cache-Aware Algorithm with a block size of 4×4 - Energy consumed by the DRAM (a) and Core (b) components

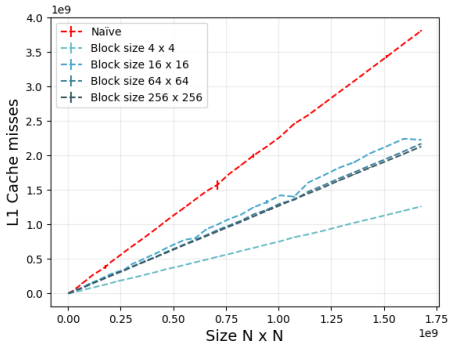
Comparing both Figures 6.24 and 6.25, we can notice that there is just a slight difference in the energy consumed on both components and that the peak stands out more with the decrease of the block size. However, Figure 6.26 shows two noticeable effects. In Figure 6.26a, there is an increase on energy consumed by the DRAM, similar to the results achieved with a block size of 256×256. On the other hand, Figure 6.26b shows a decrease on energy consumed by the memory accesses. This behaviour suggests an exploitation on the memory accesses with a better improvement, decreasing the computation time of the CPU and consequent energy Core consumed.

The previous results showed that the division of the matrix into submatrices of different sizes influence both energy and time consumption and suggest a improvement of memory usage. To understand the impact on cache performance, we analysed the number of cache references and cache misses for each block size in each cache level.

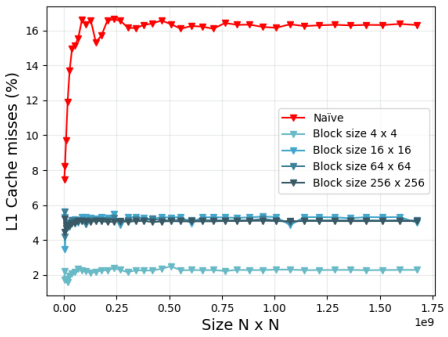
Figure 6.27 shows the number of cache references and misses (a), the number of cache misses (b) and cache misses percentage (c) at the L1 cache.



(a) References and misses



(b) Misses

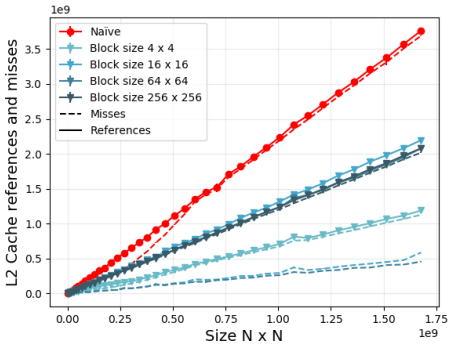


(c) Misses percentage

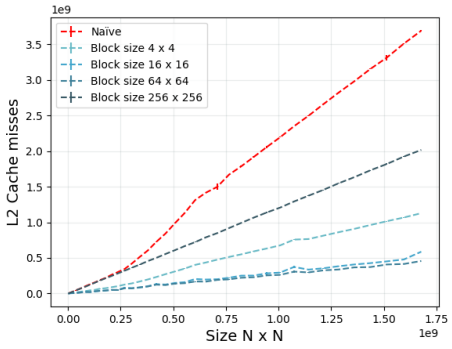
Figure 6.27: Cache-Aware Algorithm - L1 Cache references and misses (a), cache misses (b) and cache misses percentage (c)

In comparison with the Naïve Algorithm the different block sizes present a higher number of cache references. However, Figure 6.27b shows a lower number of cache misses in comparison with the Naïve Algorithm. The high number of cache references and low number of cache misses at the L1 cache indicates a better memory usage by the algorithm at this cache level. Moreover, the percentage of cache misses with the smallest block size was around 2.25% while others presented a percentage of around 5%.

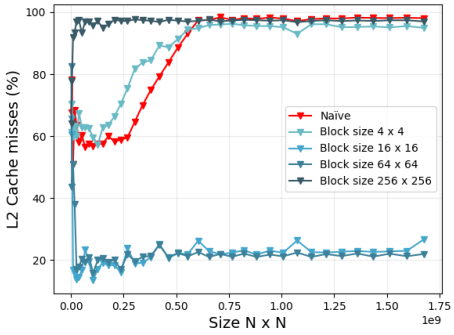
Figure 6.28 shows the number of cache references and misses (a), the number of cache misses (b) and cache misses percentage (c) at the L2 cache, L2 cache.



(a) References and misses



(b) Misses

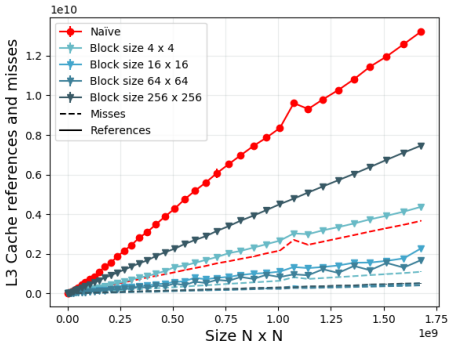


(c) Misses percentage

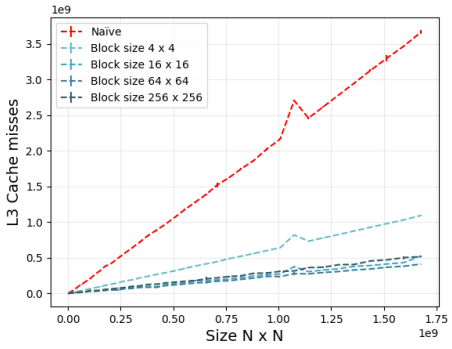
Figure 6.28: Cache-Aware Algorithm - L2 Cache references and misses (a), cache misses (b) and cache misses percentage (c)

As explained in Section 6.2, since the L1 cache is quite explored, one of the L2 and L3 caches, generally the L2, may reveal a worst usage results. Figure 6.28 shows a poor cache efficiency at the L2 cache with the lowest and highest block sizes, 256×256 and 4×4, respectively. Therefore, while the percentage of cache misses with the smallest and largest block sizes was around 95% to 97%, the other block sizes presented a percentage of around 22%.

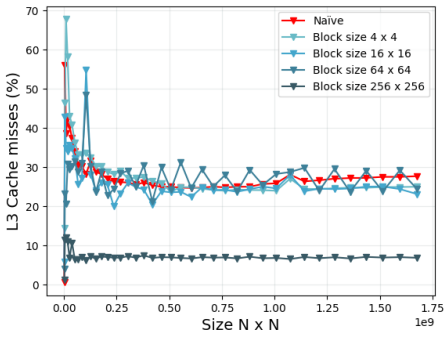
Figure 6.29 shows the number of cache references and misses (a), the number of cache misses (b) and cache misses percentage (c) at L3 cache.



(a) References and misses



(b) Misses



(c) Misses percentage

Figure 6.29: Cache-Aware Algorithm - L3 Cache references and misses (a), cache misses (b) and cache misses percentage (c)

Figure 6.29a show a similar number of cache references with the medium block sizes and a higher number of cache references with the smallest and largest block size. However, in Figure 6.29b we can notice that with the exception of the smallest block size, all other block sizes perform a similar number of cache misses. Moreover, the percentage of cache misses with the largest block size was around 7% while others presented a percentage of around 25%.

The previous results revealed that by partitioning the overall matrix into submatrices with different sizes, we can reduce the running time and energy consumption and optimise the cache's performance at different cache levels. Since the Cache-Aware Algorithm is mainly composed of nested loops, there is always the possibility of increasing the number of subdivisions inside the submatrices. Therefore, to extend this algorithm we applied a technique called *loop tilling* (see Section 2.4). This technique improves spatial and temporal locality, establishing a different memory access pattern under this algorithm, dividing the submatrices into other submatrices with a specified size. To understand the impact of this technique, we ran experiments dividing the original submatrices into submatrices of size 4×4, according to the machine's cache line size. Moreover, due to the chosen sub-matrix size, experiments were only performed on block sizes with values of 256, 64 and 16. The results showed slightly lower values on energy consumed and running time, as well as a similar cache performance than the Cache-Aware Algorithm with a block size of 4×4 but with a smaller percentage of cache misses at L2 cache.

As a conclusion, the Cache-Aware Algorithm presented better results than the Naïve Algorithm and similar to what we have observed with the traditional algorithm, the running time and energy consumed are higher in the memory accesses component than in the CPU

instructions component. Moreover, the previous results showed that we can achieve different behaviours in both components, CPU instructions and memory accesses, varying the memory accesses pattern, e.g. loop tilling technique and the block size, and that the memory access pattern can influence the energy consumption and running time, as well as the cache performance.

6.3.3 Discussion

All in all, both the Cache-Oblivious and Cache-Aware algorithms presented better results than the Naïve Algorithm, although with different energy consumption behaviours. Contrarily to what we observed in the Naïve Algorithm, the Cache-Oblivious shows higher running time and energy consumed for the CPU instructions than for the memory accesses component. However, with the increase of the base case, the memory accesses component energy consumed and running time increases, becoming similar to the Naïve Algorithm results. The Cache-Aware Algorithm, similar to what we have observed with the traditional algorithm, the running time and energy consumed are higher in the memory accesses component than in the CPU instructions component, although the memory accesses component decreases as block size also decreases.

Comparing both algorithms in terms of energy and time, the Cache-Oblivious Algorithm presented an overall better performance than the Cache-Aware Algorithm. In terms of cache performance at each cache level, the Cache-Oblivious also presented an overall better performance. Furthermore, when comparing both algorithms with the Blocked Transpose Algorithm, in terms of energy and time, we observe that the Blocked Transpose Algorithms shows similar results to the Cache-Aware Algorithm for the smaller block sizes and better results for larger block sizes. The Block Transpose Algorithm also shows better results than the smallest base case of the Cache-Oblivious Algorithm, but worst results otherwise. However, in terms of cache performance at each cache level, it presents results similar to the smallest base case of the Cache-Oblivious Algorithm, sometimes slightly better, and better results for the larger base cases and for all the block sizes of the Cache-Aware Algorithm. These practical results comparison can be seen in the Appendix B.

6.4 Research Question 4

RQ4 *One way of improving the running time of an algorithm for Matrix Transposition is to parallelize it. What is the effect of Multicore and Multithreading techniques in the energy consumption?*

The previous section provided empirical evidences that both approaches, Cache-Oblivious Algorithm and Cache-Aware Algorithm, decreased energy consumption using different memory access patterns. However, another aspect that may affect energy consumption is parallelization. To improve our understanding of the impact of parallelization using Multicore and Multithreading techniques on energy consumption, we analysed the Cache-Oblivious Parallel Algorithm and the Cache-Aware Parallel Algorithm.

6.4.1 Cache-Oblivious Parallel Algorithm

The Cache-Oblivious Parallel Algorithm (see Section 4.3.1) is the version of the Cache-Oblivious Algorithm that considers parallelism. As aforementioned, the Cache-Oblivious Algorithm performs the matrix transposition using a divide-and-conquer approach, dividing the matrix recursively until reaching a base case. To accomplish parallelization, the parallel version receives as arguments the initial row, last row, initial column, and last column of the initial submatrices, each of which to be processed by a thread.

The experiments described below were conducted considering the parallelization of the matrix by columns, similarly to the non-parallel algorithm's procedure. We also considered the experiments with a parallelization of the matrix by rows. However, the achieved results were quite similar.

From Section 6.3.1, we observed that the Cache-Oblivious Algorithm with a base case of 4×4 presented the best performance in terms of energy consumed and running time. Therefore, to understand the performance of this algorithm with parallelization, we analyse its performance considering different numbers of virtual cores, $c \in \{1, 2, 3, 4, 5, 6, 7, 8\}$, and threads per core, $t \in \{2, 4, 8, \ldots, 1024\}$. The Cache-Oblivious Algorithm with a base case of 1×1 presented the best cache performance of our experiments, as well as the lowest energy consumed by the memory accesses component. However, this base case's CPU instructions component presented a high impact on energy consumed. Therefore, to understand the parallelization effect on this algorithm's performance on energy and time, and cache performance, we also analyse the Cache-Oblivious Algorithm with a base case of 1×1.

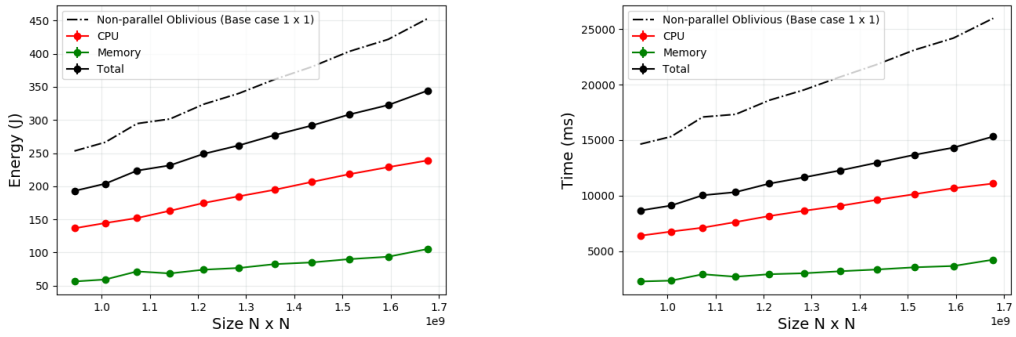
Similarly to the previous sections, we measured both the running time and energy consumed by the CPU instructions and memory accesses of the algorithm. Moreover, the increase in the number of CPU instructions and memory accesses affects time and energy consumed in a linear increasing way. Since the performance results achieved with small values of N are not relevant and to facilitate the creation of different plots, we just considered the values of $N \in \{30720, 31744, 32768, \ldots, 40960\}$.

The first experiments conducted analysed the performed of the Cache-Oblivious Parallel Algorithm using the Multicore technique (see Section 2.5) to achieve parallelization. Moreover, to achieve a multicore parallelization, each virtual core executed one single thread assigned using a Round-robin scheduling.

Cache-Oblivious Parallel Algorithm with a base case of 1×1

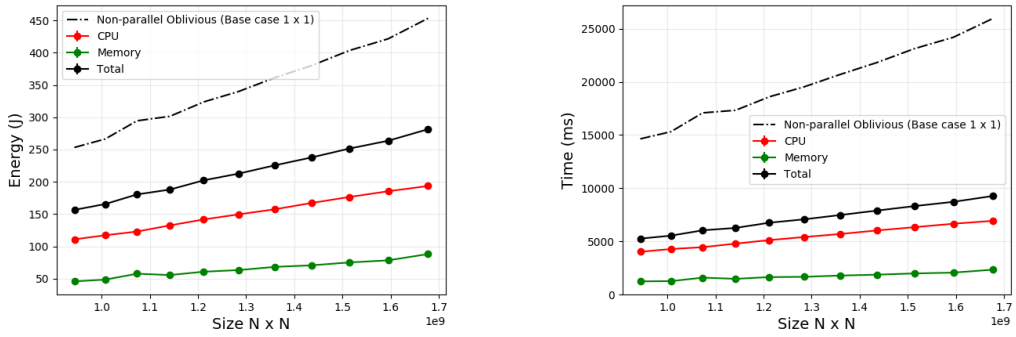
Figure 6.30, 6.31 and 6.32, show the performance of Cache-Oblivious Parallel Algorithm in terms of Energy (left plot) and Time (right plot) with respect to CPU instructions (green line) and memory access (red line) and total (black line), with a base case of 1×1 on two, four and eight virtual cores, respectively.[1] In addition, dashed lines present the total energy consumed and running time by the Non-parallel Cache-Oblivious Algorithm with a base case of 1×1. Moreover, for each base case we performed a linear regression, where N^2 is the number of matrix elements.

[1] The figures on one, three, five, six and seven virtual cores are available in the Appendix C.



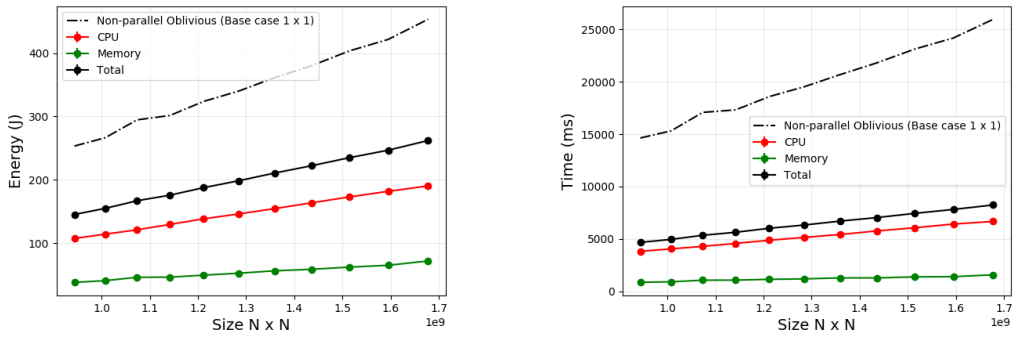
(a) Energy (b) Time

Figure 6.30: Performance of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on two virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

Figure 6.31: Performance of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on four virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

Figure 6.32: Performance of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on eight virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses

Linear regression models of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on two virtual cores:

$$E_{CPU} = 0.5939 + 14.30e^{-08}N^2 \ (R^2 = 0.999) \tag{6.13a}$$

$$T_{CPU} = 150 + 657.3e^{-08}N^2 \ (R^2 = 0.9989) \tag{6.13b}$$

$$E_{Memory} = 1.432 + 5.951e^{-08}N^2 \ (R^2 = 0.9607) \tag{6.13c}$$

$$T_{Memory} = 121.9 + 230.2e^{-08}N^2 \ (R^2 = 0.9283) \tag{6.13d}$$

Linear regression models of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on four virtual cores:

$$E_{CPU} = 0.9481 + 11.54e^{-08}N^2 \ (R^2 = 0.9989) \tag{6.14a}$$

$$T_{CPU} = 171.7 + 405.8e^{-08}N^2 \ (R^2 = 0.9987) \tag{6.14b}$$

$$E_{Memory} = -3.026 + 5.224e^{-08}N^2 \ (R^2 = 0.9714) \tag{6.14c}$$

$$T_{Memory} = -22.63 + 134.4e^{-08}N^2 \ (R^2 = 0.9466) \tag{6.14d}$$

Linear regression models of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on eight virtual cores:

$$E_{CPU} = -1.152 + 11.45e^{-08}N^2 \ (R^2 = 0.9997) \tag{6.15a}$$

$$T_{CPU} = 51.25 + 396.7e^{-08}N^2 \ (R^2 = 0.9994) \tag{6.15b}$$

$$E_{Memory} = -2.324 + 4.291e^{-08}N^2 \ (R^2 = 0.9879) \tag{6.15c}$$

$$T_{Memory} = 63.22 + 87.14e^{-08}N^2 \ (R^2 = 0.9667) \tag{6.15d}$$

The previous figures suggest that the number of CPU instructions has a strong influence on the both time and energy. Furthermore, analysing the models presented above. Since R^2 values are above 0.9 and close to 1, we conclude that there is a linear relationship between the independent variable, N^2, and the dependent variables, E_{CPU}, E_{Memory}, T_{CPU} and T_{Memory}.

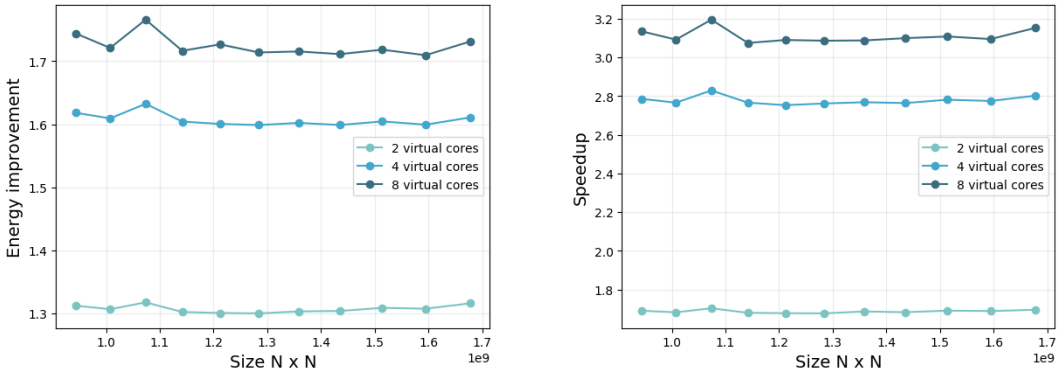
From the previous models, with the increase of the number of cores, the resource usage ratio of the CPU instructions component to the memory accesses component increases for both energy and time, although time presents a more noticeable increase. Considering two virtual cores, the CPU instructions component is approximately 2.4 times greater on energy and 2.85 times greater on time than the memory accesses component. Considering four virtual cores, the CPU instructions component is approximately 2.21 times greater on energy and 3.02 times greater on time than the memory accesses component. Finally, considering eight virtual cores, the CPU instructions component is approximately 2.67 times greater on energy and 4.55 times greater on time than the memory accesses component. The other resource usage ratios are presented in Table 6.3. In comparison with the Non-parallel Cache-Oblivious Algorithm with a base case of 1×1 models (see Equations 6.4), the dominant component changed, and the resource usage ratio between the components on energy remained similar but increased on time.

Number of virtual cores	Dominant component	$\frac{E_{DominantComponent}}{E_{SubmissiveComponent}}$	$\frac{T_{DominantComponent}}{T_{SubmissiveComponent}}$
2	CPU instructions	2.4	2.85
3	CPU instructions	2.54	3.28
4	CPU instructions	2.21	3.02
5	CPU instructions	2.26	3.13
6	CPU instructions	2.72	3.93
7	CPU instructions	2.54	3.92
8	CPU instructions	2.67	4.55

Table 6.3: Cache-Oblivious Parallel Algorithm with a base case of 1×1 - Resource usage ratio of the dominant component to the submissive component

Upon analysing the change of the resource usage ratio between the components energy and time with the increase of the number of virtual cores, it is noticeable that the interval resource usage ratio between both components is higher than the interval on energy consumed measurements. Specifically, the resource usage ratio between the components on energy increased from 2.4 to 2.67 and from 2.85 to 4.55 on time. This observation suggests that increasing the number of virtual cores has more impact on time than on energy. To understand more about the performance of the algorithm on energy consumed and running time with the increase of the number of virtual cores, we analyse the energy consumption of the algorithm considering a different number of virtual cores and the correspondent achieved speedup. Note that, to keep this analysis succinct, we only discuss the results for two, four and eight virtual cores in this section. However, the results for other values, which can be seen in Appendix C, are coherent with these.

Figures 6.33 shows the energy consumed with the different number of virtual cores (a) and the achieved speedup (b) with a base case of 1×1.



(a) Energy improvement

(b) Speedup

Figure 6.33: Performance of Cache-Oblivious Parallel Algorithm with different number of cores in terms of Energy (a) and Speedup (b)

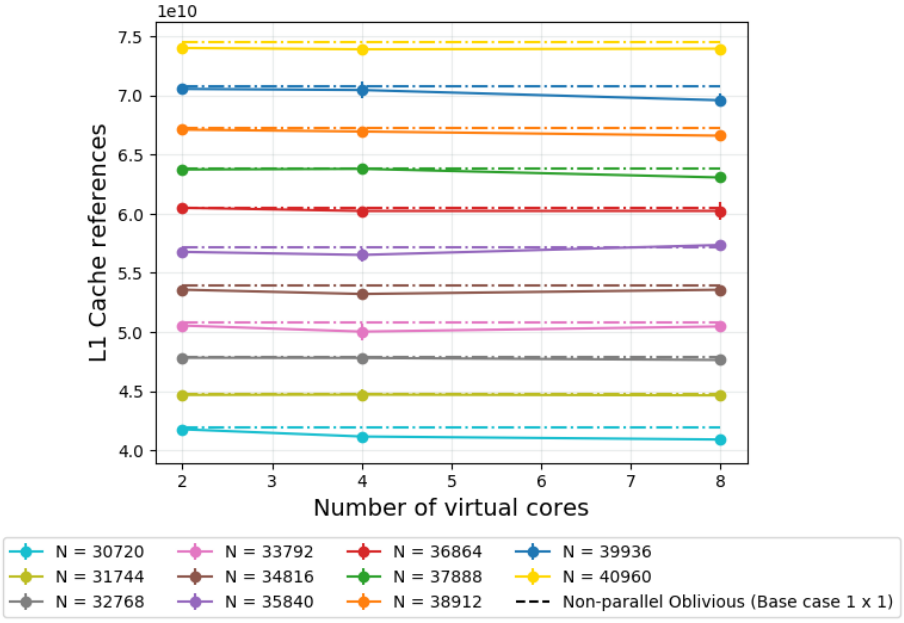
Figure 6.33a shows a slight energy improvement from one virtual core to two and from two virtual cores to four, but with more than four virtual cores the decrease is not so substantial. Figure 6.33b also shows a large speed up from one virtual core to two and from two virtual cores to four. However, with more than four virtual cores the speedup is not so large. Therefore, both figures suggest that the increase of the number of virtual cores affects the running time more than the energy consumed.

The impact of adding more virtual cores seems to be larger on the running time than on the energy consumed. Moreover, the increase in the number of virtual cores causes the computer to activate more physical cores to execute the algorithm. Therefore, to understand what is the impact on energy with the activation of multiple physical cores, we conducted two different experiments. In the first experiment, the algorithm executed with four threads, one thread per physical core. Then, in the second experiment, the algorithm executed with two threads per physical core, using just two physical cores. The results showed that the execution of the algorithm with one thread per physical core consumed more energy. Since the number of physical cores used increases, it is expected that the energy consumed increases as well. For that reason, there is a higher difference between the variation of the running time than the energy consumed.

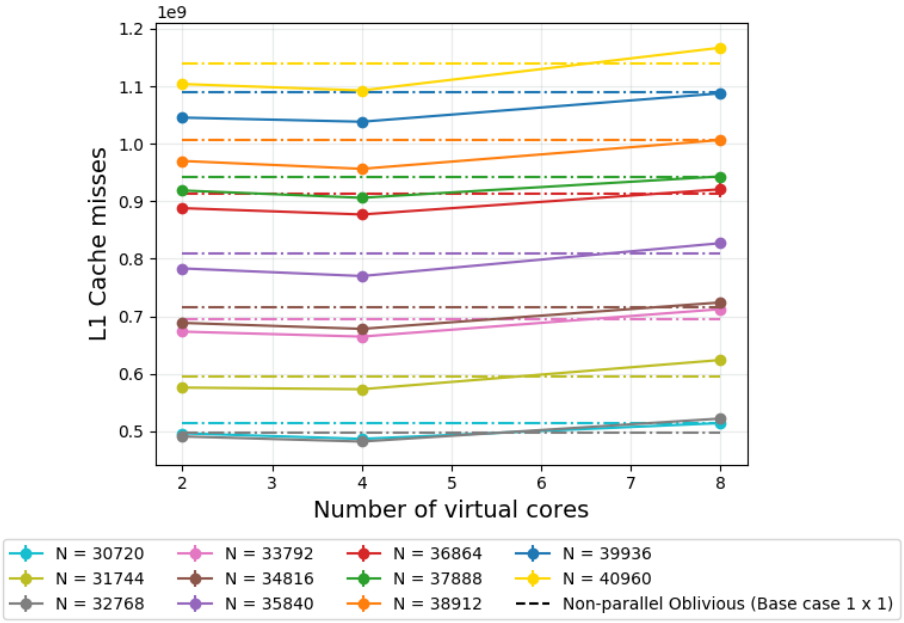
On another note, Figures 6.30, 6.31 and 6.32 show that the energy consumed and running time by the memory accesses component decreased slightly with the increase of the number of virtual cores. Therefore, this observation suggests that the cache performance did not deteriorate with the increase in the number of virtual cores. To understand the impact on cache performance, we analysed the number of cache references and misses for each number of virtual cores in each cache level.[2]

Figure 6.34 shows the number of cache references and misses (a) and cache misses (b) at L1 cache.

[2] Note that the cache performance's figures are presented on a larger scale for a better understanding of the reader.



(a) Cache references

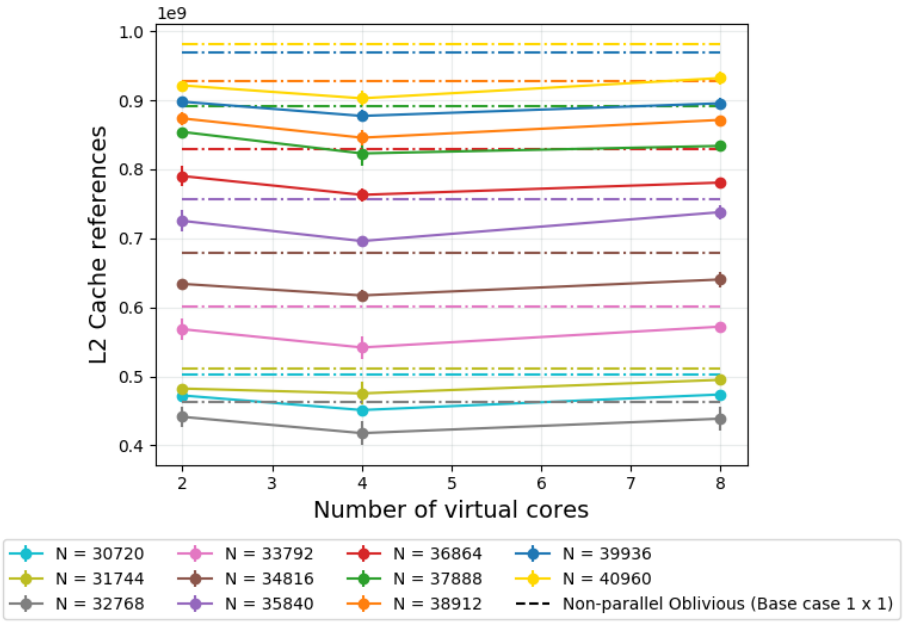


(b) Cache Misses

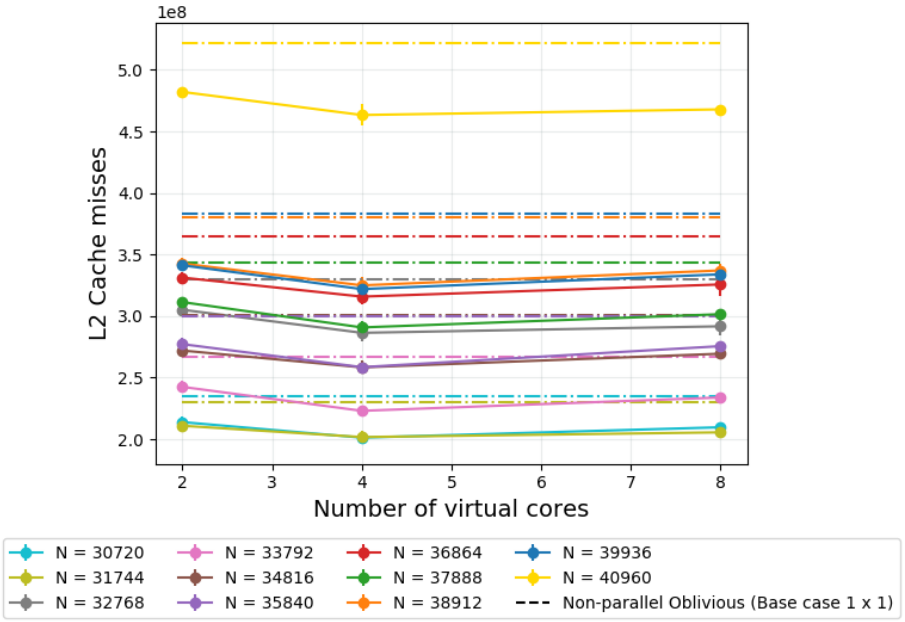
Figure 6.34: Cache-Oblivious Parallel Algorithm with different number of cores - L1 Cache references (a) and cache misses (b)

Figure 6.34a, in comparison with the Cache-Oblivious Algorithm, shows a similar number of references. However, analysing the number of cache misses in Figure 6.34b, we can notice that the number of cache misses generally decreases, except at $N = 32768$ (see Section 6.1 to a more detailed analysis of this particular size), with the increase of the number of virtual cores to four. However, with the increase of the number of virtual cores increases to eight, the number of cache misses increases and, in most matrix sizes, the number of cache misses surpasses the number of cache misses performed by the Cache-Oblivious Algorithm. These results suggest that from a certain number of virtual cores, the cache accesses performed by multiple cores conflict between each other at higher cache levels.

The plots of Figure 6.35 show the number of cache references (a) and cache misses (b) at L2 cache.



(a) Cache references



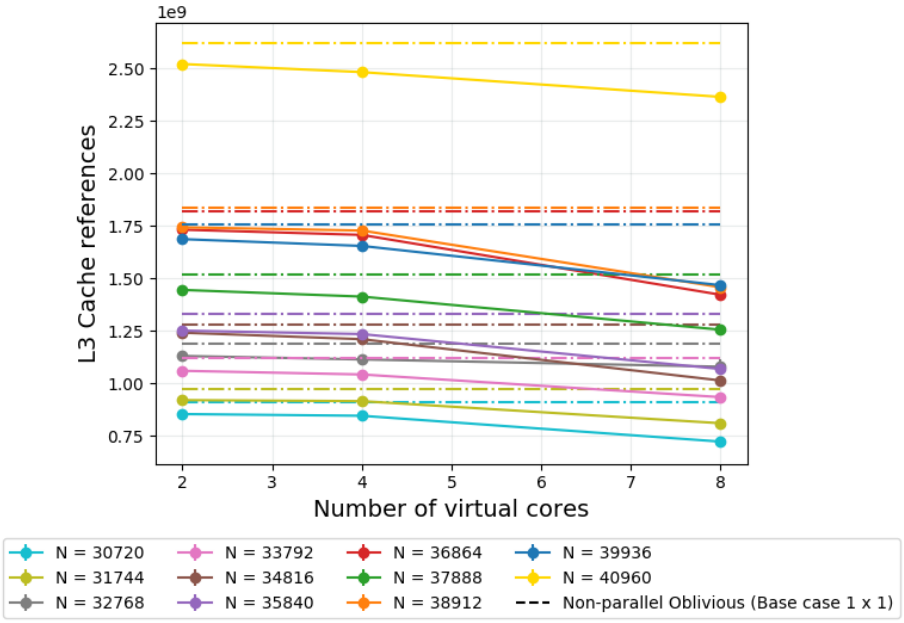
(b) Cache Misses

Figure 6.35: Cache-Oblivious Parallel Algorithm with different number of cores - L2 Cache references (a) and cache misses (b)

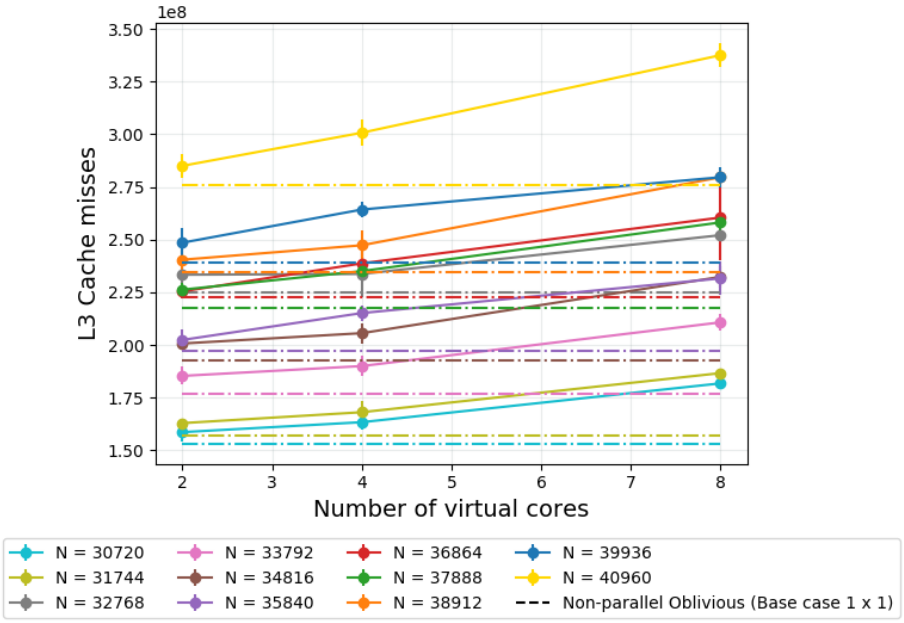
Figure 6.35a, in comparison with the Cache-Oblivious Algorithm, shows a decrease of the number of cache references until up to four virtual cores followed by an increase with more than four virtual cores. However, the number of cache misses at different matrix sizes never surpasses the number of cache misses on the Non-parallel Cache-Oblivious Algorithm. This effect is due to a similar behaviour observed in Figure 6.34b. In Figure 6.35b, we can notice that the number of cache misses decreased with the increase of the number of virtual cores. Therefore, both figures suggest that the algorithm takes advantage of the shared cache, L2

cache, between virtual cores common to the same physical core.

The plots of Figure 6.36 show the number of cache references (a) and cache misses (b) at L3 cache.



(a) Cache references



(b) Cache Misses

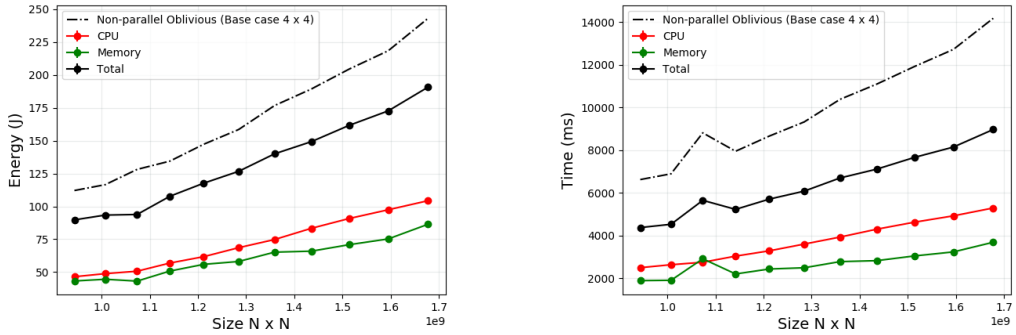
Figure 6.36: Cache-Oblivious Parallel Algorithm with different number of cores - L3 Cache references (a) and cache misses (b)

Figure 6.36a, in comparison with the Cache-Oblivious Algorithm, as expected, shows a reduction of the number of cache references, a consequence of the good performance at the L2 cache. However, Figure 6.36b shows an increase of the number of cache misses with the increase of the number of virtual cores. Since the L3 cache is common to all cores (see Section 2.2), with the increase of the number virtual cores, the number of cache conflicts

also increases inducing more cache misses at this cache level.

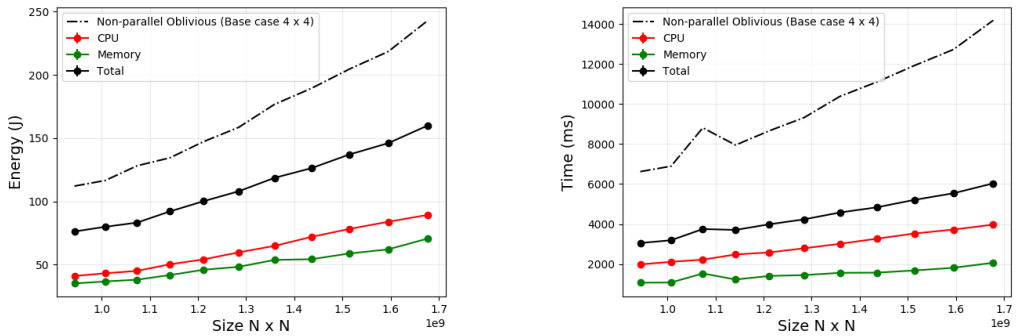
Cache-Oblivious Parallel Algorithm with a base case of 4×4

Figure 6.37, 6.38 and 6.39, show the performance of Cache-Oblivious Parallel Algorithm in terms of Energy (a) and Time (b) with respect to CPU instructions (green line) and memory access (red line) and total (black line), with a base case of 4×4 on two, four and eight virtual cores, respectively.[3] In addition, dashed lines represent the total energy consumed and running time by the Non-parallel Cache-Oblivious Algorithm with a base case of 4×4. Moreover, for each base case we performed a linear regression, where N^2 is the number of matrix elements.



(a) Energy (b) Time

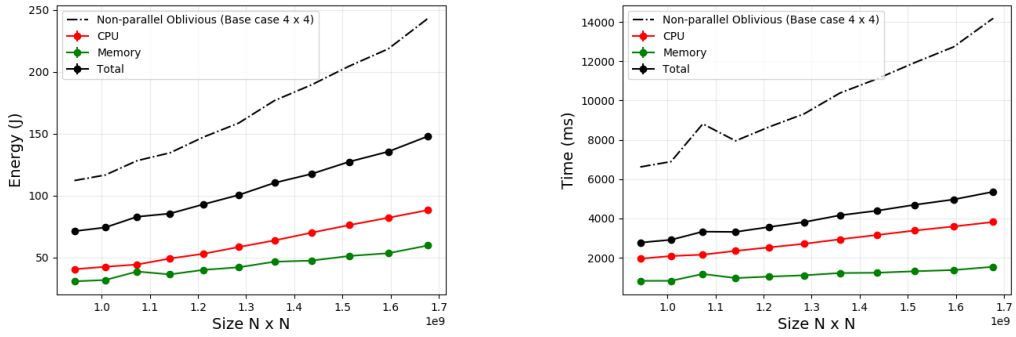
Figure 6.37: Performance of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on two virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

Figure 6.38: Performance of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on four virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses

[3] The figures on one, three, five, six and seven virtual cores are available in the Appendix C.



(a) Energy (b) Time

Figure 6.39: Performance of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on eight virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses

Linear regression models of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on two virtual cores (without the outlier at $N = 32768$):

$$E_{CPU} = -36.53 + 8.325e^{-08}N^2 \ (R^2 = 0.9873) \tag{6.16a}$$

$$T_{CPU} = -1435 + 397.3e^{-08}N^2 \ (R^2 = 0.9917) \tag{6.16b}$$

$$E_{Memory} = -14.33 + 5.736e^{-08}N^2 \ (R^2 = 0.9638) \tag{6.16c}$$

$$T_{Memory} = -440 + 234.9e^{-08}N^2 \ (R^2 = 0.9703) \tag{6.16d}$$

Linear regression models of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on four virtual cores (without the outlier at $N = 32768$):

$$E_{CPU} = -27.78 + 6.932e^{-08}N^2 \ (R^2 = 0.9889) \tag{6.17a}$$

$$T_{CPU} = -708.5 + 277e^{-08}N^2 \ (R^2 = 0.9946) \tag{6.17b}$$

$$E_{Memory} = -10.58 + 4.648e^{-08}N^2 \ (R^2 = 0.9786) \tag{6.17c}$$

$$T_{Memory} = -188 + 127.5e^{-08}N^2 \ (R^2 = 0.9625) \tag{6.17d}$$

Linear regression models of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on eight virtual cores (without the outlier at $N = 32768$):

$$E_{CPU} = -27.06 + 6.777e^{-08}N^2 \ (R^2 = 0.9877) \tag{6.18a}$$

$$T_{CPU} = -601.9 + 261.7e^{-08}N^2 \ (R^2 = 0.9947) \tag{6.18b}$$

$$E_{Memory} = -4.465 + 3.697e^{-08}N^2 \ (R^2 = 0.9686) \tag{6.18c}$$

$$T_{Memory} = -120 + 95.34e^{-08}N^2 \ (R^2 = 0.9811) \tag{6.18d}$$

The previous figures suggest that the number of CPU instructions has a strong influence on the both time and energy. Furthermore, analysing the models presented above, since R^2

values are above 0.9 and close to 1, we conclude that there is a linear relationship between the independent variable, N^2, and the dependent variables, E_{CPU}, E_{Memory}, T_{CPU} and T_{Memory}.

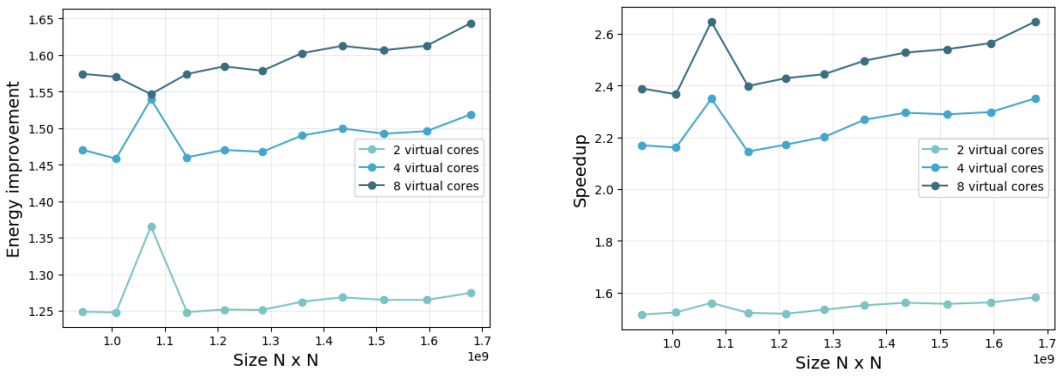
From the previous models, with the increase of the number of cores, the resource usage ratio of the CPU instructions component to the memory accesses component increases for both energy and time, although time presents a more noticeable increase. Considering two virtual cores, the CPU instructions component is approximately 1.45 times greater on energy and 1.69 times greater on time than the memory accesses component. Considering four virtual cores, the CPU instructions component is approximately 1.49 times greater on energy and 2.2 times greater on time than the memory accesses component. Finally, considering eight virtual cores, the CPU instructions component is approximately 1.71 times greater on energy and 2.75 times greater on time than the memory accesses component. The other resource usage ratios are presented in Table 6.4. In comparison with the Cache-Oblivious Algorithm with a base case of 4×4 models (see Equations 6.5), the dominant component remained, and the resource usage ratio between the components energy and time on energy remained similar, with some peaks, while on time increased.

Number of virtual cores	Dominant component	$\frac{E_{DominantComponent}}{E_{SubmissiveComponent}}$	$\frac{T_{DominantComponent}}{T_{SubmissiveComponent}}$
2	CPU instructions	1.45	1.69
3	CPU instructions	1.12	1.31
4	CPU instructions	1.49	2.2
5	CPU instructions	1.1	1.71
6	CPU instructions	1.1	1.76
7	CPU instructions	1.57	2.69
8	CPU instructions	1.71	2.75

Table 6.4: Cache-Oblivious Parallel Algorithm with a base case of 4×4 - Resource usage ratio of the dominant component to the submissive component

Upon analysing the change of the resource usage ratio between the components energy and time with the increase of the number of virtual cores, it is noticeable that the running time disparity between both components is higher than the disparity of energy consumed measurements. Specifically, the resource usage ratio between the components on energy increased from 1.45 to 1.71, with some valleys, and from 1.96 to 2.75 on time. This observation suggests that increasing the number of virtual cores has more impact on time than on energy. To understand more about the performance of the algorithm on energy consumed and running time with the increase of the number of virtual cores, we analyse the energy consumption of the algorithm considering different number of virtual cores and the correspondent achieved speedup. Note that, to keep this analysis succinct, we only discuss the results for two, four and eight virtual cores in this section. However, the results for other values, which can be seen in Appendix C, are coherent with these.

Figures 6.40 shows the energy consumed with the different number of virtual cores (a) and the achieved speedup (b) with a base case of 4×4.



(a) Energy improvement

(b) Speedup

Figure 6.40: Performance of Cache-Oblivious Parallel Algorithm with different number of cores in terms of Energy (a) and Speedup (b)

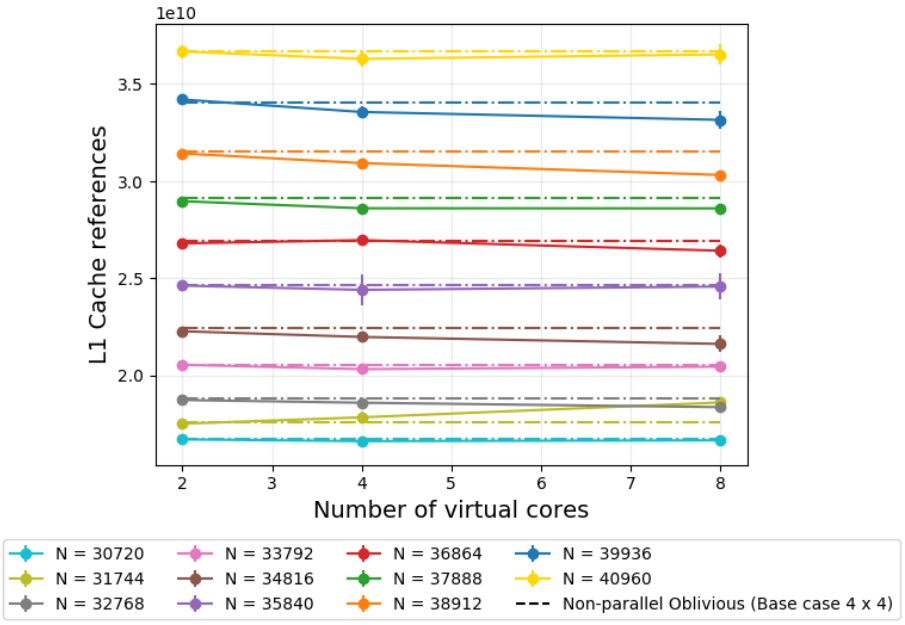
Figure 6.40a shows a slight energy improvement from one virtual core to two and from two virtual cores to four, but with more than four virtual cores the decrease is not so substantial. Moreover, Figure 6.40a shows that certain number of virtual cores achieve a better performance at higher values of N, and considering the algorithm with a base case of 1×1, this algorithm presents a more disturbed performance. Figure 6.40b shows a large speed up from one core to two. However, with more than four virtual cores the speedup is not so large. Moreover, Figure 6.40b also presents some peaks on the speedup, at $N = 32768$ (see Section 6.1 to a more detailed analysis of this particular size). Furthermore, the speedup achieved with a base case of 4×4 is inferior to the speedup achieved with a base case of 1×1. This difference can be due to the influence of the previous performance difference between both base cases (see Section 6.3.1) and the Amdahl's law (see Section 2.5).

Both Figures 6.40a and 6.40b suggest that the increase of the number of virtual cores affects the running time more than the energy consumed. However, the impact on the running time seems to be slightly larger than on energy consumed. As explained previously, the increase in the number of virtual cores causes the computer to activate more physical cores to execute the algorithm and consequently affects energy consumption. Therefore, since the number of physical cores used increases, it is expected that the energy consumed increases as well. For that reason, there is a slightly higher difference between the variation of the running time than the energy consumed.

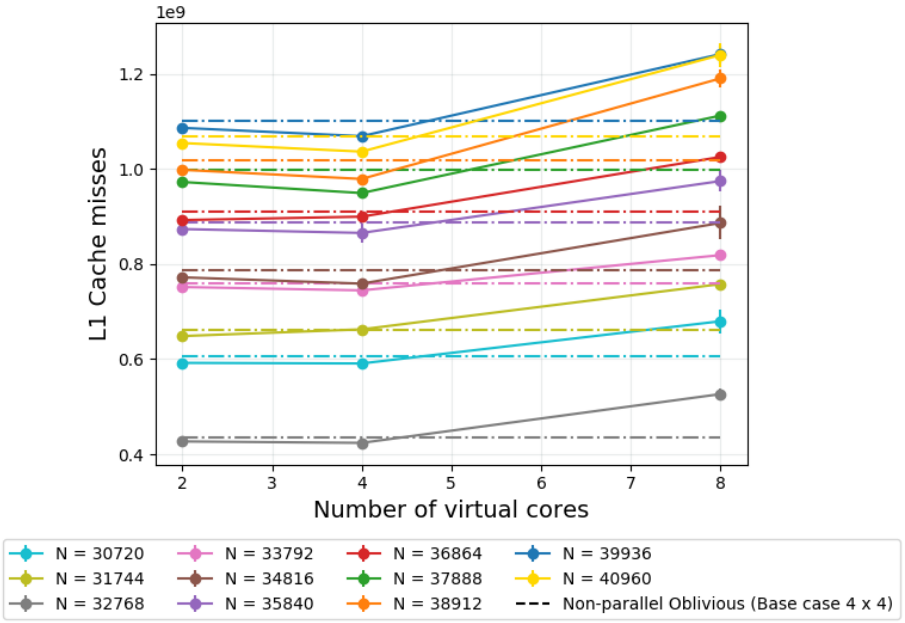
On another note, Figures 6.37, 6.38 and 6.39, show that the energy consumed and running time by the memory accesses component present some casual variances, up and downs, with the increase of the number of virtual cores. Therefore, this observation suggest that the cache performance presents some deterioration with the increase in the number of virtual cores. To understand the impact on cache performance, we analysed the number of cache references and misses for each number of virtual cores in each cache level.[4]

Figure 6.41 shows the number of cache references and misses (a) and cache misses (b) at L1 cache.

[4] Note that the cache performance's figures are presented on a larger scale for a better understanding of the reader.



(a) Cache references



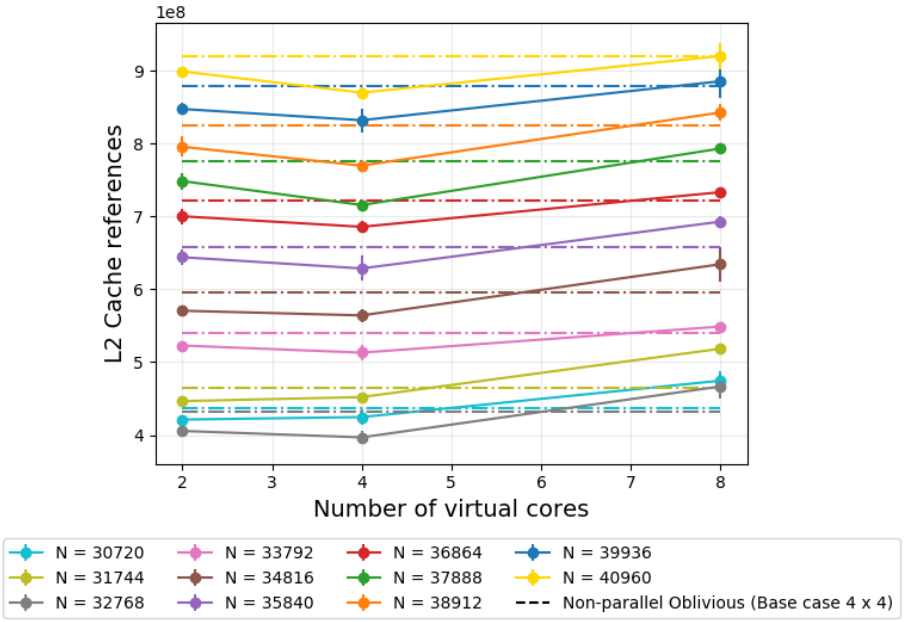
(b) Cache Misses

Figure 6.41: Cache-Oblivious Parallel Algorithm with different number of cores - L1 Cache references (a) and cache misses (b)

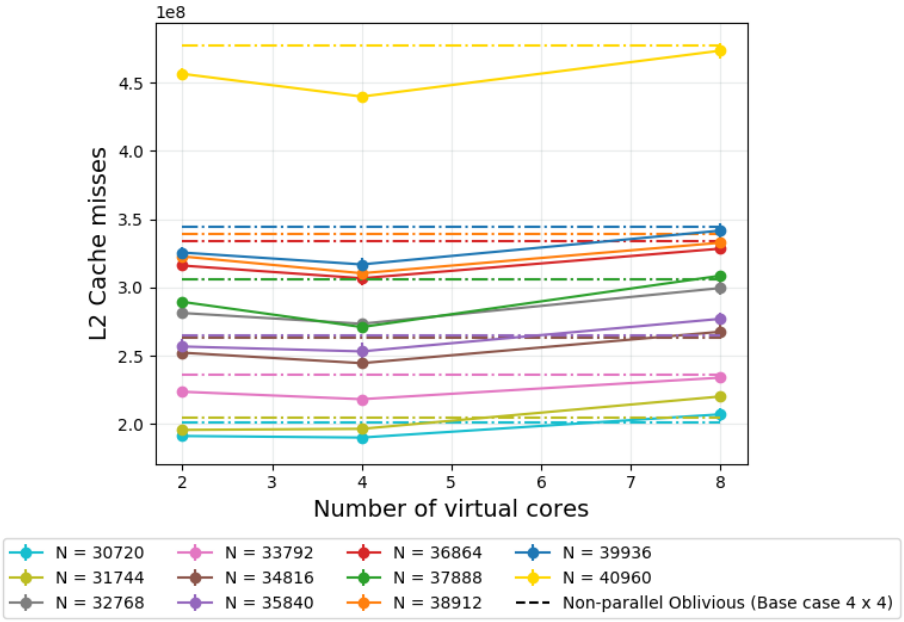
Figure 6.41a, in comparison with the Cache-Oblivious Algorithm, shows a higher number of references in the first values of N, although with the increase of N, the number of references decreases. However, analysing the number of cache misses in Figure 6.41b, we can notice that the number of cache misses surpasses the number of cache misses performed by the Cache-Oblivious Algorithm with the increase of the number of virtual cores in most values of N. These results suggest that from a certain number of virtual cores and with the increase of the value of N, the cache accesses performed by multiple cores conflict between each other at higher cache levels.

Figure 6.42 shows the number of cache references and misses (a) and cache misses (b) at

L2 cache.



(a) Cache references



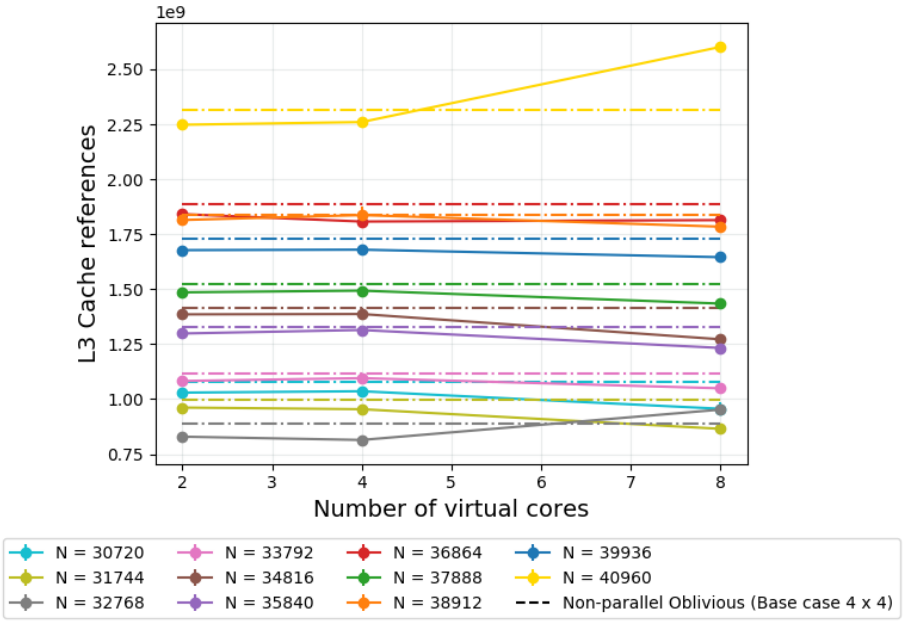
(b) Cache Misses

Figure 6.42: Cache-Oblivious Parallel Algorithm with different number of cores - L2 Cache references (a) and cache misses (b)

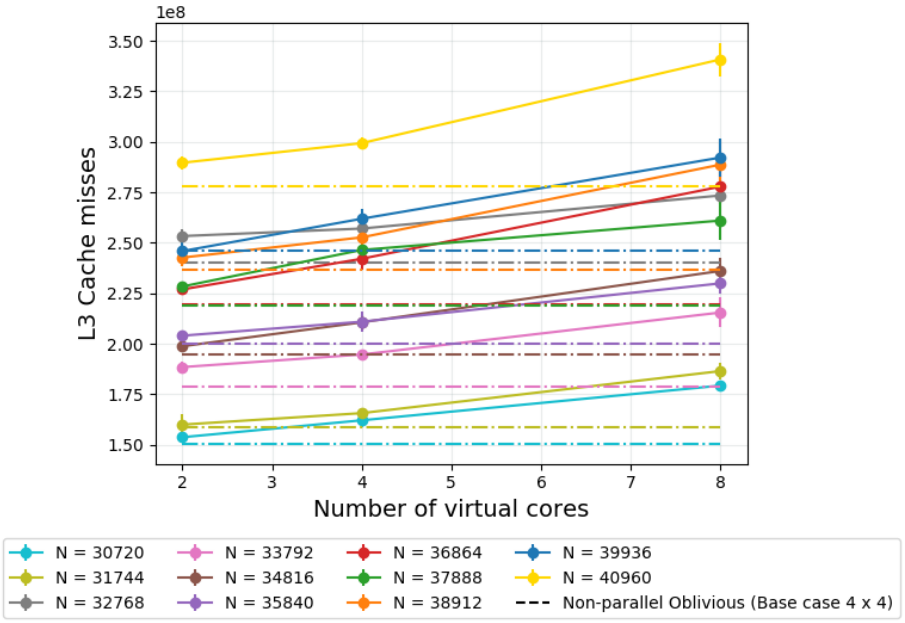
Figure 6.42a, in comparison with the Cache-Oblivious Algorithm, shows a decrease of the number of cache references until up to five virtual cores and an increase with more than five virtual cores. This effect is due to a similar behaviour observed in Figure 6.41b. In Figure 6.42b we can notice that the number of cache misses decreased with the increase of the number of virtual cores. Moreover, at $N = 32768$ and $N = 40960$, the number of cache misses presents a unusual behaviour. However, this behaviour seems to be related with the problematic values of N (see Section 6.1 to a more detailed analysis). Therefore, despite the previous of N, both figures suggest that the algorithm takes advantage from

the shared cache, L2 cache, between virtual cores common to the same physical core (see Section 2.2 and 2.5).

Figure 6.43 shows the number of cache references and misses (a) and cache misses (b) at L3 cache.



(a) Cache references



(b) Cache Misses

Figure 6.43: Cache-Oblivious Parallel Algorithm with different number of cores - L3 Cache references (a) and cache misses (b)

Figure 6.43a, in comparison with the Cache-Oblivious Algorithm, contrary to what as expected, shows higher number of cache references, which can be a consequence of the bad performance at the L2 cache. Moreover, Figure 6.43b also show an increase of the number of cache misses with the increase of the number of virtual cores. Since the L3 cache is

common to all cores (see Section 2.2), with the increase of the number virtual cores and the deterioration of the L2 cache performance, the number of cache conflicts also increases.

Comparing the Cache-Oblivious Parallel Algorithm with a base case of 1×1 with the Cache-Oblivious Parallel Algorithm with a base case of 4×4, we can notice that the largest base case presents the best performance results on energy consumed and running time. However, analysing the cache performance results, the smallest base case presents the better exploitation of the L1 and L2 caches on the multicore architecture.

As a conclusion, the Cache-Oblivious Parallel Algorithm presented better results than the Cache-Oblivious Algorithm and a notorious speed-up by increasing the number of virtual cores. Moreover, as expected, the CPU instructions component presented the highest difference of energy consumed in comparison with the Cache-Oblivious Algorithm, although the memory accesses component also presented a slightly decreased.

The second experiments were performed using the Multithreading technique (see Section 2.5) to achieve parallelization. However, analysing the obtained results we observed that increasing the number of threads at each virtual core, in this benchmark, does not bring any performance improvement. These results can be explained, not also because of the work overhead that is created in a single core but also because the fact of the cache deterioration performance. Since the threads in a single virtual core share the different cache levels, the access of threads to different data degrades the memory access pattern of the Cache-Oblivious Algorithm.

6.4.2 Cache-Aware Parallel Algorithm

The Cache-Aware Parallel Algorithm (see Section 4.4.1) is a parallel version of the Cache-Aware Algorithm. We recall that the Cache-Aware Algorithm performs the matrix transposition dividing the full matrix into submatrices. To accomplish parallelization, the parallel version receives as arguments the initial row, last row, initial column, and last column of the initial submatrices, each of which to be processed by a thread or a processor. The experiments below were conducted considering the parallelization of the matrix by columns, similar to the non-parallel algorithm procedure. We also considered the experiments with a parallelization of the matrix by rows. However the achieved results were quite similar.

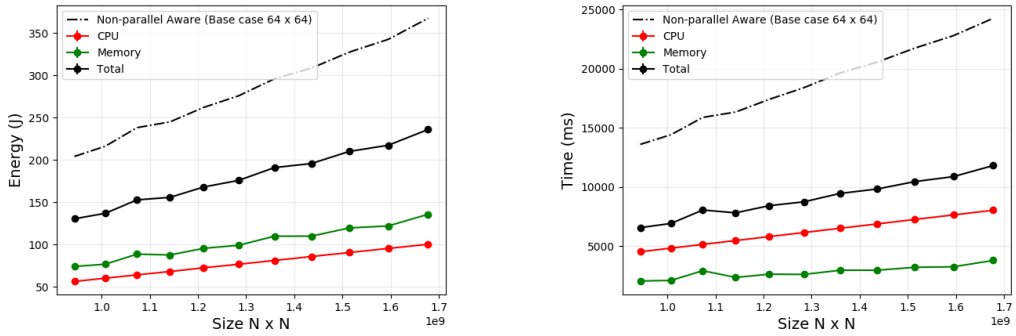
From Section 6.3.2, we observed that the Cache-Aware Algorithm with a block size of 64×64 presented the best algorithm performance in terms of energy consumed and running time. Therefore, to understand the performance of this algorithm with parallelization, we analyse its performance considering different numbers of virtual cores, $c \in \{1, 2, 3, 4, 5, 6, 7, 8\}$, and threads per core, $t \in \{2, 4, 8, \ldots, 1024\}$.

Similarly to the previous sections, we measured both running time and energy consumed by the CPU instructions and memory accesses of the algorithm. Moreover, the increase in the number of CPU instructions and memory accesses affects time and energy consumed in a linear increasing way. Since the performance results achieved with small values of N are not relevant and to facilitate the creation of different plots, we just considered the values of $N \in \{30720, 31744, 32768, \ldots, 40960\}$.

The first experiments analysed the performed of the Cache-Aware Parallel Algorithm using the Multicore technique (see Section 2.5) to achieve parallelization. Moreover, to achieve a multicore parallelization, each virtual core executed one single thread assigned using a Round-robin scheduling.

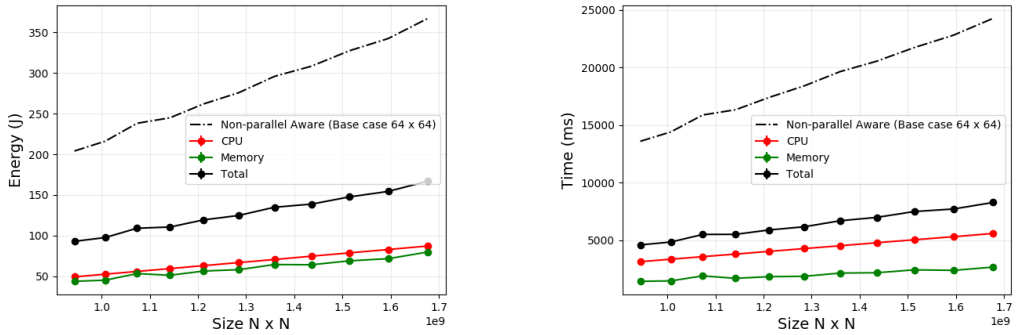
Figure 6.44, 6.45 and 6.46, show the performance of Cache-Aware Parallel Algorithm in

terms of Energy (a) and Time (b) with respect to CPU instructions (green line) and memory access (red line) and total (black line), with a block size of 64×64 on two, four and eight virtual cores, respectively.[5] In addition, dashed lines present the total energy consumed and running time by the Non-parallel Cache-Aware Algorithm with a block size of 64×64. Moreover, for each base case we performed a linear regression, where N^2 is the number of matrix elements.



(a) Energy (b) Time

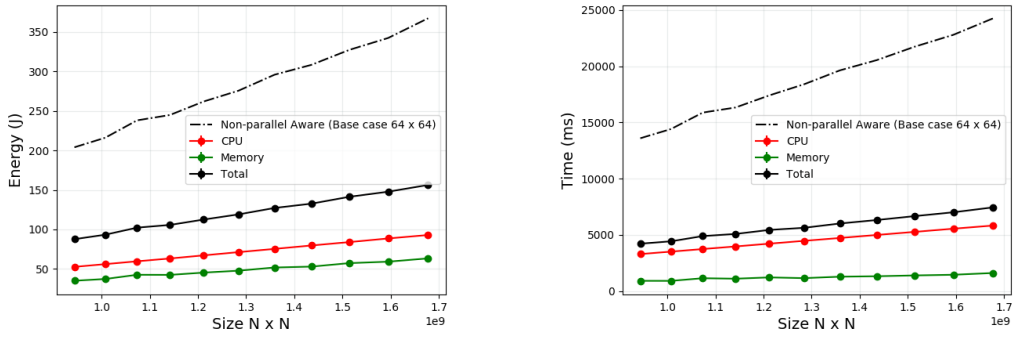
Figure 6.44: Performance of Cache-Aware Parallel Algorithm with a block size of 64×64 on two virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

Figure 6.45: Performance of Cache-Aware Parallel Algorithm with a block size of 64×64 on four virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses

[5] The figures on one, three, five, six and seven virtual cores are available in the Appendix D.



(a) Energy (b) Time

Figure 6.46: Performance of Cache-Aware Parallel Algorithm with a block size of 64×64 on two virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses

Linear regression models of Cache-Aware Parallel Algorithm with a block size of 64×64 on two virtual cores (without the outlier at $N = 32768$):

$$E_{CPU} = -0.1782 + 5.987e^{-08}N^2 \ (R^2 = 1) \tag{6.19a}$$

$$T_{CPU} = -0.043 + 478.3e^{-08}N^2 \ (R^2 = 1) \tag{6.19b}$$

$$E_{Memory} = -4.326 + 8.149e^{-08}N^2 \ (R^2 = 0.9842) \tag{6.19c}$$

$$T_{Memory} = -121 + 220.3e^{-08}N^2 \ (R^2 = 0.9564) \tag{6.19d}$$

Linear regression models of Cache-Aware Parallel Algorithm with a block size of 64×64 on two virtual cores (without the outlier at $N = 32768$):

$$E_{CPU} = 0.2874 + 5.176e^{-08}N^2 \ (R^2 = 1) \tag{6.20a}$$

$$T_{CPU} = 12.8 + 333.2e^{-08}N^2 \ (R^2 = 1) \tag{6.20b}$$

$$E_{Memory} = -1.858 + 4.719e^{-08}N^2 \ (R^2 = 0.98) \tag{6.20c}$$

$$T_{Memory} = -188.2 + 168.5e^{-08}N^2 \ (R^2 = 0.9747) \tag{6.20d}$$

Linear regression models of Cache-Aware Parallel Algorithm with a block size of 64×64 on two virtual cores (without the outlier at $N = 32768$):

$$E_{CPU} = 0.358 + 5.524e^{-08}N^2 \ (R^2 = 0.9999) \tag{6.21a}$$

$$T_{CPU} = 27.28 + 346.3e^{-08}N^2 \ (R^2 = 1) \tag{6.21b}$$

$$E_{Memory} = -1.112 + 3.825e^{-08}N^2 \ (R^2 = 0.9963) \tag{6.21c}$$

$$T_{Memory} = 53.33 + 90.29e^{-08}N^2 \ (R^2 = 0.9603) \tag{6.21d}$$

The previous figures suggest that the number of CPU instructions has a strong influence on the both time and energy. Furthermore, analysing the models presented above, since R^2

values are close to 1, we conclude that there is a linear relationship between the independent variable, N^2, and the dependent variables, E_{CPU}, E_{Memory}, T_{CPU} and T_{Memory}.

From the previous models, with the increase of the number of virtual cores, the resource usage ratio of the CPU instructions component to the memory accesses component increases for both energy and time, although time presents a more noticeable increase. Considering two virtual cores, the memory accesses component is approximately 1.36 times greater on energy and 2.17 times lower on time than the CPU instructions component. Considering four virtual cores, the CPU instructions component is approximately 1.1 times greater on energy and 1.98 times greater on time than the memory accesses component. Finally, considering eight virtual cores, the CPU instructions component is approximately 1.44 times greater on energy and 3.84 times higher on time than the memory accesses component. The other resource usage ratios are presented in Table 6.5. In comparison with the Cache-Aware Algorithm with a base case of 64×64 models (see Equations 6.10), the dominant component changed and the resource usage ratio between the components on energy are slightly inferior while on time they are largely superior.

Number of virtual cores	Dominant component	$\frac{E_{DominantComponent}}{E_{SubmissiveComponent}}$	$\frac{T_{DominantComponent}}{T_{SubmissiveComponent}}$
2	CPU instructions*	1.36	2.17
3	CPU instructions	1.1	2.19
4	CPU instructions	1.1	1.98
5	CPU instructions	1.1	2.62
6	CPU instructions	1.23	2.8
7	CPU instructions	1.34	3.21
8	CPU instructions	1.44	3.84

* Note that in this case both energy and time present different dominant components.

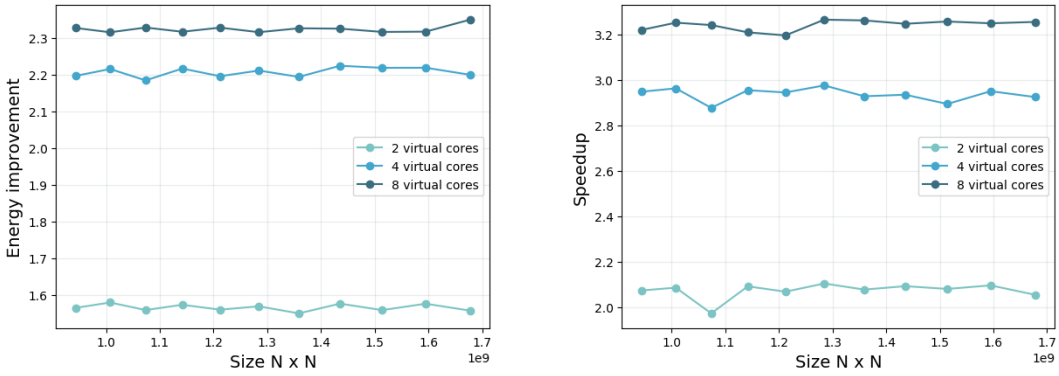
Table 6.5: Cache-Aware Parallel Algorithm with a block size of 64×64 - Resource usage ratio of the dominant component to the submissive component

Similarly to the Cache-Oblivious Parallel Algorithm (see Section 6.4.1), the previous figures show that with the increase of the number of virtual cores, energy and time decrease on both components. However, Figure 6.44, with two virtual cores, presents peculiar performance results. While in Figure 6.44b the dominant component are the CPU instructions, in Figure 6.44a, the dominant component are the memory accesses. Remembering the performance of the non-parallel Cache-Aware Algorithm (see Section 6.3.2), the dominant component were the memory accesses on energy and time. Therefore, these results with two cores suggest that, despite the parallelization in two cores and the increased performance on energy and time, the memory accesses component dominance prevailed, although with the increase of the number of cores this dominance disappears. Moreover, in comparison with the Cache-Oblivious Parallel Algorithm, the energy consumed by the memory accesses component decreased more substantially.

Upon analysing the change of the resource usage ratio between the components energy and time with the increase of the number of virtual cores, it is noticeable that the running time disparity between both components is higher than the disparity of energy consumed measurements. Specifically, the resource usage ratio between the components on energy remained approximately with the same value, with some valleys, but the on time increased from 2.17. to 3.84. This observation suggests that increasing the number of virtual cores has more impact on time than on energy. To understand more about the performance of

the algorithm on energy consumed and running time with the increase of the number of virtual cores, we analyse the energy consumption of the algorithm considering a different number of virtual cores and the correspondent achieved speedup. Note that, to keep this analysis succinct, we only discuss the results for two, four and eight virtual cores in this section. However, the results for other values, which can be seen in Appendix D, are coherent with these.

Figures 6.47 shows the energy consumed with the different number of virtual cores (a) and the achieved speedup (b) with a block size of 64×64.



(a) Energy improvement

(b) Speedup

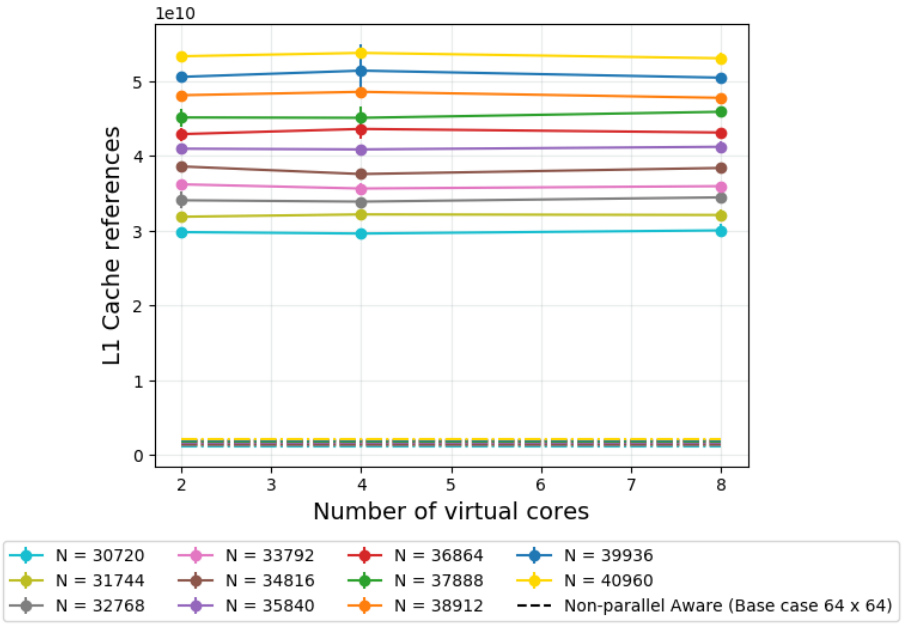
Figure 6.47: Performance of Cache-Aware Parallel Algorithm with different number of cores in terms of Energy (a), Energy Ratio (b) and Speedup (c)

Figure 6.47a shows a great energy improvement from one virtual core to two and from two virtual cores to four, but with more than four virtual cores the decrease is not so substantial. Figure 6.47b also shows a large speed up from one virtual core to two and from two virtual cores to four. However, with more than four virtual cores the speedup is not so large. Therefore, both figures suggest that the increase of the number of virtual cores affects the running time more than the energy consumed. As explained in the previous subsection, the increase in the number of virtual cores causes the computer to activate more physical cores to execute the algorithm and consequently affects energy consumption. For that reason, there is a slightly higher difference noticed between the variation of the running time than the energy consumed.

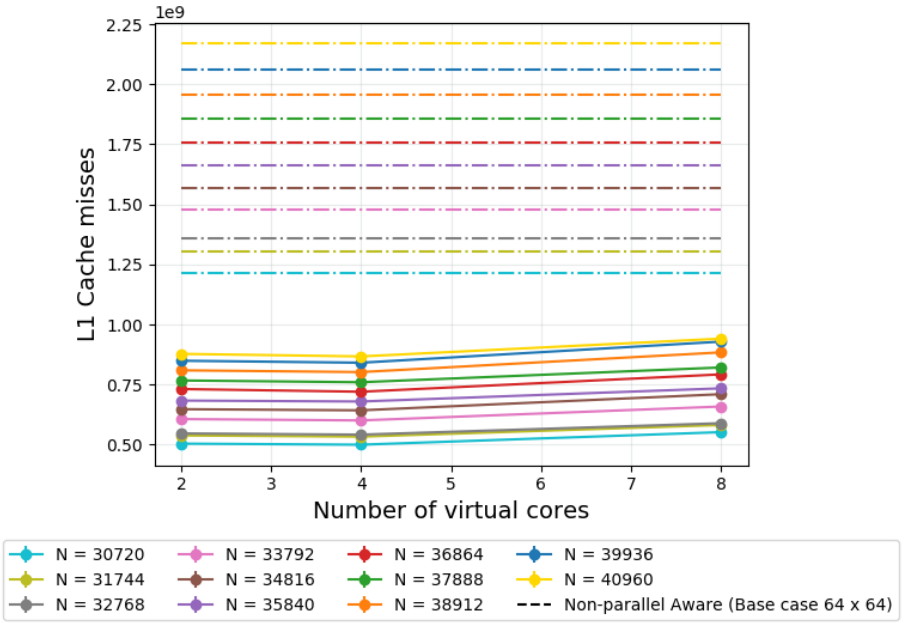
On another note, Figures 6.44, 6.45 and 6.46, show that the energy consumed and running time by the memory accesses component present some casual variances, up and downs, with the increase of the number of virtual cores. Therefore, this observation suggest that the cache performance presents some deterioration with the increase in the number of virtual cores. To understand the impact on cache performance, we analysed the number of cache references and misses for each number of virtual cores in each cache level.[6]

Figure 6.48 shows the number of cache references and misses (a) and cache misses (b) at L1 cache.

[6] Note that the cache performance's figures are presented on a larger scale for a better understanding of the reader.



(a) Cache references

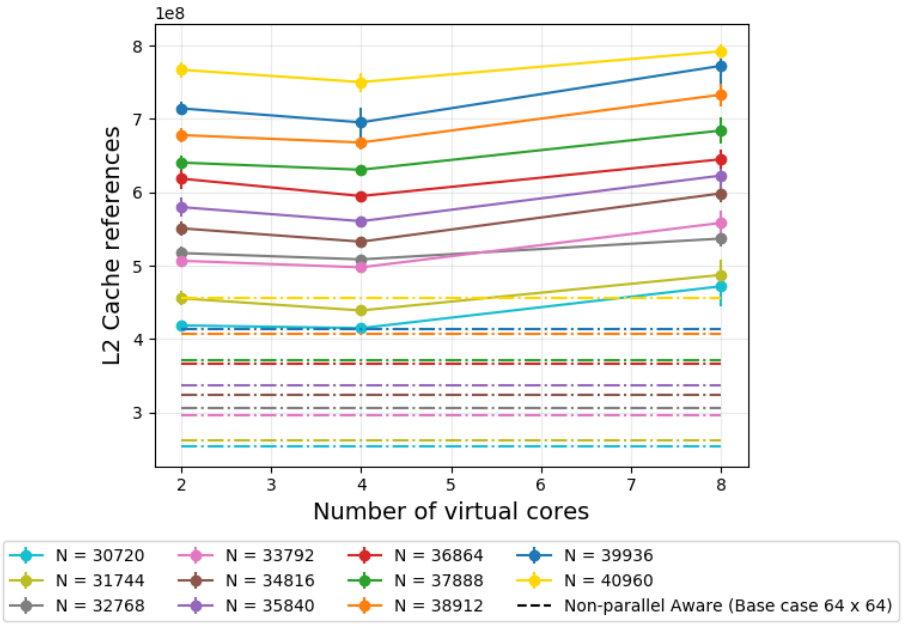


(b) Cache Misses

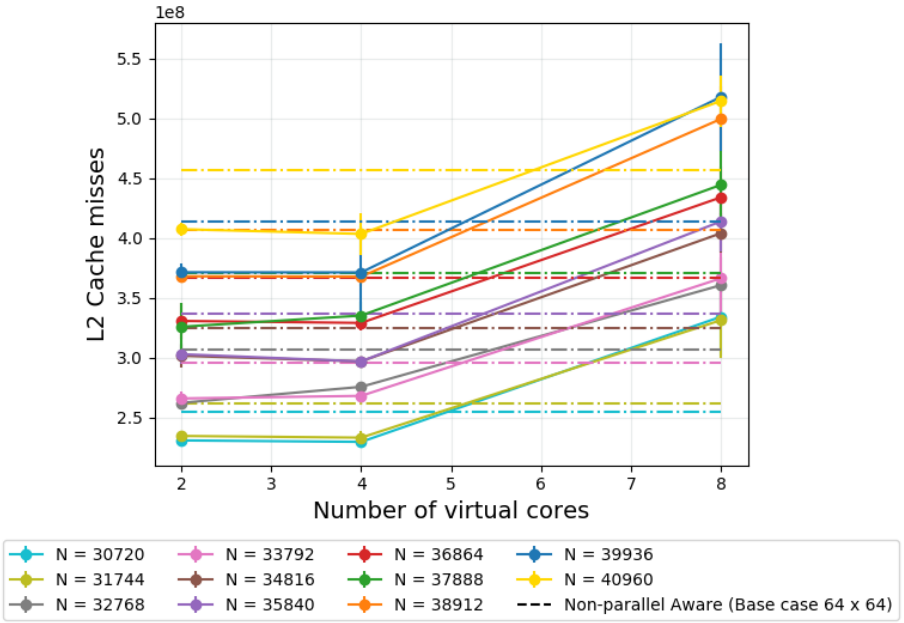
Figure 6.48: Cache-Aware Parallel Algorithm with different number of cores - L1 Cache references (a) and cache misses (b)

Figure 6.48a, in comparison with the Cache-Aware Algorithm, shows a largely higher number of references. However, analysing the number of cache misses in Figure 6.48b, we can notice a lower number of cache misses, although with a slightly increase in the number of cache misses with more than four virtual cores. These results suggest a high exploitation of the L1 cache.

Figure 6.49 shows the number of cache references and misses (a) and cache misses (b) at L2 cache.



(a) Cache references



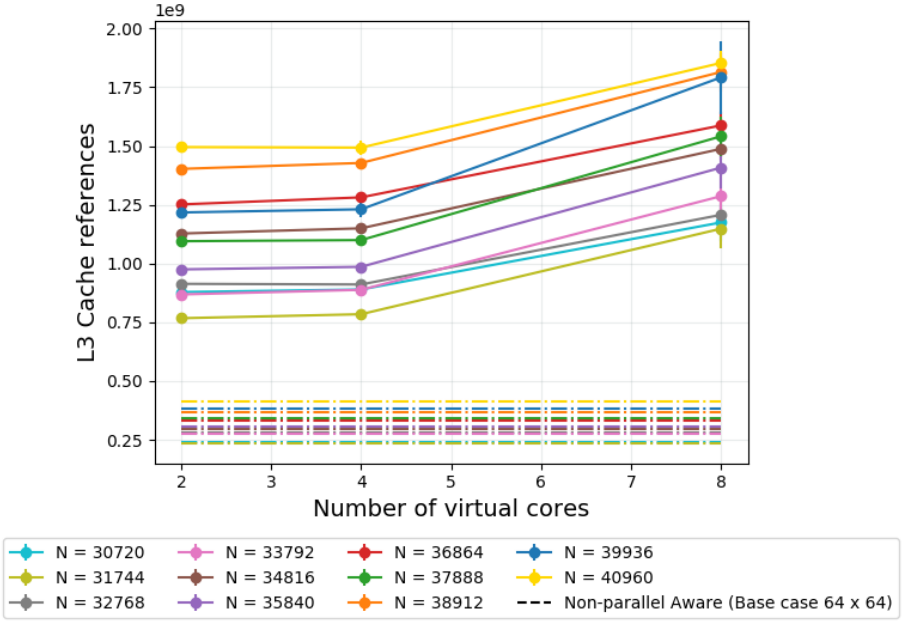
(b) Cache Misses

Figure 6.49: Cache-Aware Parallel Algorithm with different number of cores - L2 Cache references (a) and cache misses (b)

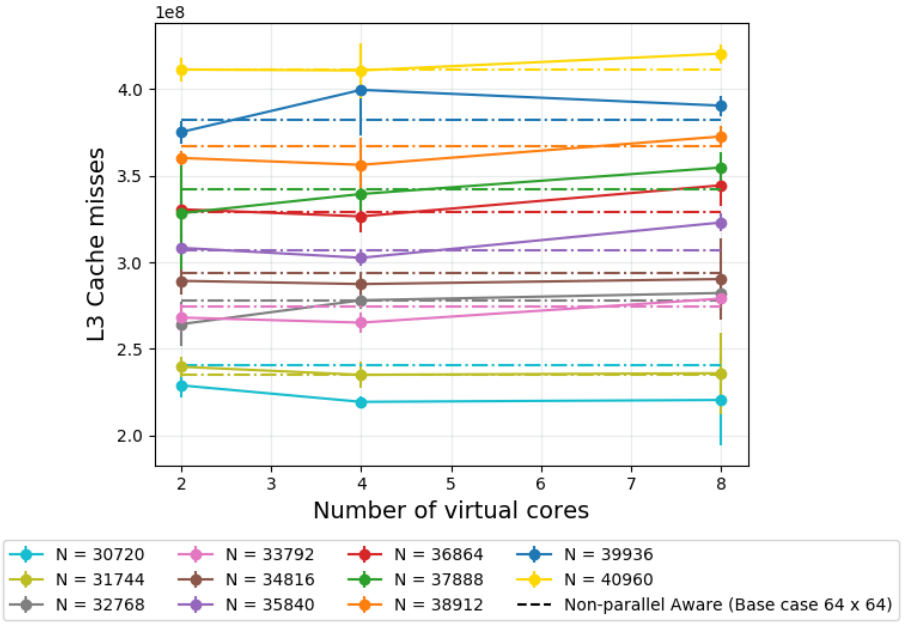
Figure 6.49a, similar to the L1 cache, in comparison with the Cache-Aware Algorithm, shows a largely higher number of references. However, analysing the number of cache misses in Figure 6.49b, we can notice that the number of cache misses surpasses the number of cache misses performed by the Cache-Aware Algorithm with the increase of the number of virtual cores in all values of N. Therefore, these results suggest that the algorithm takes advantage from the shared cache, L2 cache, although from a certain number of virtual cores, the cache accesses performed by multiple cores conflict between each other at higher cache levels.

Figure 6.50 shows the number of cache references and misses (a) and cache misses (b) at

L3 cache.



(a) Cache references



(b) Cache Misses

Figure 6.50: Cache-Aware Parallel Algorithm with different number of cores - L3 Cache references (a) and cache misses (b)

Figure 6.50a, similar to the L1 and L2 cache, in comparison with the Cache-Aware Algorithm, shows a largely higher number of references. However, analysing the number of cache misses in Figure 6.49b, we can notice that the number of cache misses surpasses the number of cache misses performed by the Cache-Aware Algorithm with the increase of the value of N with more than four virtual cores. Therefore, these results suggest that from a certain number of virtual cores and with the increase of the value of N, the cache accesses performed by multiple cores conflict between each other at higher cache levels. Moreover, since the L3 cache is common to all cores (see Section 2.2), with the increase of the number

virtual cores, the number of cache conflicts seems to increase.

As a conclusion, the Cache-Aware Parallel Algorithm presented better results than the Cache-Aware Algorithm and a notorious speed-up by increasing the number of virtual cores. In comparison with the Cache-Oblivious Parallel Algorithm, it presented similar energy and time performance values. These practical results comparison can be seen in the Appendix E. Moreover, as observed in the previous research question, both present different cache performances using different memory accesses patterns and, as expected, the CPU instructions component presented the highest difference of energy consumed in comparison with the Cache-Aware Algorithm.

The second experiments were performed using the Multithreading technique (see Section 2.5) to achieve parallelization. However, similarly to what has already been explained in the previous section, the obtained results revealed that increasing the number of threads at each virtual core, in this benchmark, does not bring any performance improvement.

This page is intentionally left blank.

Chapter 7

Conclusions and Future Work

In the present thesis, we studied the relation between the energy consumption of an algorithm, its running-time and cache performance considering different memory access patterns.

The Energy Complexity Model by Roy et al. [33] suggests that the total energy consumed by an algorithm can be modelled as a linear combination of the energy consumed by the CPU instructions, or non-memory accesses, and memory accesses. Therefore, taking this model suggestion into consideration, we measured the individual and total running time and energy consumed by two components: the CPU instructions and memory accesses of an algorithm.

In order to perform our analysis, we chose the Matrix Transposition operation because it is an operation mainly dependent on memory accesses that can be resolved using different memory access patterns. Moreover, we studied the behaviour of different Matrix Transposition algorithms in terms of energy, time and cache performance considering a single, multicore and multithreading perspective. In order to better understand, we considered four research questions considering four algorithms of the Matrix Transposition with different memory access patterns.

The first research question aimed to identify the impact of each component and how energy and time are correlated in the context of Matrix Transposition. Therefore, we performed an experimental analysis of the traditional matrix transposition algorithm, also called Naïve Algorithm. The results indicate that energy and time seem to be strongly correlated and that memory accesses, rather than CPU instructions, impact energy consumed and running time. Moreover, results show a large number of cache references and misses. This suggests that energy consumption can be related with the number of cache misses. Therefore, as memory accesses caused by cache misses can have this substantial impact, it is expected that energy consumption can be lowered by reducing the number of instructions and/or the number of memory accesses. However, in the context of the benchmark performed, this assumption cannot be proved.

For the second research question, we aimed to study the behaviour of an algorithm that takes into account how the main memory is organised and accessed. To do this we considered an algorithm proposed by Roy et al. [33] based on parallel memory models. The results indicate that the way the algorithm takes advantage of memory organisation can improve energy consumed and running time. Moreover, the collected cache performance results suggested that memory access patterns can also improve energy and time efficiency.

Regarding the third research question, in order to analyse different memory access patterns,

we considered two algorithms, Cache-Oblivious and Cache-Aware algorithms, and analysed their energy, time and cache performance. In comparison with the Naïve Algorithm, the Cache-Oblivious Algorithm presented better results. Moreover, contrarily to what was observed with the Naïve Algorithm, the dominant component, on both energy and time, was the CPU instructions component. The Cache-Aware Algorithm, in comparison with the Naïve Algorithm, also presented better results. However, contrarily to the Cache-Oblivious Algorithm and similar to the Naïve Algorithm, the dominant component in this algorithm was the memory accesses component. Since both algorithms presented energy and time performance improvements with different dominant components, we can remark that using different memory access patterns display different energy consumption models on both components.

As for the final research question, we conducted experiments to improve the running time of an algorithm using parallelization. For these experiments, we considered two parallel versions of the Cache-Oblivious and Cache-Aware algorithms. The results of each algorithm showed that with the increase in the number of virtual cores, the energy consumed and running time decreased. However, running time improved more than energy. These results suggest that the number of activated cores also influences energy consumption. Moreover, we can notice that both algorithms present an improved cache performance with the increase of the number of virtual cores up-to four. With more than four virtual cores the cache presents a certain deterioration caused by the cache conflicts between different threads on different cache levels.

All in all, the present thesis supports that the energy consumed by an algorithm can be modelled as a linear combination of the energy consumed by the CPU instructions and memory accesses and that different memory access patterns can perform distinct energy consumption behaviours. Moreover, it is important that an algorithm takes advantage of the cache to decrease the energy consumption of the memory accesses component, as it was demonstrated that this is an important factor to have into account. Furthermore, the parallelization is also a relevant technique to improve energy consumption and reduce running-time in the problem considered, however, some special attention is needed regarding the behaviour of energy consumption and cache performance with the increase of the number of cores used.

7.1 Future work

Our aim was to understand the relation between the energy consumption of an algorithm in practice, its running time and cache performance using different memory access patterns and different parallelization strategies. To accomplish that we analysed different benchmark problems, mainly dependent on memory accesses and cache performance, and that can be resolved using different memory access patterns. Eventually, we performed experiments with several algorithms for Matrix Transposition problem. However, other related linear algebra problems can be placed as a benchmark of our experiments. It would also be relevant, as future work, to analyse other linear algebra problems, such as Matrix Multiplication. Moreover, there are others Matrix Transposition algorithms, such as in-place algorithms, that could be relevant to analyse. However, these algorithms are more complex to implement and analyse.

It would also be relevant to perform a deep analysis on memory parameters, e.g. cache policies and cache sizes, since our experiments show that memory access patterns and organisation have a significant impact on energy consumption. Additionally, it could be

relevant to analyse other types of cache misses, such as overall cache misses and instructions cache misses, and consider different memory parameters.

Furthermore, our analysis focused on the relation between the energy consumption of an algorithm, its running time and cache performance considering different memory access patterns. However, nowadays architectures and processors allow us to access and write directly to RAM memory, bypassing the cache levels, using register instructions, also called non-temporal instructions. Moreover, there are also others instructions that use non-temporal instructions to perform vectorial operations, also called SIMD instructions. To understand how significant this analysis can be, we performed some preliminary experiences, which show that we can also achieve great performance results in terms of energy consumed and running time. Therefore, as future work, we can perform an analysis over the Matrix Transposition or similar benchmark problems to understand what impact these non-temporal and SIMD instructions have in terms of energy consumption.

This page is intentionally left blank.

Bibliography

[1] Young-Woo Kwon and Eli Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pages 170–179, 2013.

[2] Luiz André Barroso. The price of performance. *ACM Queue*, 3(7):48–53, 2005.

[3] Dieter Will. Head north for the data center gold rush. `https://blog.advaoptical.com/en/head-north-for-the-data-center-gold-rush`, March 2017. [Online; accessed 17-May-2019].

[4] Bimal K. Bose. Global energy scenario and impact of power electronics in 21st century. *IEEE Trans. Industrial Electronics*, 60(7):2638–2651, 2013.

[5] Irene Manotas, Christian Bird, Rui Zhang, David C. Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori L. Pollock, and James Clause. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 237–248, 2016.

[6] Gustavo Pinto, Francisco Soares-Neto, and Fernando Castor Filho. Refactoring for energy efficiency: A reflection on the state of the art. In *4th IEEE/ACM International Workshop on Green and Sustainable Software, GREENS 2015, Florence, Italy, May 18, 2015*, pages 29–35, 2015.

[7] Gustavo Pinto and Fernando Castor. Energy efficiency: a new concern for application software developers. *Commun. ACM*, 60(12):68–75, 2017.

[8] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 22–31, 2014.

[9] Fernando Castor Gustavo Pinto. Energy efficiency: A new concern for application software developers. `https://cacm.acm.org/magazines/2017/12/223044-energy-efficiency/fulltext`, December 2017. [Online; accessed 15-May-2019].

[10] Mark Halper. Supercomputing's super energy needs, and what to do about them. `https://cacm.acm.org/news/192296-supercomputings-super-energy-needs-and-what-to-do-about-them/fulltext`, September 2015. [Online; accessed 15-May-2019].

[11] Yao Guo, Pritish Narayanan, Mahmoud A. Bennaser, Saurabh Chheda, and Csaba Andras Moritz. Energy-efficient hardware data prefetching. *IEEE Trans. VLSI Syst.*, 19(2):250–263, 2011.

[12] Karan Aggarwal, Abram Hindle, and Eleni Stroulia. Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 311–320, 2015.

[13] Ding Li, Shuai Hao, Jiaping Gui, and William G. J. Halfond. An empirical study of the energy consumption of android applications. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 121–130, 2014.

[14] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, 2016.

[15] Alexandra Yates Kristen C. Accardi. Powertop user guide. `https://01.org/sites/default/files/page/powertop_users_guide_201412.pdf`, 2014. [Online; accessed 20-October-2018].

[16] Patrick Konsor. Intel® power gadget. `https://software.intel.com/en-us/articles/intel-power-gadget/`, 2014. [Online; accessed 20-October-2018].

[17] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. A validation of DRAM RAPL power measurements. In *Proceedings of the Second International Symposium on Memory Systems, MEMSYS 2016, Alexandria, VA, USA, October 3-6, 2016*, pages 455–470, 2016.

[18] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. RAPL: memory power estimation and capping. In *Proceedings of the 2010 International Symposium on Low Power Electronics and Design, 2010, Austin, Texas, USA, August 18-20, 2010*, pages 189–194, 2010.

[19] Tomofumi Yuki and Sanjay V. Rajopadhye. Folklore confirmed: Compiling for speed = compiling for energy. In *Languages and Compilers for Parallel Computing - 26th International Workshop, LCPC 2013, San Jose, CA, USA, September 25-27, 2013. Revised Selected Papers*, pages 169–184, 2013.

[20] Susanne Albers. Energy-efficient algorithms. `https://cacm.acm.org/magazines/2010/5/87271-energy-efficient-algorithms/fulltext`, May 2010. [Online; accessed 19-May-2019].

[21] Mohammad Rashid, Luca Ardito, and Marco Torchiano. Energy consumption analysis of algorithms implementations. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2015, Beijing, China, October 22-23, 2015*, pages 82–85, 2015.

[22] Soontae Kim, Narayanan Vijaykrishnan, Mahmut T. Kandemir, Anand Sivasubramaniam, and Mary Jane Irwin. Partitioned instruction cache architecture for energy efficiency. *ACM Trans. Embedded Comput. Syst.*, 2(2):163–185, 2003.

[23] Michele Co, Dee A. B. Weikle, and Kevin Skadron. Evaluating trace cache energy efficiency. *TACO*, 3(4):450–476, 2006.

[24] Kenan Liu, Gustavo Pinto, and Yu David Liu. Data-oriented characterization of application-level energy optimization. In *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 316–331, 2015.

[25] Luis Gabriel Lima, Francisco Soares-Neto, Paulo Lieuthier, Fernando Castor, Gilberto Melfe, and João Paulo Fernandes. Haskell in green land: Analyzing the energy behavior of a purely functional language. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 517–528, 2016.

[26] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 256–267, 2017.

[27] Cagri Sahin, Lori L. Pollock, and James Clause. How do code refactorings affect energy usage? In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, pages 36:1–36:10, 2014.

[28] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. On the impact of code smells on the energy consumption of mobile applications. *Information & Software Technology*, 105:43–55, 2019.

[29] Ding Li and William G. J. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software, GREENS 2014, Hyderabad, India, June 1, 2014*, pages 46–53, 2014.

[30] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 225–236, 2016.

[31] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. The influence of the java collection framework on overall energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software, GREENS@ICSE 2016, Austin, Texas, USA, May 16, 2016*, pages 15–21, 2016.

[32] Gilberto Melfe, Alcides Fonseca, and João Paulo Fernandes. Helping developers write energy efficient haskell through a data-structure evaluation. In *Proceedings of the 6th International Workshop on Green and Sustainable Software, GREENS@ICSE 2018, Gothenburg, Sweden, May 27, 2018*, pages 9–15, 2018.

[33] Swapnoneel Roy, Atri Rudra, and Akshat Verma. An energy complexity model for algorithms. In *Innovations in Theoretical Computer Science, ITCS '13, Berkeley, CA, USA, January 9-12, 2013*, pages 283–304, 2013.

[34] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[35] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1):4:1–4:22, 2012.

[36] Piyush Kumar. Cache oblivious algorithms. In *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, pages 193–212, 2002.

[37] D. Tsifakis, Alistair P. Rendell, and Peter E. Strazdins. Cache oblivious matrix transposition: Simulation and experiment. In *Computational Science - ICCS 2004, 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part II*, pages 17–25, 2004.

[38] Daniel Etiemble. 45-year CPU evolution: one law and two equations. *CoRR*, abs/1803.00254, 2018.

[39] Saravanan Vijayalakshmi, Senthil Chandran, Sasikumar Punnekkat, and Dwarkadas Kothari. A study on factors influencing power consumption in multithreaded and multicore cpus. *WSEAS Transactions on Computers*, 10, 03 2011.

[40] Robert Basmadjian and Hermann de Meer. Evaluating and modeling power consumption of multi-core processors. In *Proceedings of the 3rd International Conference on Energy-Efficient Computing and Networking, e-Energy'12, Madrid, Spain, May 9-11, 2012*, page 12, 2012.

[41] Chengling Tseng and Silvia Figueira. An analysis of the energy efficiency of multithreading on multi-core machines. In *International Green Computing Conference 2010, Chicago, IL, USA, 15-18 August 2010*, pages 283–290, 2010.

[42] Yumna Zahid, Hina Khurshid, and Zulfiqar Ali Memon. On improving efficiency and utilization of last level cache in multicore systems. *ITC*, 47(3):588–607, 2018.

[43] Alvaro Tzul. Multicore architecture and cache optimization techniques for solving graph problems. *CoRR*, abs/1807.03383, 2018.

[44] Mathias Jacquelin, Loris Marchal, and Yves Robert. Complexity analysis and performance evaluation of matrix product on multicore architectures. In *ICPP 2009, International Conference on Parallel Processing, Vienna, Austria, 22-25 September 2009*, pages 196–203, 2009.

[45] Guy E. Blelloch, Rezaul Alam Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 501–510, 2008.

[46] Song Ho Ahn. Opengl transformation. `https://www.songho.ca/opengl/gl_transform.html`, 2018. [Online; accessed 14-January-2019].

[47] Richard E. Twogood and Michael P. Ekstrom. An extension of eklundh's matrix transposition algorithm and its application in digital image processing. *IEEE Trans. Computers*, 25(9):950–952, 1976.

[48] Sheng Ma, Yuanwu Lei, Libo Huang, and Zhiying Wang. MT-DMA: A DMA controller supporting efficient matrix transposition for digital signal processing. *IEEE Access*, 7:5808–5818, 2019.

[49] Alex Shinsel. Understanding the instruction pipeline. `https://techdecoded.intel.io/resources/understanding-the-instruction-pipeline/#gs.j3ht9u`. [Online; accessed 24-May-2019].

[50] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA*, pages 73–80, 1972.

[51] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, pages 114–118, 1978.

[52] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[53] Lars Arge, Michael T. Goodrich, Michael J. Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*, pages 197–206, 2008.

[54] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 285–298, 1999.

[55] Alexander Sandler. Aligned vs. unaligned memory access. `http://www.alexonlinux.com/aligned-vs-unaligned-memory-access`, 2008. [Online; accessed 15-December-2018].

[56] Wikipedia. Data structure alignment. `https://en.wikipedia.org/wiki/Data_structure_alignment`. [Online; accessed 20-October-2018].

[57] Ulrich Drepper. What every programmer should know about memory, part 1. `https://lwn.net/Articles/250967/`, 2007. [Online; accessed 20-October-2018].

[58] Agner Fog. Optimizing software in c++. `https://www.agner.org/optimize/optimizing_cpp.pdf`, 2004-2018. [Online; accessed 15-December-2018].

[59] Armin Größlinger. Some experiments on tiling loop programs for shared-memory multicore architectures. In *Programming Models for Ubiquitous Parallelism, 02.09. - 07.09.2007*, 2007.

[60] Saeed Parsa and Mohammad Hamzei. Nested-loops tiling for parallelization and locality optimization. *Computing and Informatics*, 36(3):566–596, 2017.

[61] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[62] Stefanos Kaxiras and Margaret Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2008.

[63] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, pages 483–485, 1967.

[64] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory II: hierarchical multilevel memories. *Algorithmica*, 12(2/3):148–169, 1994.

[65] Swapnoneel Roy, Atri Rudra, and Akshat Verma. Energy aware algorithmic engineering. In *IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems, MASCOTS 2014, Paris, France, September 9-11, 2014*, pages 321–330, 2014.

[66] Henry Cook, Miquel Moretó, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 308–319, 2013.

[67] Krisztián Flautner, Nam Sung Kim, Steven M. Martin, David T. Blaauw, and Trevor N. Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *29th International Symposium on Computer Architecture (ISCA 2002), 25-29 May 2002, Anchorage, AK, USA*, pages 148–157, 2002.

[68] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA 2001, Göteborg, Sweden, June 30-July 4, 2001*, pages 240–251, 2001.

[69] Intel. Intel® 64 and ia-32 architectures - software developer's manual complete. `https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4`, May 2019. [Online; accessed 10-May-2019].

[70] Adhemerval Zanella Netto and Ryan S. Arnold. Evaluate performance for linux on power. `https://www.ibm.com/developerworks/linux/library/l-evaluatelinuxonpower/`, June 2012. [Online; accessed 15-April-2019].

[71] Rik van Riel. Documentation for the sysctl files in /proc/sys/kernel/. `https://www.kernel.org/doc/Documentation/sysctl/kernel.txt`, 1999. [Online; accessed 25-October-2018].

[72] Brendan D. Gregg. perf examples. `http://www.brendangregg.com/perf.html#Events`. [Online; accessed 25-October-2018].

[73] Michael Kerrisk. Linux programmer's manual - perf_event_open. `http://www.man7.org/linux/man-pages/man2/perf_event_open.2.html`, March 2019. [Online; accessed 10-March-2019].

[74] Intel. Intel® 64 and ia-32 architectures - software developer's manual. `https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-volume-3b-system-programming-guide-part-2`, May 2019. [Online; accessed 10-May-2019].

[75] Perf - wall clock time. `https://github.com/torvalds/linux/blob/master/tools/perf/builtin-stat.c`. [Online; accessed 3-February-2019].

[76] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 89–100, 2007.

[77] Valgrind documentation. `http://valgrind.org/docs/manual/manual-intro.html#manual-intro.overview`. [Online; accessed 15-October-2018].

[78] Rohit Chandra. *Parallel programming in openMP*. Morgan Kaufmann, 2001.

[79] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[80] Michael V. Intel® threading building blocks, openmp, or native threads? `https://software.intel.com/en-us/intel-threading-building-blocks-openmp-or-native-threads`, September 2011. [Online; accessed 2-January-2019].

[81] Marilyn Wolf. Cache conflict. `https://www.sciencedirect.com/topics/computer-science/cache-conflict`. [Online; accessed 13-February-2019].

This page is intentionally left blank.

Appendices

This page is intentionally left blank.

Appendix A

Perf events

Perf available and gathered events on the Skylake architecture:

- Energy values:
  - *power/energy-pkg/*: energy consumed by the processor package, including the energy consumed by the GPU and cores, therefore, $Energy\_GPU + Energy\_Cores \leq Energy\_Package$;
  - *power/energy-ram/*: energy consumed by the RAM;
  - *power/energy-gpu/*: energy consumed by the GPU;
  - *power/energy-cores/*: energy consumed by the processors components;
- L1 Cache:
  - *L1-dcache-loads*: data loads from the L1 cache;
  - *L1-dcache-loads-misses*: data load misses from the L1 cache;
- L2 Cache:
  - *l2_rqsts.references*: L2 cache references;
  - *l2_rqsts.miss*: requests that miss L2 cache;
  - *l2_rqsts.all_demand_references*: demand requests to L2 cache;
  - *l2_rqsts.all_demand_miss*: demand data requests that miss L2 cache;
  - *l2_rqsts.all_demand_data_rd*: demand data read requests to the L2 cache;
  - *l2_rqsts.demand_data_rd_miss*: demand data read requests that missed L2 cache;
  - *l2_rqsts.demand_data_rd_hit*: demand data read requests that hit L2 cache;
  - *l2_rqsts.all_code_rd*: L2 cache code requests;
  - *l2_rqsts.code_rd_miss*: L2 cache misses when fetching instructions;
  - *l2_rqsts.all_rfo*: RFO requests to L2 cache;
  - *l2_rqsts.rfo_miss*: RFO requests that miss L2 cache;
  - *l2_rqsts.rfo_hit*: RFO requests that hit L2 cache;
- L3 Cache:

  - *longest_lat_cache.reference*: Counts core-originated cacheable requests to the L3 cache (including data and code reads, RFOs and prefetches);
  - *longest_lat_cache.miss*: Counts core-originated cacheable requests that miss the L3 cache (including data and code reads, RFOs and prefetches);
  - *offcore_requests.demand_data_rd*: demand data read requests sent to uncore;
  - *offcore_requests.l3_miss_demand_data_rd*: demand data read requests that missed L3;
  - *LLC-loads*: data loads from the LLC cache;
  - *LLC-loads-misses*: data load misses from the LLC cache;
  - *LLC-stores*: data stores to the LLC cache;
  - *LLC-stores-misses*: data store misses to the LLC cache;

- Others:
  - *cycles*: CPU cycles;
  - *instructions*: instructions executed;
  - *cycle_activity.cycles_l1d_miss*: cycles while L1 data cache miss demand load is outstanding;
  - *cycle_activity.cycles_l2_miss*: cycles while L2 data cache miss demand load is outstanding;
  - *cycle_activity.cycles_l3_miss*: cycles while L3 data cache miss demand load is outstanding;

After gathering all the previous presented events, we can derive them into more readable meanings (Note: this values are an estimation!):

- L1 Cache Misses = *L1-dcache-loads-misses* + *l2_rqsts.all_code_rd*;
- L1 Data Misses = *L1-dcache-loads-misses*;
- L1 Data Accesses = *L1-dcache-loads*;
- L1 Data Load Misses = *l2_rqsts.all_demand_data_rd*;
- L1 Data Store Misses = *l2_rqsts.all_rfo*;
- L2 Cache Misses = *longest_lat_cache.reference*;
- L2 Data Misses = *l2_rqsts.all_demand_miss*;
- L2 Data Load Misses = *l2_rqsts.demand_data_rd_miss*;
- L2 Data Store Misses = *l2_rqsts.rfo_miss*;
- L2 Data Cache Accesses = *l2_rqsts.all_demand_references*;
- L2 Data Cache Reads = *l2_rqsts.all_demand_data_rd*;
- L2 Total Reads = *l2_rqsts.all_demand_data_rd* + *l2_rqsts.all_code_rd*;
- L2 Total Accesses = *l2_rqsts.all_demand_references* + *l2_rqsts.all_code_rd*;
- L3 Cache Misses = *longest_lat_cache.miss*;

- L3 Data Misses = *LLC-loads-misses* + *LLC-stores-misses*;
- L3 Loads = *LLC-loads*;
- L3 Load Misses = *LLC-loads-misses*;
- L3 Stores = *LLC-stores*;
- L3 Store Misses = *LLC-stores-misses*;
- L3 Data Accesses = *longest_lat_cache.reference* - *l2_rqsts.code_rd_miss*;
- L3 Data Reads = *offcore_requests.demand_data_rd*;
- L3 Total Reads = *longest_lat_cache.reference* - *l2_rqsts.rfo_hit*;
- L3 Total Accesses = *longest_lat_cache.reference*;

This page is intentionally left blank.

Appendix B

Blocked Transpose Algorithm vs Cache-Oblivious Algorithm vs Cache-Aware Algorithm

B.1 Energy and Time

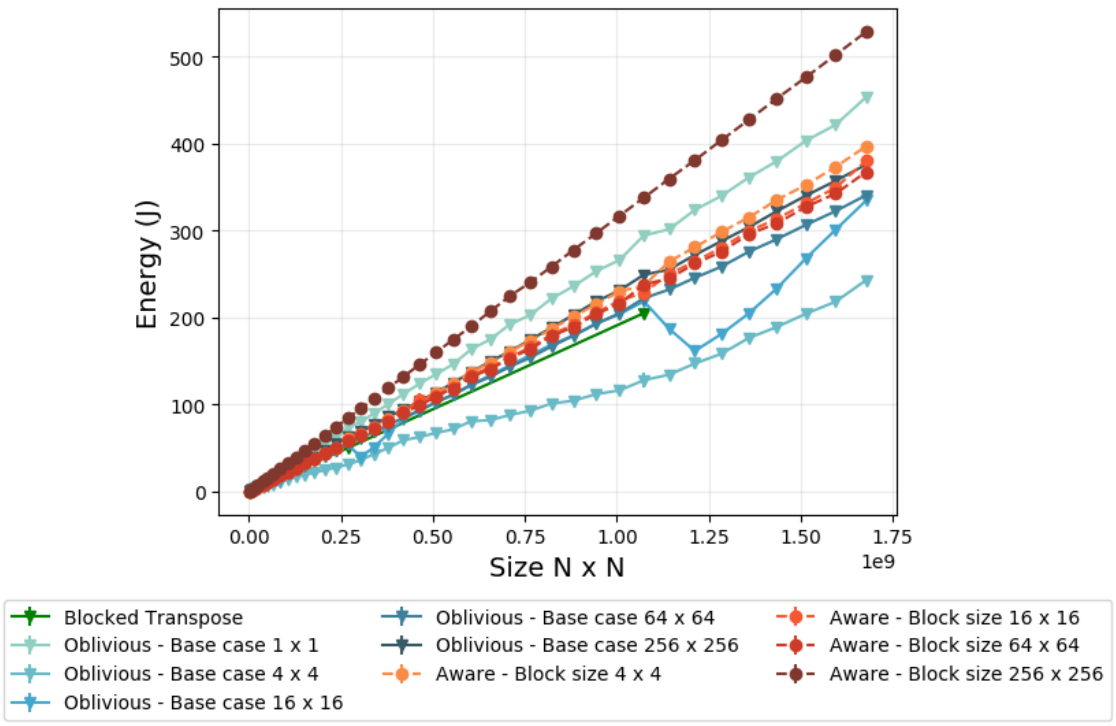


Figure B.1: Performance of Blocked Transpose, Cache-Oblivious and Cache-Aware algorithms in terms of Energy

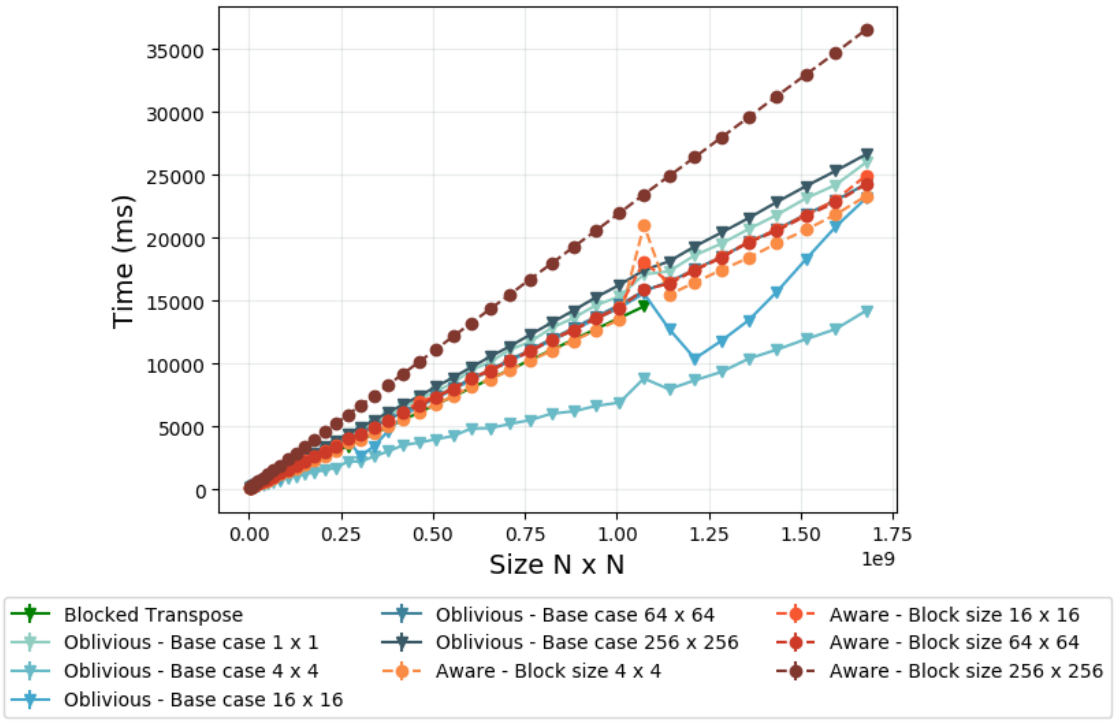


Figure B.2: Performance of Blocked Transpose, Cache-Oblivious and Cache-Aware algorithms in terms of Time

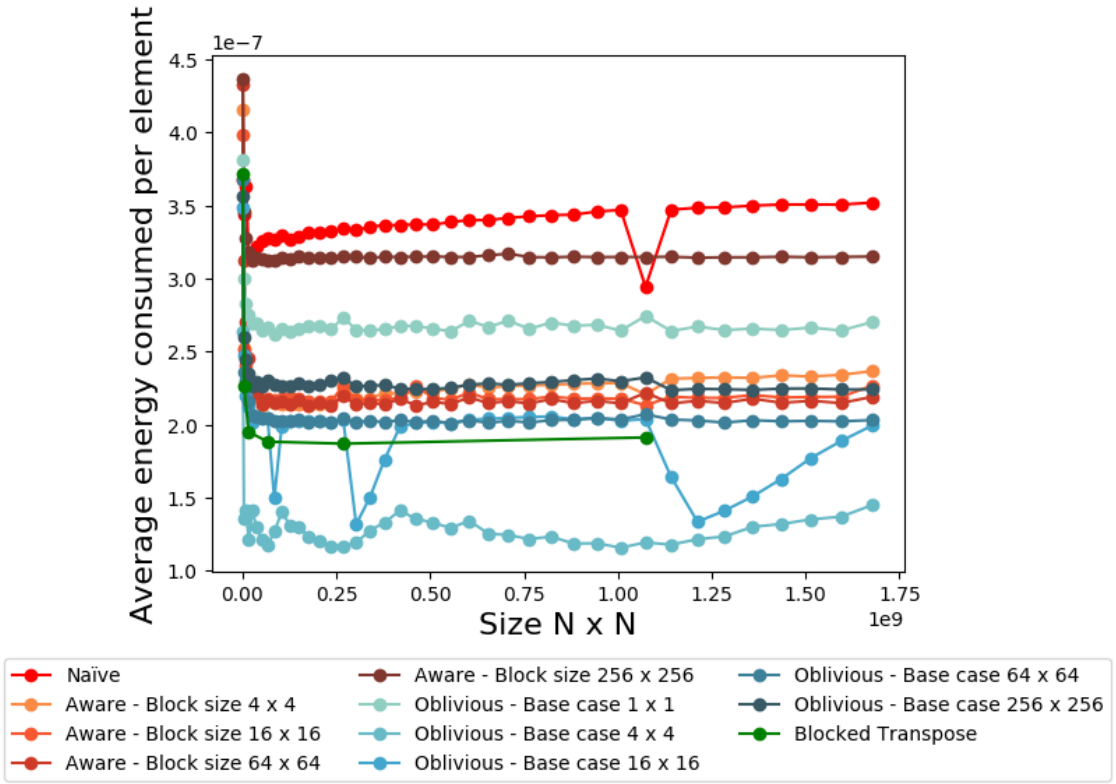


Figure B.3: Average energy consumed per matrix element of Naïve, Blocked Transpose, Cache-Oblivious and Cache-Aware algorithms

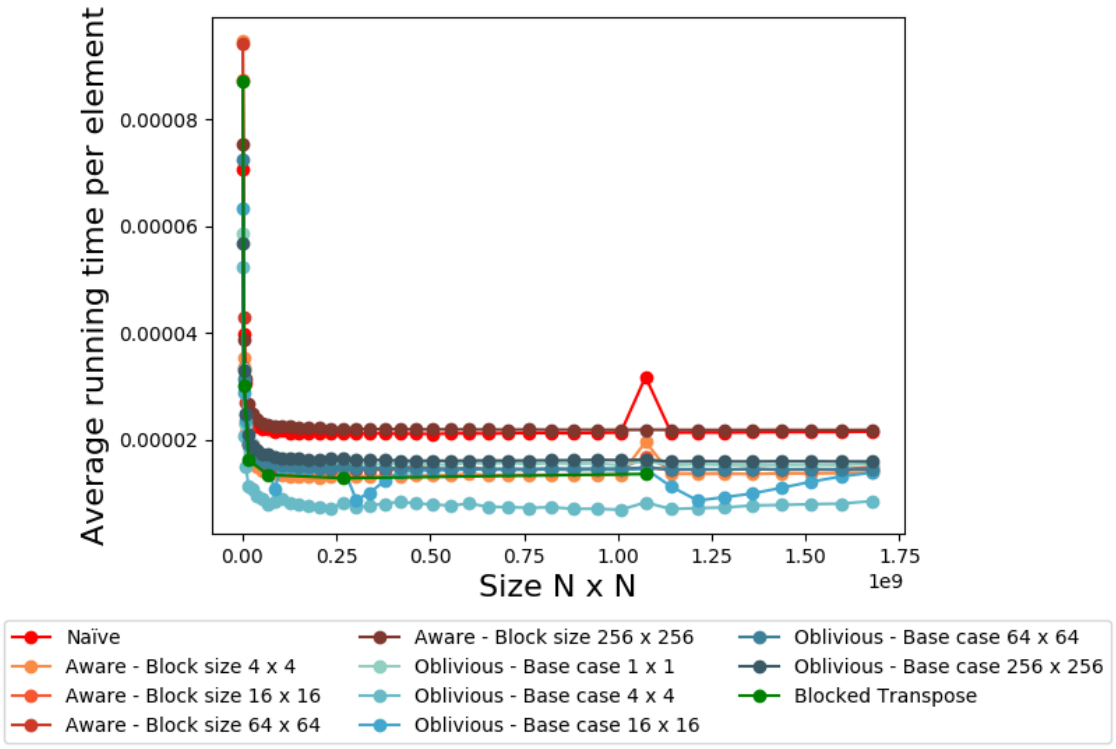


Figure B.4: Average running time per matrix element of Naïve, Blocked Transpose, Cache-Oblivious and Cache-Aware algorithms

B.2 Cache references and misses

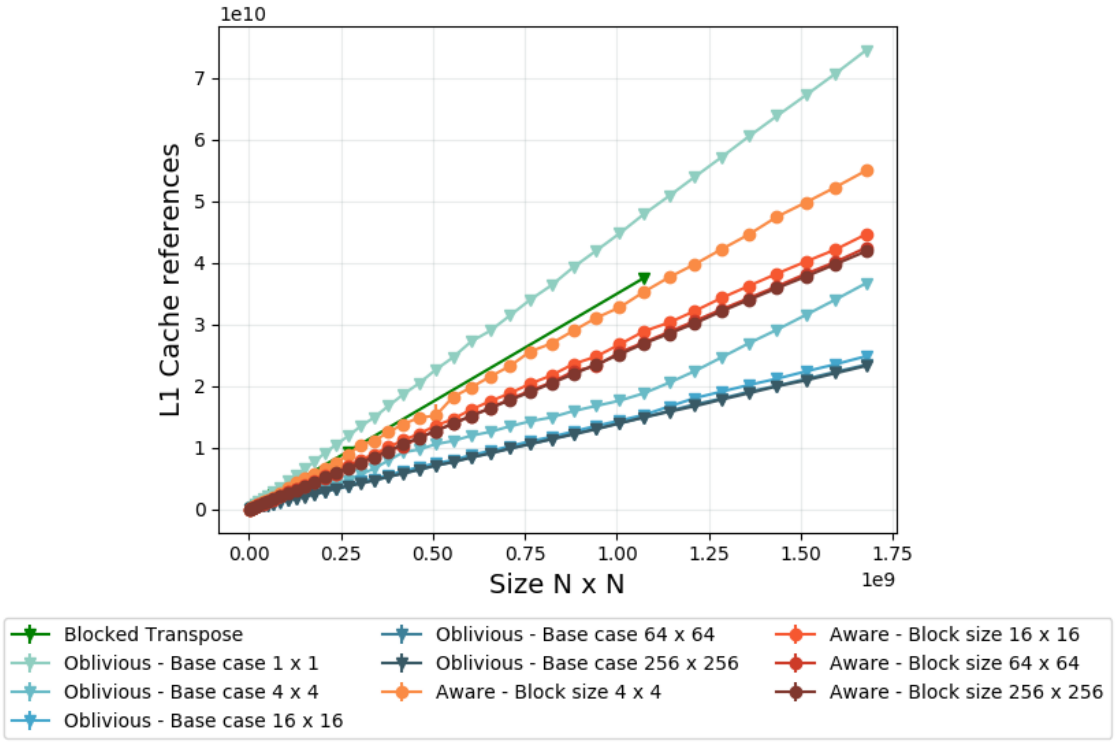


Figure B.5: Performance of Blocked Transpose, Cache-Oblivious and Cache-Aware algorithms in terms of cache references at L1 cache

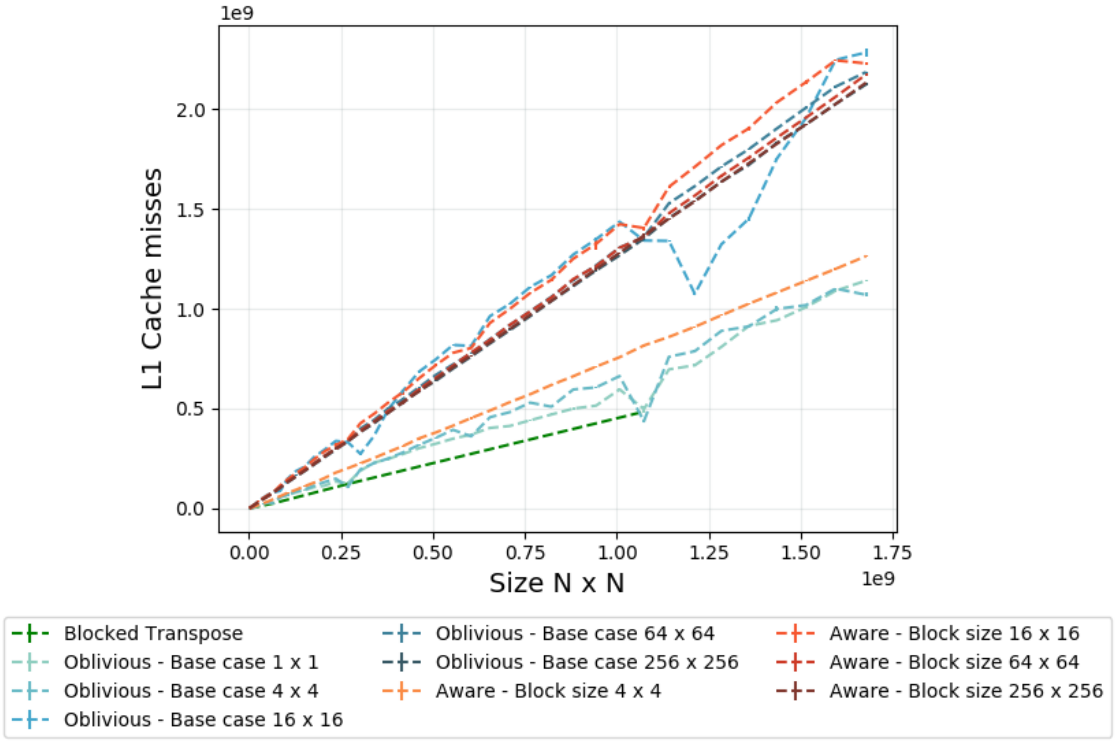


Figure B.6: Performance of Blocked Transpose, Cache-Oblivious and Cache-Aware algorithms in terms of cache misses at L1 cache

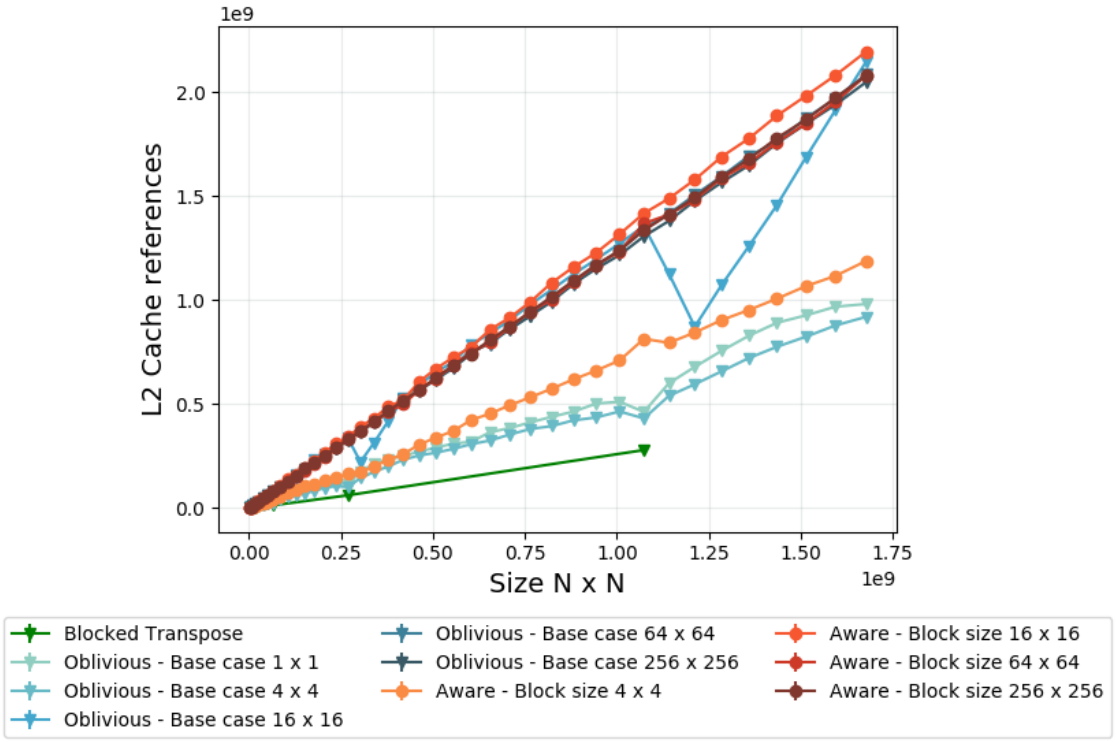


Figure B.7: Performance of Blocked Transpose, Cache-Oblivious and Cache-Aware algorithms in terms of cache references at L2 cache

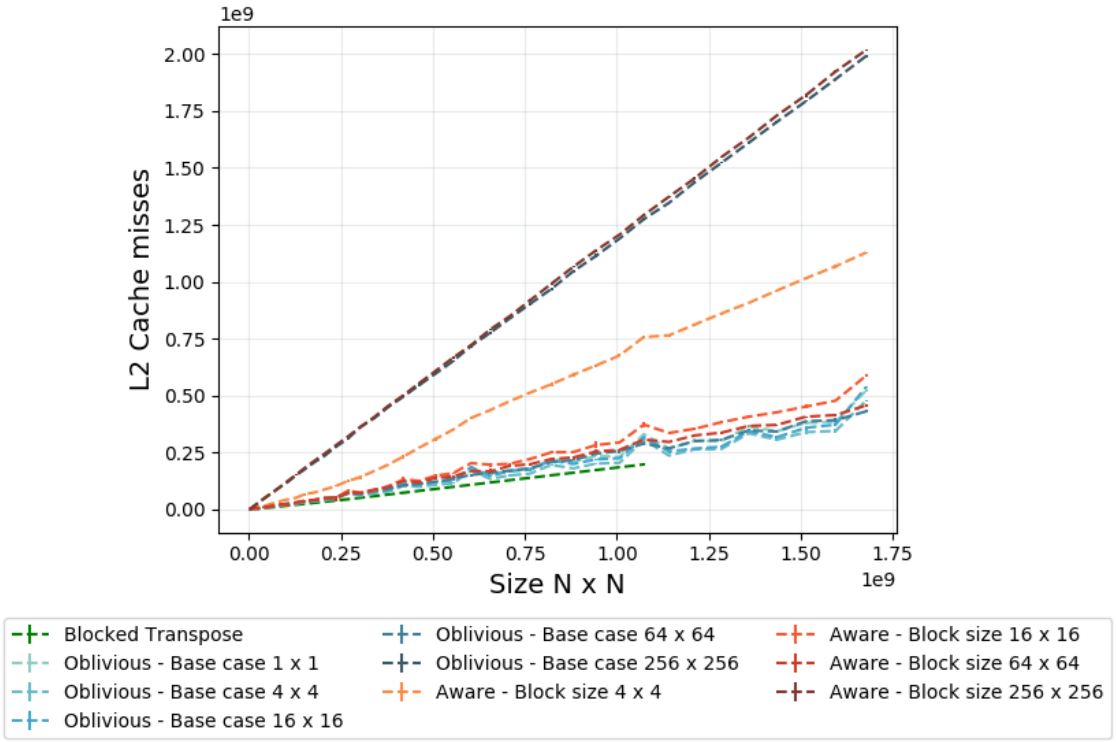


Figure B.8: Performance of Blocked Transpose, Cache-Oblivious and Cache-Aware algorithms in terms of cache misses at L2 cache

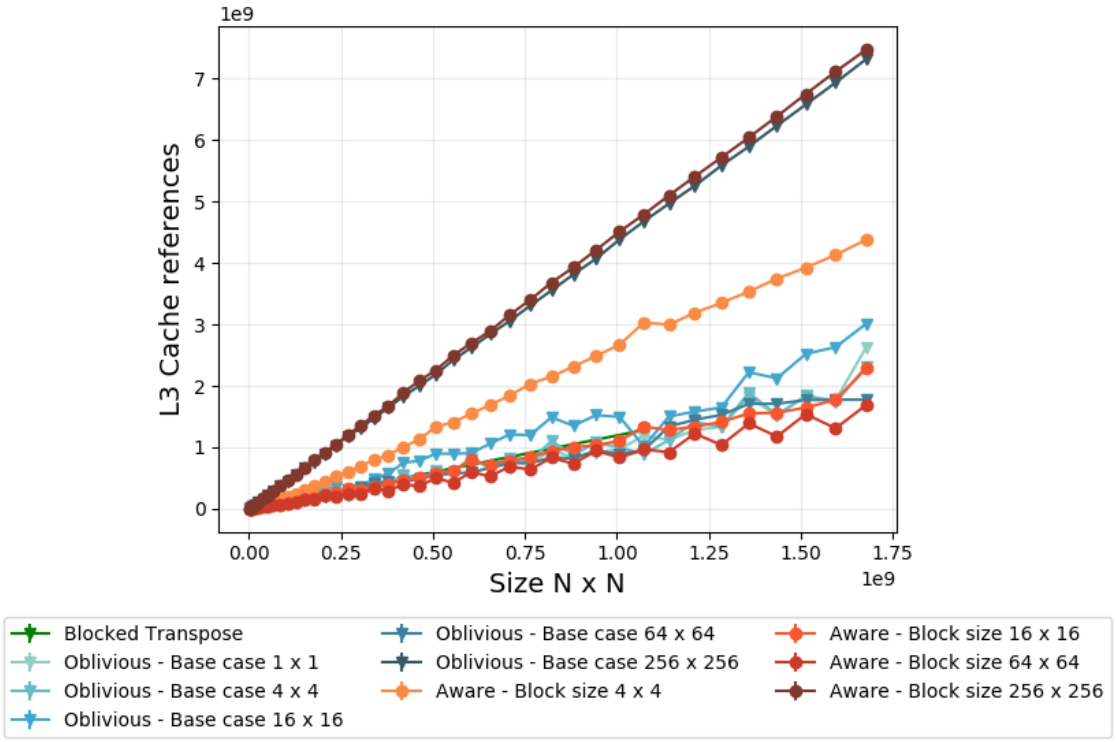


Figure B.9: Performance of Blocked Transpose, Cache-Oblivious and Cache-Aware algorithms in terms of cache references at L3 cache

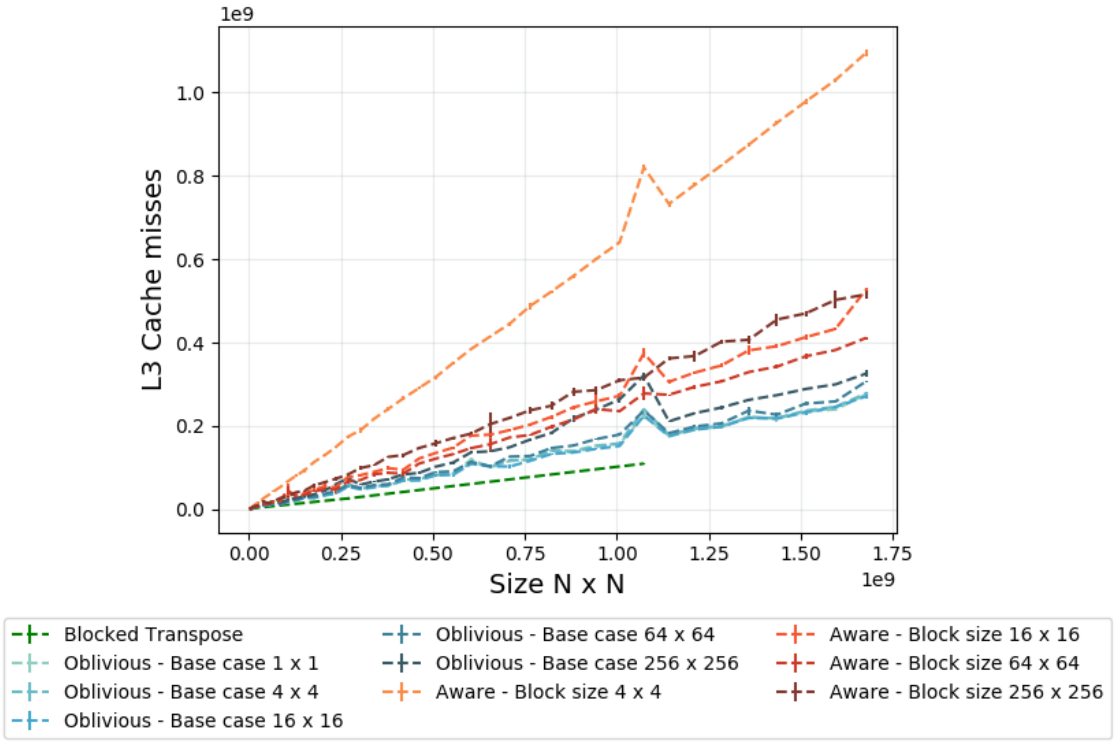


Figure B.10: Performance of Blocked Transpose, Cache-Oblivious and Cache-Aware algorithms in terms of cache misses at L3 cache

N	Naïve	Blocked Transpose	Oblivious - B.c. 1 x 1	Oblivious - B.c. 16 x 16	Oblivious - B.c. 256 x 256
2048	2,093,556 | 501,406	1,639,422 | 64,756	3,223,076 | 84,042	622,576 | 93,202	1,803,036 | 150,340
4096	5,067,085 | 1,183,372	9,028,847 | 157,319	12,280,304 | 265,994	4,219,433 | 484,164	4,915,075 | 730,019
6144	8,074,898 | 2,013,956	None | None	18,167,765 | 382,196	6,556,420 | 709,071	7,780,646 | 1,054,922
8192	11,860,113 | 2,815,153	19,725,118 | 332,005	22,815,116 | 483,526	8,282,328 | 877,556	9,761,572 | 1,369,038
10240	15,075,496 | 3,583,301	None | None	29,859,109 | 665,672	11,273,335 | 1,184,896	12,704,431 | 1,694,979
12288	18,666,627 | 4,347,538	None | None	35,265,458 | 811,991	13,139,676 | 1,411,055	14,893,931 | 2,041,973
14336	21,408,324 | 5,104,549	None | None	41,959,835 | 883,308	15,550,977 | 1,662,014	17,826,395 | 2,390,039
16384	24,899,833 | 5,901,084	37,955,541 | 750,591	47,745,294 | 1,011,962	16,878,192 | 1,806,618	20,567,167 | 2,805,953
18432	28,673,169 | 6,914,500	None | None	53,751,366 | 1,269,370	19,837,112 | 1,708,337	22,819,885 | 3,092,993
20480	31,669,714 | 7,975,311	None | None	60,687,168 | 1,420,104	23,327,452 | 2,407,613	25,312,474 | 3,460,704
22528	35,205,258 | 9,095,212	None | None	66,740,733 | 1,503,068	25,601,556 | 2,689,830	28,109,029 | 3,796,456
24576	38,814,798 | 10,299,988	None | None	73,938,340 | 1,739,620	27,310,682 | 2,885,313	30,589,801 | 4,202,448
26624	41,771,622 | 11,042,506	None | None	78,249,926 | 1,679,633	29,598,709 | 3,109,475	33,009,510 | 4,514,927
28672	45,087,022 | 12,034,773	None | None	84,228,025 | 1,854,985	31,813,605 | 3,381,373	35,501,899 | 4,884,675
30720	48,119,622 | 12,935,444	None | None	90,350,726 | 1,879,567	33,826,778 | 3,567,283	37,888,748 | 5,297,154
32768	52,858,937 | 14,672,335	76,366,213 | 1,544,641	96,781,490 | 2,054,982	34,521,989 | 3,676,841	40,770,847 | 5,753,834
34816	53,987,268 | 14,741,816	None | None	102,783,170 | 2,223,716	37,651,109 | 2,816,158	43,104,792 | 5,895,483
36864	56,850,639 | 15,633,305	None | None	109,660,597 | 2,604,345	41,031,812 | 3,494,218	45,770,900 | 6,236,165
38912	59,858,366 | 16,517,867	None | None	115,088,032 | 2,667,786	43,703,742 | 4,184,339	48,247,551 | 6,582,111
40960	63,028,824 | 17,445,832	None | None	122,057,780 | 3,027,425	46,849,547 | 4,829,081	50,972,990 | 6,934,770

[1] Note that each table cell has the following structure: Number of data blocks transferred | Number of data blocks transferred due to cache misses

Table B.1: Performance of Naïve, Blocked Transpose and Cache-Oblivious algorithms with different base cases in terms of the number of data blocks transferred and the number of data blocks transferred due to cache misses

N	Naïve	Blocked Transpose	Aware - B.s. 4 x 4	Aware - B.s. 64 x 64	Aware - B.s. 256 x 256
2048	2,093,556 | 501,406	1,639,422 | 64,756	2,780,254 | 163,492	4,064,863 | 238,849	3,792,973 | 284,688
4096	5,067,085 | 1,183,372	9,028,847 | 157,319	10,324,539 | 423,744	6,546,371 | 530,109	7,708,598 | 717,956
6144	8,074,898 | 2,013,956	None | None	13,416,623 | 678,455	10,339,086 | 689,196	11,941,541 | 1,134,451
8192	11,860,113 | 2,815,153	19,725,118 | 332,005	16,856,761 | 917,866	13,274,207 | 884,920	15,513,520 | 1,403,046
10240	15,075,496 | 3,583,301	None | None	23,830,896 | 1,176,348	18,710,220 | 1,262,540	19,827,646 | 1,784,208
12288	18,666,627 | 4,347,538	None | None	28,728,440 | 1,406,320	20,962,891 | 1,364,691	23,911,180 | 2,165,375
14336	21,408,324 | 5,104,549	None | None	32,473,534 | 1,645,320	24,841,302 | 1,610,776	28,237,837 | 2,539,371
16384	24,899,833 | 5,901,084	37,955,541 | 750,591	37,172,545 | 1,948,499	28,479,598 | 1,860,231	31,720,218 | 2,886,857
18432	28,673,169 | 6,914,500	None | None	41,765,692 | 2,210,178	31,849,700 | 2,101,589	36,131,268 | 3,268,061
20480	31,669,714 | 7,975,311	None | None	47,039,113 | 2,531,170	34,699,088 | 2,316,025	40,660,963 | 3,639,862
22528	35,205,258 | 9,095,212	None | None	47,820,219 | 2,856,926	39,252,648 | 2,593,787	44,198,808 | 4,008,809
24576	38,814,798 | 10,299,988	None | None	56,593,642 | 3,224,838	42,912,841 | 2,851,755	48,284,491 | 4,345,270
26624	41,771,622 | 11,042,506	None | None	61,424,466 | 3,489,219	46,624,713 | 3,079,250	52,239,495 | 4,724,801
28672	45,087,022 | 12,034,773	None | None	66,118,845 | 3,770,733	49,982,604 | 3,291,115	56,068,564 | 5,084,801
30720	48,119,622 | 12,935,444	None | None	71,246,199 | 4,045,046	52,827,255 | 3,555,835	60,105,749 | 5,461,062
32768	52,858,937 | 14,672,335	76,366,213 | 1,544,641	76,417,142 | 4,673,664	57,439,858 | 3,793,598	64,468,374 | 5,809,875
34816	53,987,268 | 14,741,816	None | None	80,515,974 | 4,593,715	61,145,162 | 4,015,037	68,138,809 | 6,173,936
36864	56,850,639 | 15,633,305	None | None	85,214,855 | 4,871,832	64,777,880 | 4,256,184	72,381,335 | 6,540,213
38912	59,858,366 | 16,517,867	None | None	90,121,252 | 5,151,637	68,409,708 | 4,490,079	76,278,749 | 6,926,167
40960	63,028,824 | 17,445,832	None | None	94,580,942 | 5,446,050	72,226,021 | 4,749,224	80,332,580 | 7,286,088

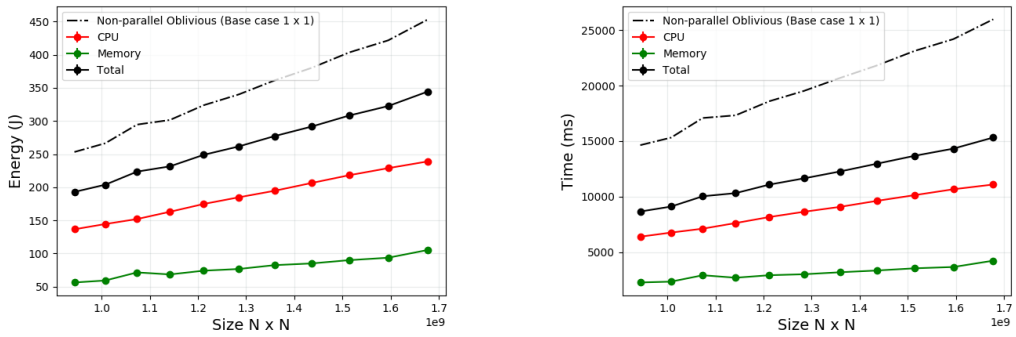
[1] Note that each table cell has the following structure: Number of data blocks transferred | Number of data blocks transferred due to cache misses

Table B.2: Performance of Naïve, Blocked Transpose and Cache-Aware algorithms with different block sizes in terms of the number of data blocks transferred and the number of data blocks transferred due to cache misses

Appendix C

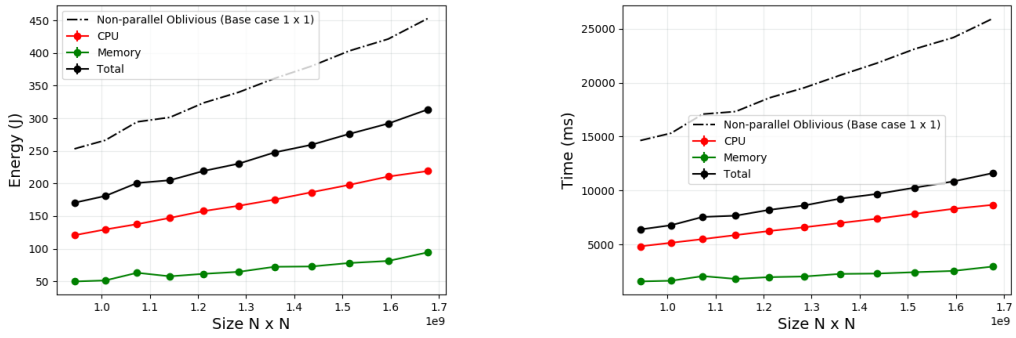
Cache-Oblivious Parallel Algorithm

Cache-Oblivious Parallel Algorithm with a base case of 1×1



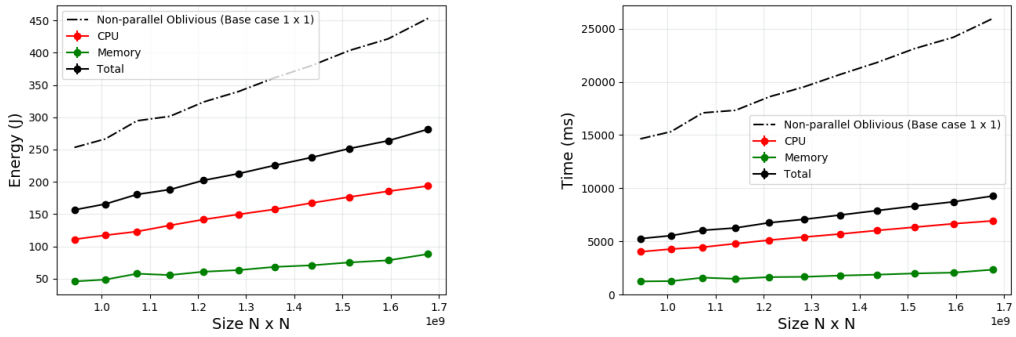
(a) Energy (b) Time

Figure C.1: Performance of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on two virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



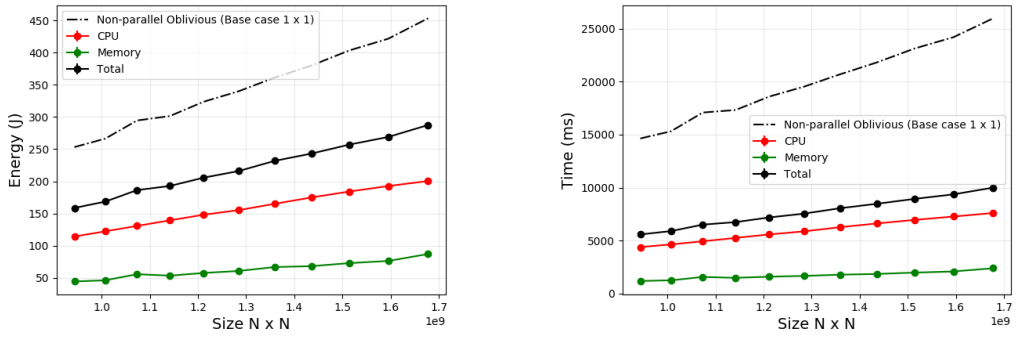
(a) Energy (b) Time

Figure C.2: Performance of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on three virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



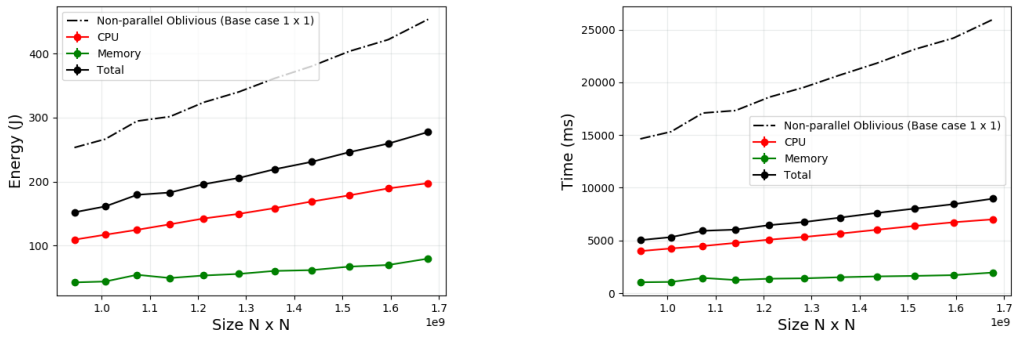
(a) Energy (b) Time

Figure C.3: Performance of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on four virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



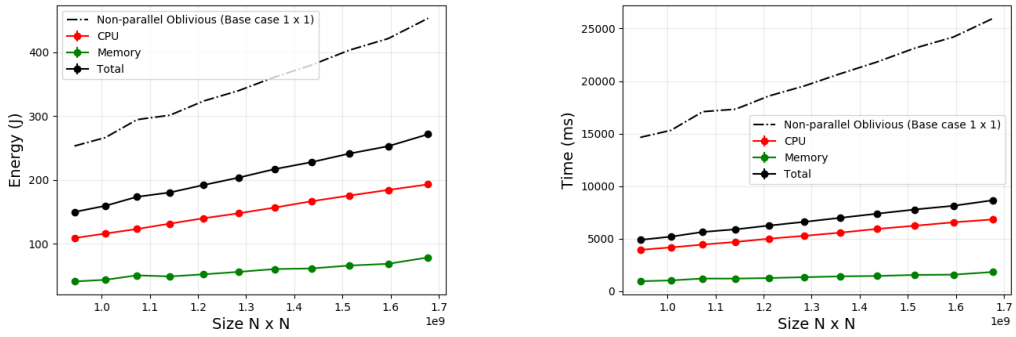
(a) Energy (b) Time

Figure C.4: Performance of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on five virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



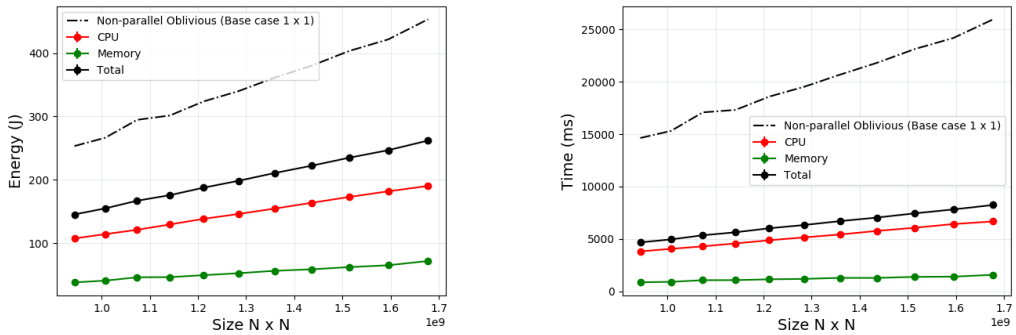
(a) Energy (b) Time

Figure C.5: Performance of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on six virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

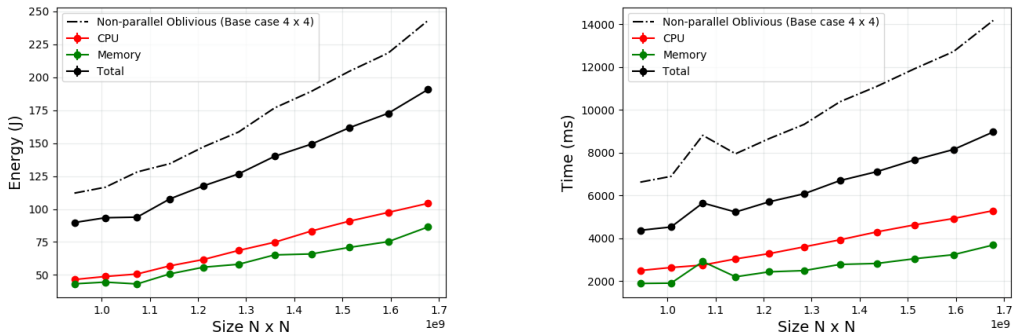
Figure C.6: Performance of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on seven virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

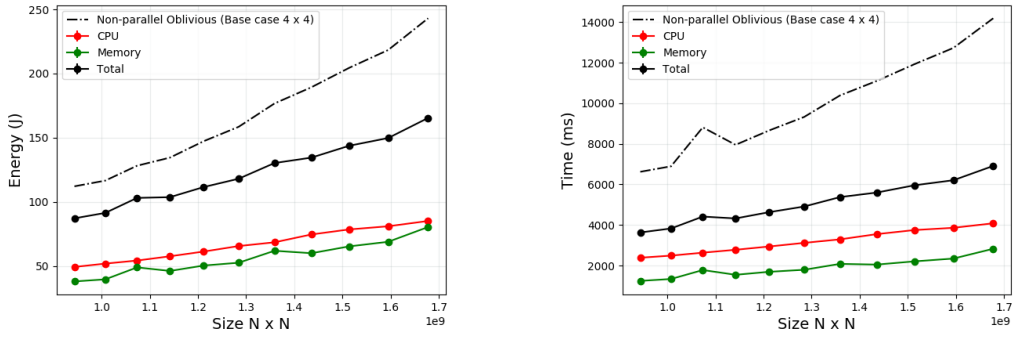
Figure C.7: Performance of Cache-Oblivious Parallel Algorithm with a base case of 1×1 on eight virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses

Cache-Oblivious Parallel Algorithm with a base case of 4×4



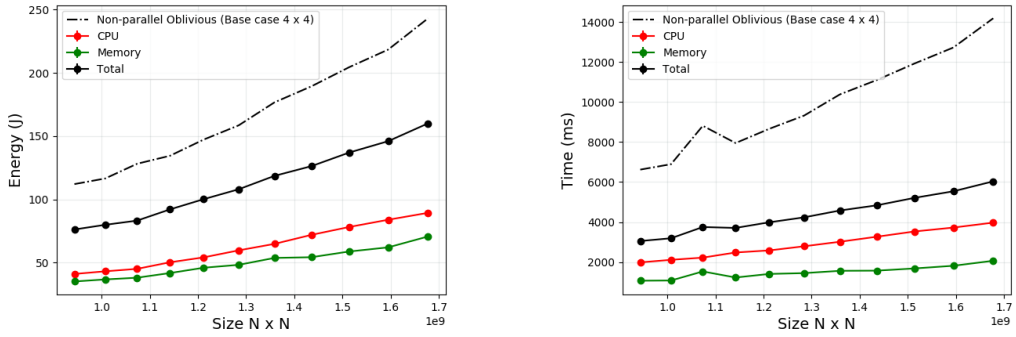
(a) Energy (b) Time

Figure C.8: Performance of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on two virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



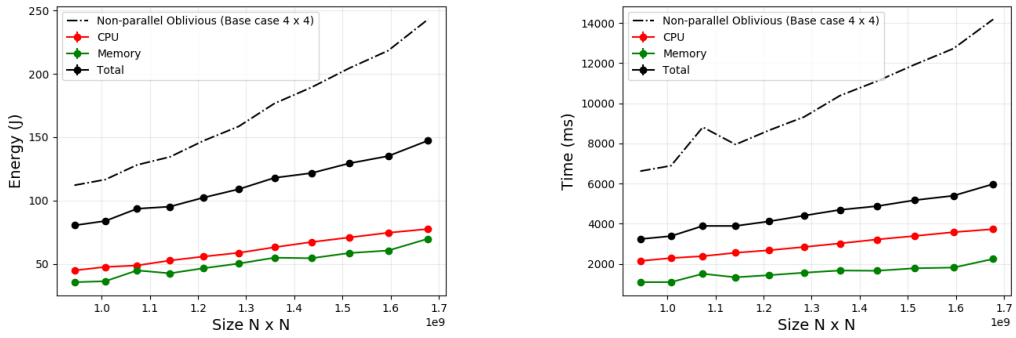
(a) Energy (b) Time

Figure C.9: Performance of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on three virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



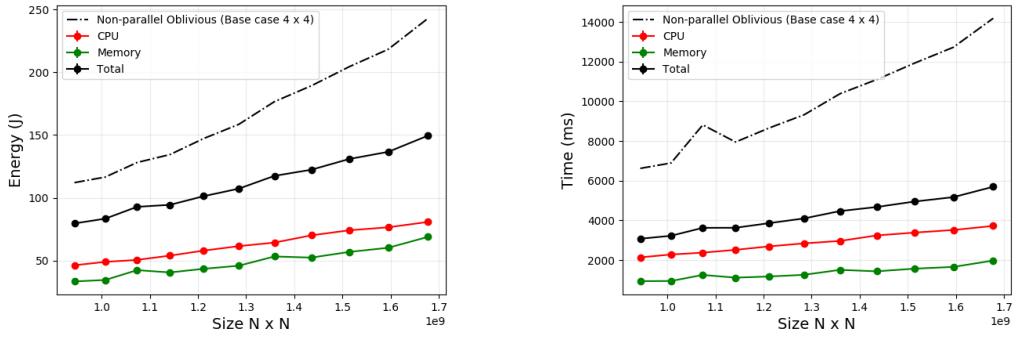
(a) Energy (b) Time

Figure C.10: Performance of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on four virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



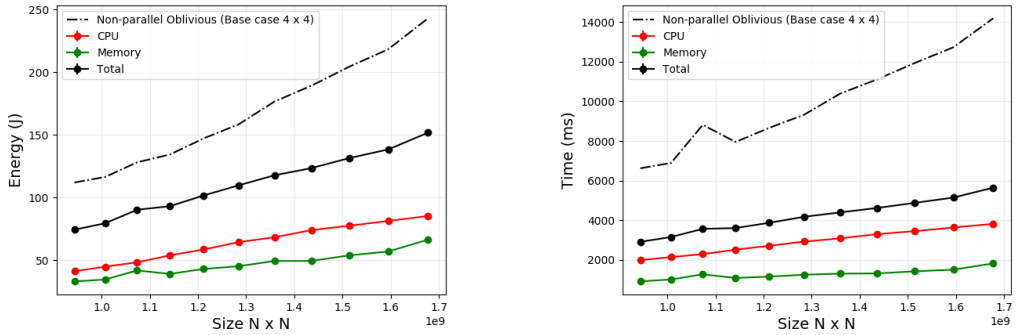
(a) Energy (b) Time

Figure C.11: Performance of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on five virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



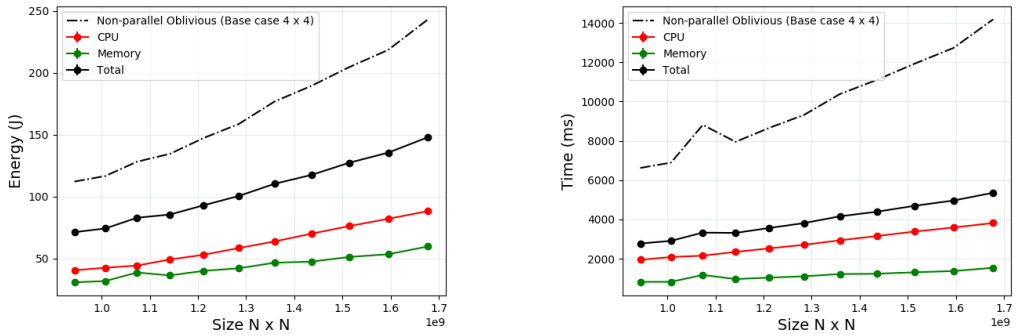
(a) Energy (b) Time

Figure C.12: Performance of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on six virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

Figure C.13: Performance of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on seven virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



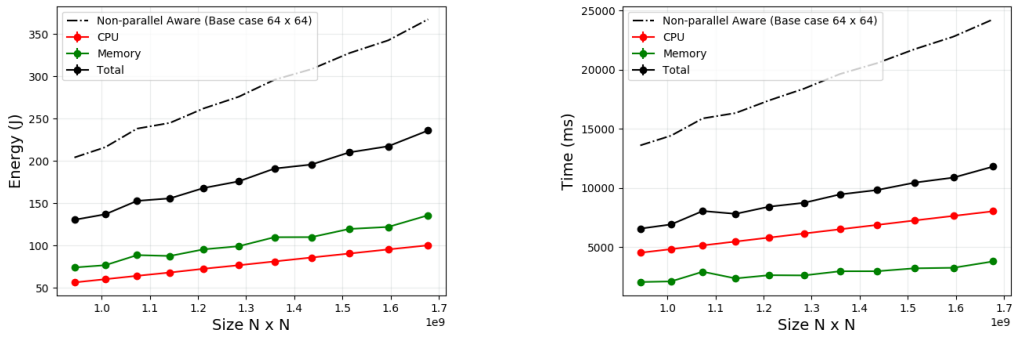
(a) Energy (b) Time

Figure C.14: Performance of Cache-Oblivious Parallel Algorithm with a base case of 4×4 on eight virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses

This page is intentionally left blank.

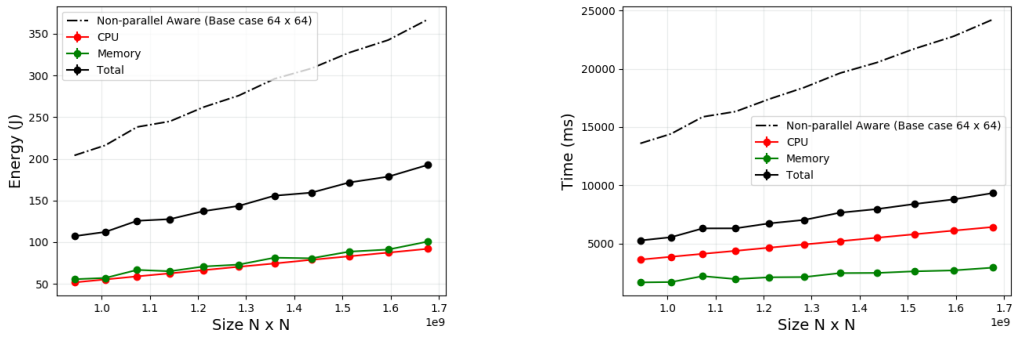
Appendix D

Cache-Aware Parallel Algorithm



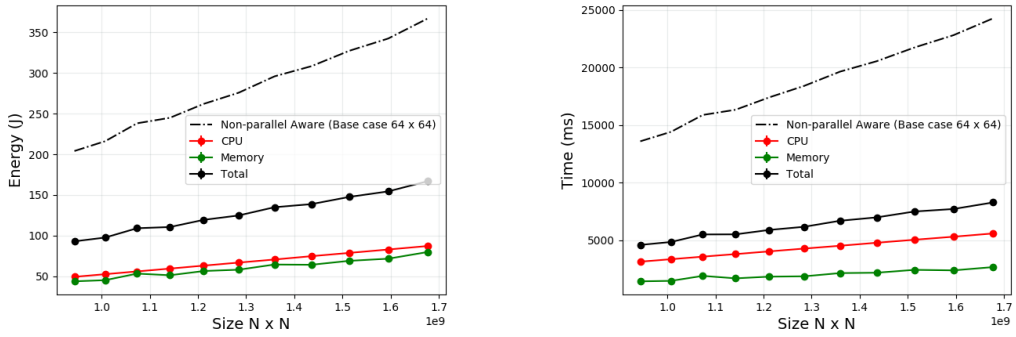
(a) Energy (b) Time

Figure D.1: Performance of Cache-Aware Parallel Algorithm with a block size of 64×64 on two virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



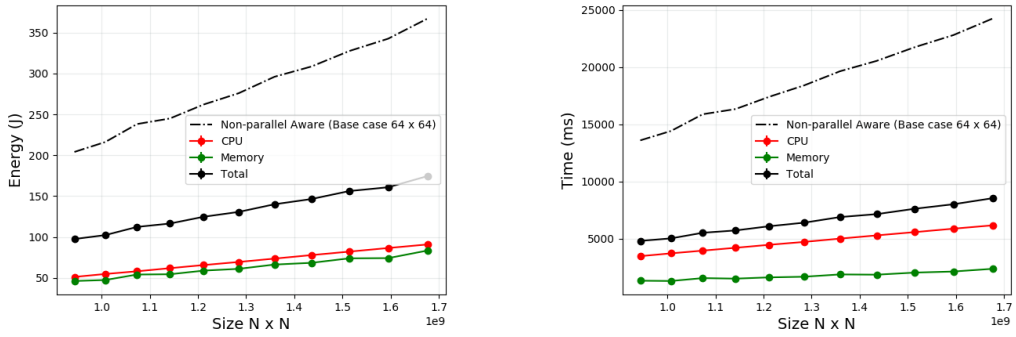
(a) Energy (b) Time

Figure D.2: Performance of Cache-Aware Parallel Algorithm with a block size of 64×64 on three virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



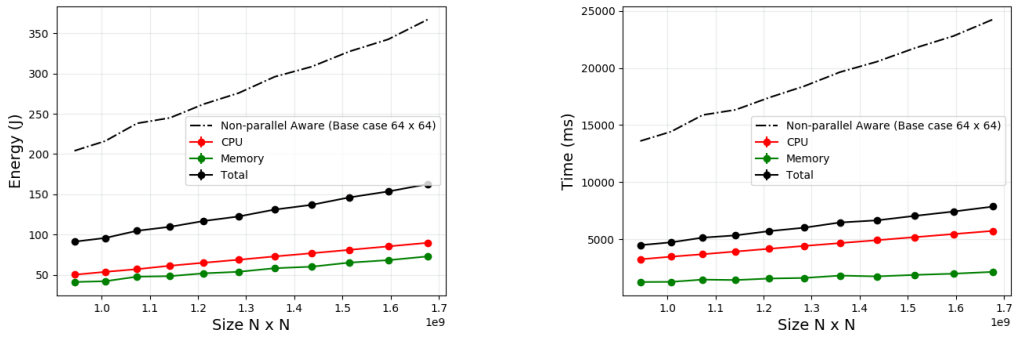
(a) Energy (b) Time

Figure D.3: Performance of Cache-Aware Parallel Algorithm with a block size of 64×64 on four virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



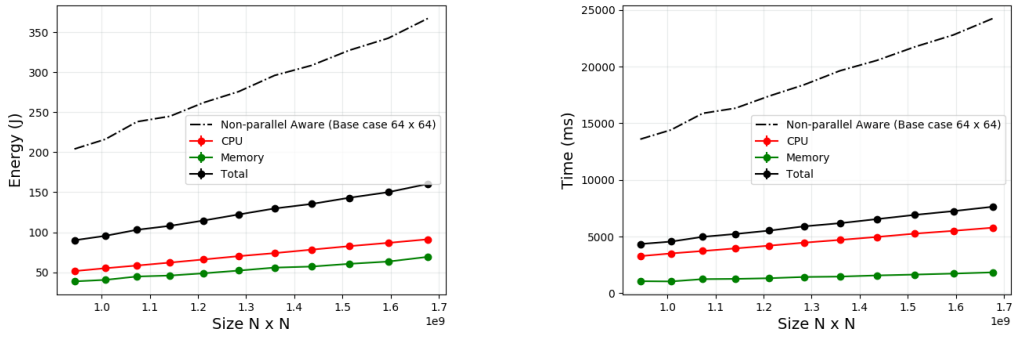
(a) Energy (b) Time

Figure D.4: Performance of Cache-Aware Parallel Algorithm with a block size of 64×64 on five virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



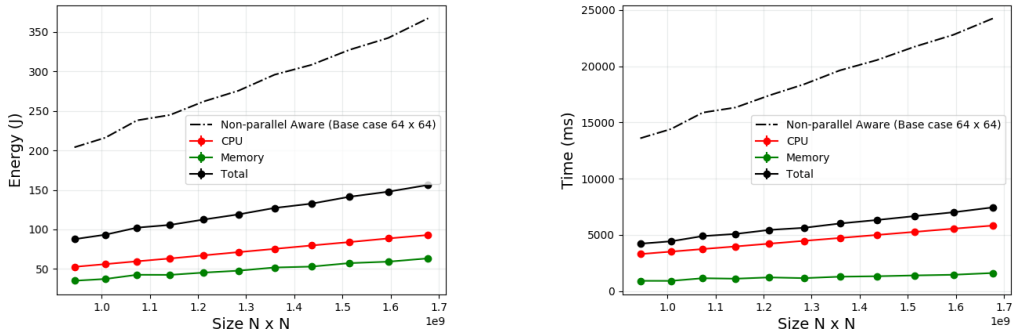
(a) Energy (b) Time

Figure D.5: Performance of Cache-Aware Parallel Algorithm with a block size of 64×64 on six virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

Figure D.6: Performance of Cache-Aware Parallel Algorithm with a block size of 64×64 on seven virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses



(a) Energy (b) Time

Figure D.7: Performance of Cache-Aware Parallel Algorithm with a block size of 64×64 on two virtual cores in terms of Energy (a) and Time (b) with respect to CPU instructions and memory accesses

This page is intentionally left blank.

Appendix E

Cache-Oblivious Parallel Algorithm vs Cache-Aware Parallel Algorithm

E.1 Energy and Time

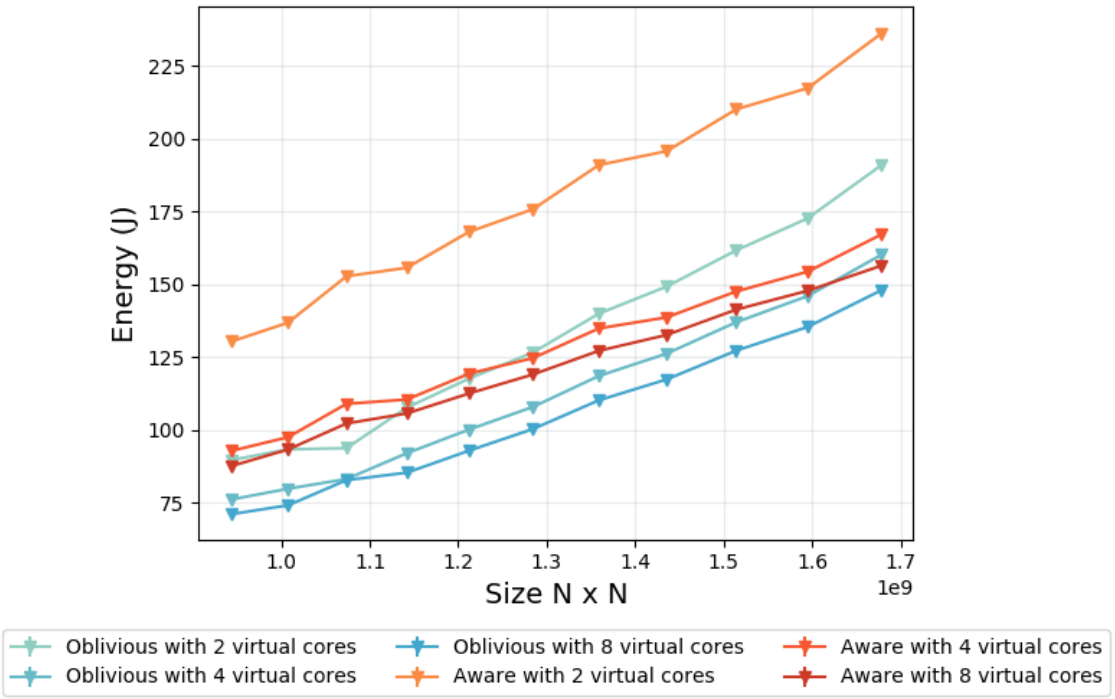


Figure E.1: Performance of Cache-Oblivious Algorithm with a base case of 4×4 and Cache-Aware Parallel Algorithm with a block size of 64×64 on diverse numbers of virtual cores in terms of Energy

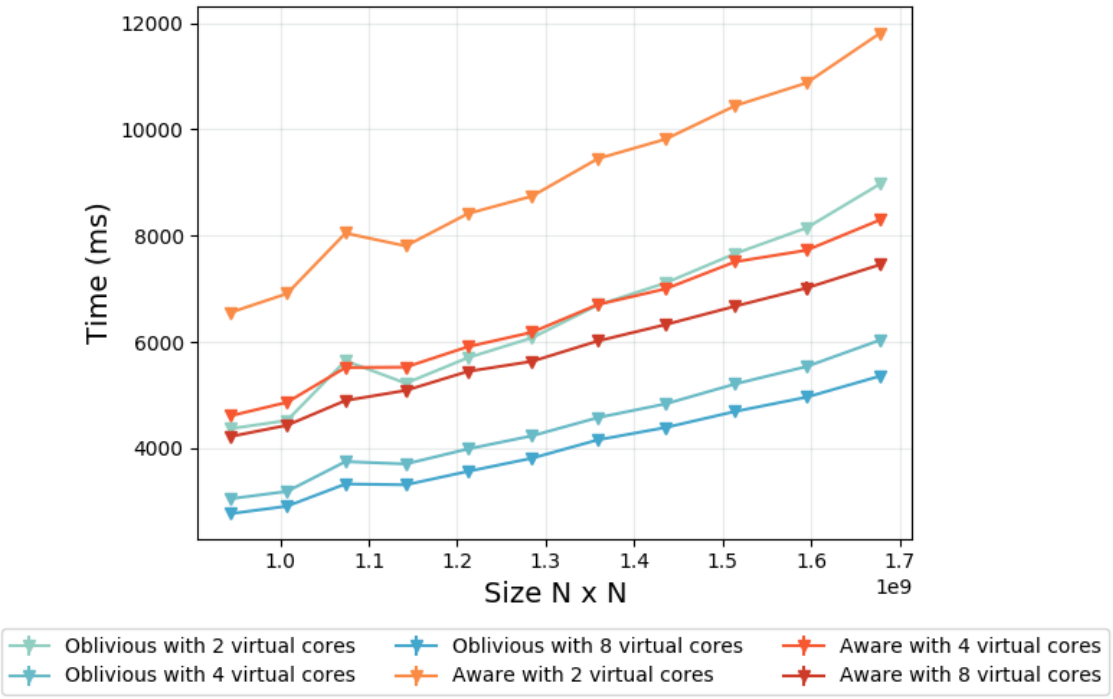


Figure E.2: Performance of Cache-Oblivious Algorithm with a base case of 4×4 and Cache-Aware Parallel Algorithm with a block size of 64×64 on diverse numbers of virtual cores in terms of Time

E.2 Cache references and misses

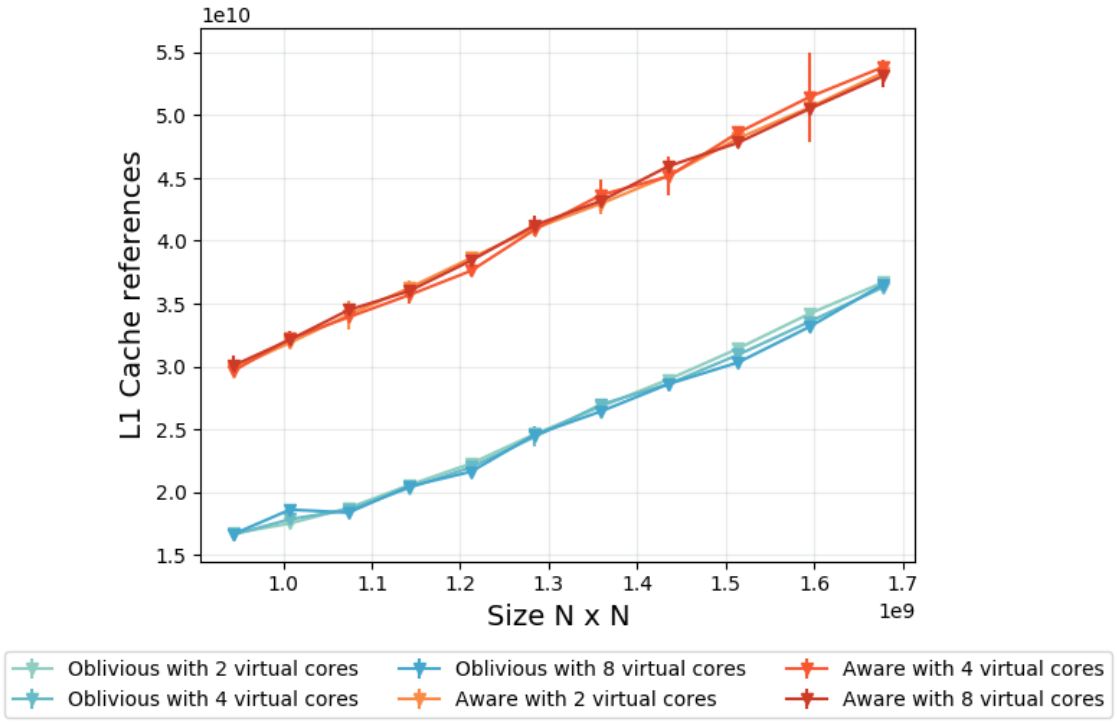


Figure E.3: Performance of Cache-Oblivious Algorithm with a base case of 4×4 and Cache-Aware Parallel Algorithm with a block size of 64×64 on diverse numbers of virtual cores in terms of cache references at L1 cache

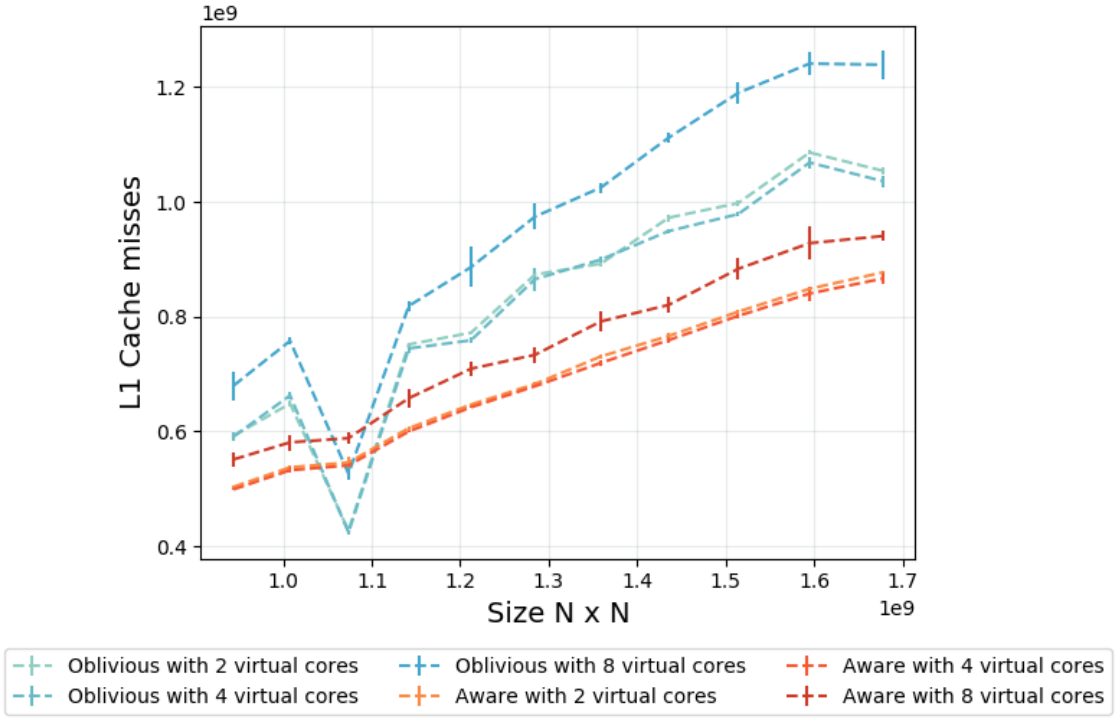


Figure E.4: Performance of Cache-Oblivious Algorithm with a base case of 4×4 and Cache-Aware Parallel Algorithm with a block size of 64×64 on diverse numbers of virtual cores in terms of cache misses at L1 cache

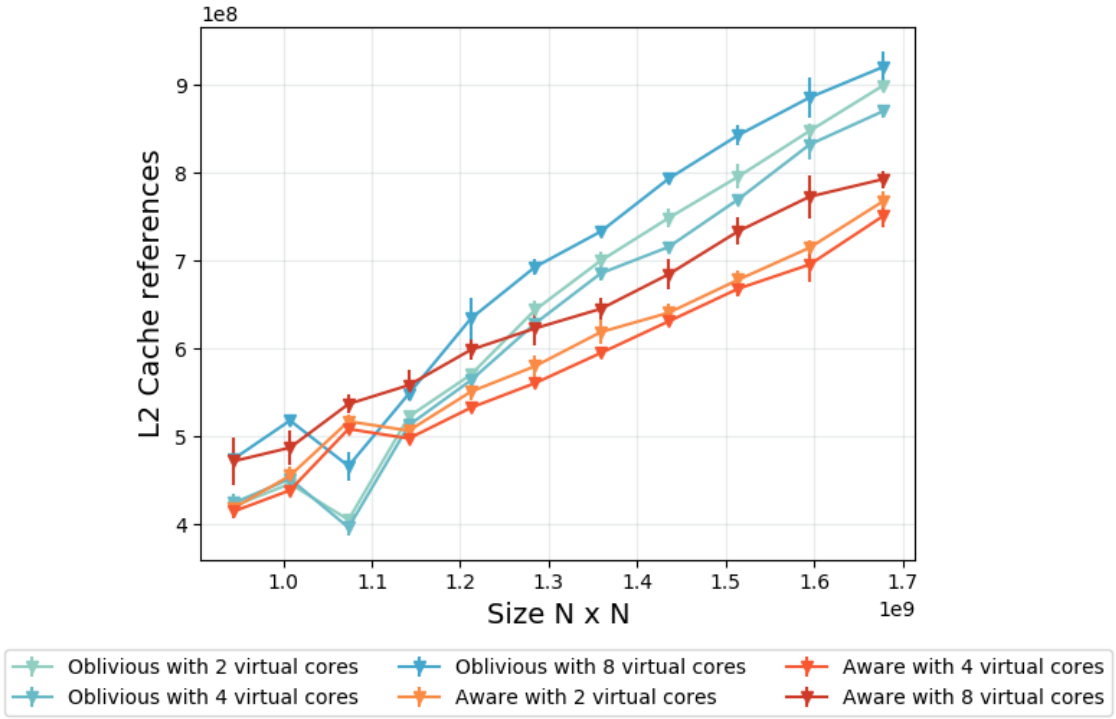


Figure E.5: Performance of Cache-Oblivious Algorithm with a base case of 4×4 and Cache-Aware Parallel Algorithm with a block size of 64×64 on diverse numbers of virtual cores in terms of cache references at L2 cache

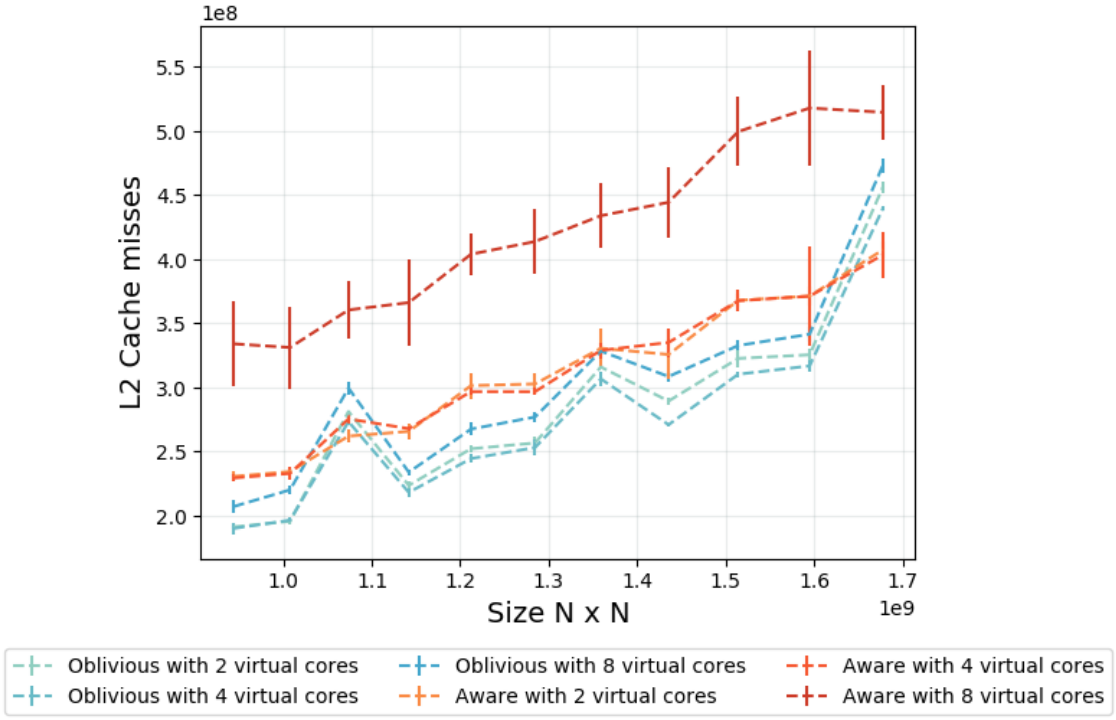


Figure E.6: Performance of Cache-Oblivious Algorithm with a base case of 4×4 and Cache-Aware Parallel Algorithm with a block size of 64×64 on diverse numbers of virtual cores in terms of cache misses at L2 cache

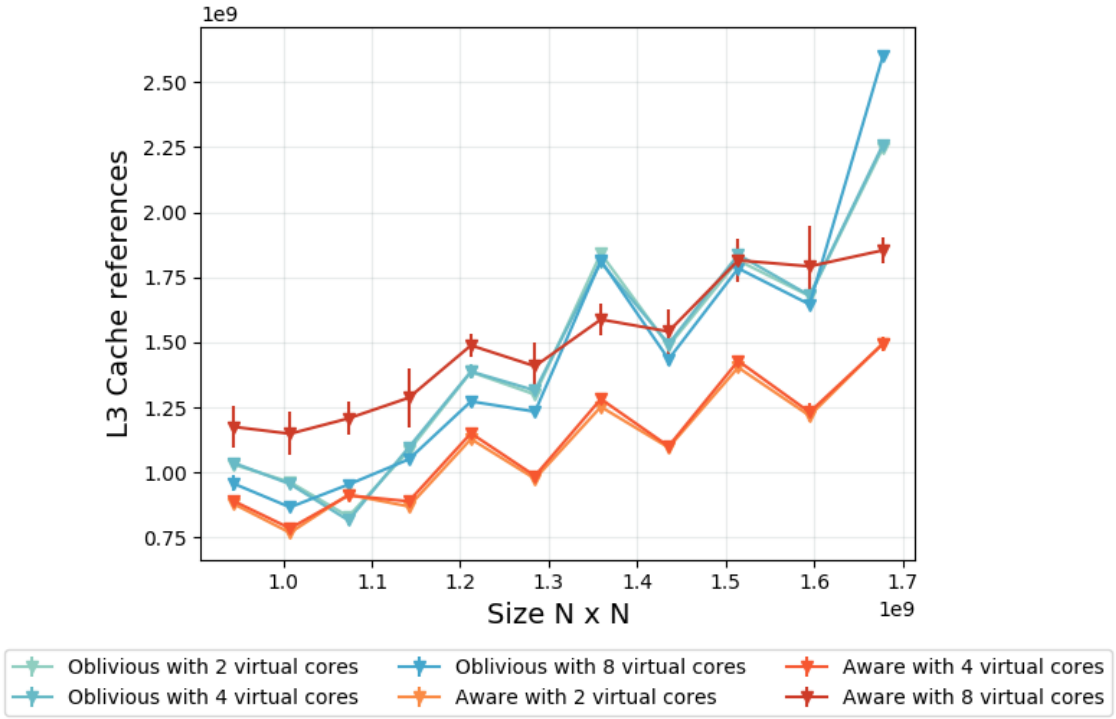


Figure E.7: Performance of Cache-Oblivious Algorithm with a base case of 4×4 and Cache-Aware Parallel Algorithm with a block size of 64×64 on diverse numbers of virtual cores in terms of cache references at L3 cache

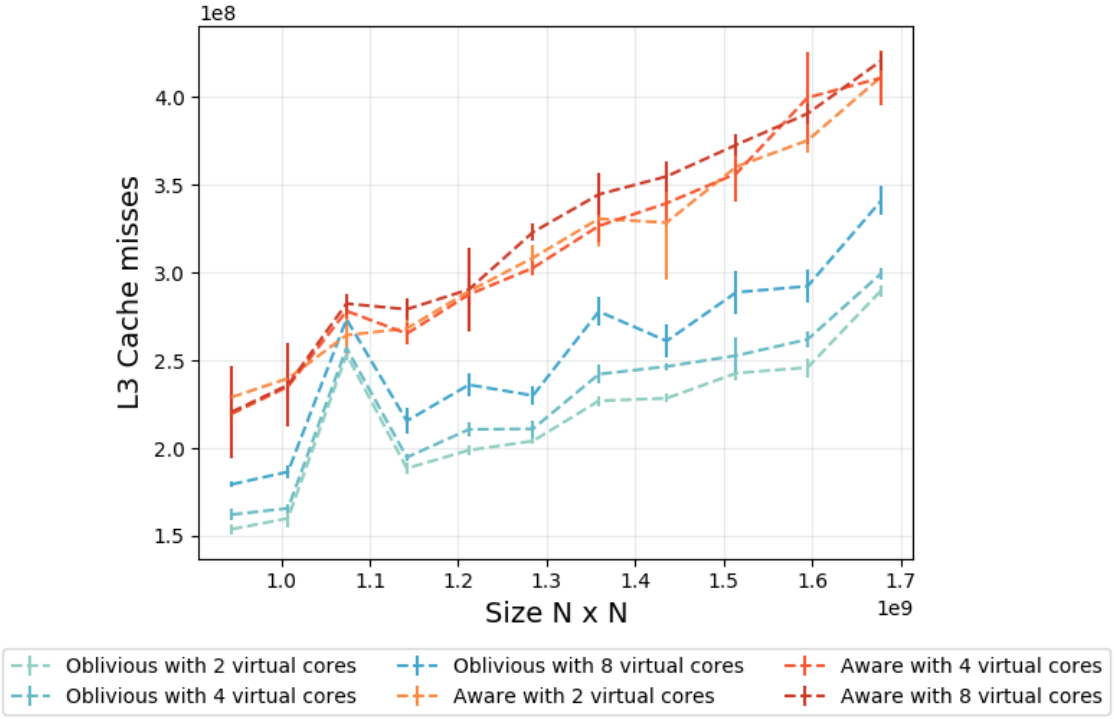


Figure E.8: Performance of Cache-Oblivious Algorithm with a base case of 4×4 and Cache-Aware Parallel Algorithm with a block size of 64×64 on diverse numbers of virtual cores in terms of cache misses at L3 cache