Karmarkar's interior point algorithm: A parallel implementation with OpenMP and CUDA

Samuele Fonio, Marco Edoardo Santimaria

October 13, 2023

Abstract

In this paper, we introduce a parallelized version of Karmarkar's linear solver to enhance its performance by reducing execution time. We explore various parallelism paradigms to achieve this goal. Specifically, we assess parallelization strategies for individual operations within the algorithm using OpenMP and GPU acceleration with CUDA. The result of our efforts is a parallelized implementation of the algorithm that leverages OpenMP and CUDA. Through this parallelization, we were able to achieve a significant speedup, with a maximum performance improvement of up to 300 times compared to the original sequential implementation.

Contents

1 Introduction

Large linear optimization problems have always been difficult to be computed in feasible time. The complexity of the algorithms has always been bounded with terms of the same size as the space of the solution, which grows exponentially. The simplex method, traditionally used for linear optimization problems, often resulted in sub-optimal solutions for many complex problems. Luckily, in the 1980s, solving large linear programs became more efficient, thanks to the need for algorithms with better theoretical properties. Researchers began redefining these linear problems as nonlinear, and applied various adaptations of nonlinear algorithms, such as Newton's method, to address them. The hallmark of these methods was their emphasis on ensuring boundary conditions represented by the problem's inequality constraints. These approaches gained recognition under the name of **Interior Point Methods**.

Interior Point Methods, also known as interior-point or barrier methods, are a class of optimization algorithms originally developed in the 1980s. They have since become a fundamental tool in mathematical optimization and operations research. These methods are characterized by:

- **Convex Optimization**: Interior point methods are particularly suitable for problems where both the objective function and the inequality constraints are convex, a common scenario in many real-world applications.

- **Barrier Function**: These methods introduce a barrier function that penalizes points violating the inequality constraints, guiding the optimization process towards feasible solutions while maintaining convexity.

In practice, interior point methods focus on ensuring both feasibility (for the constraints) and optimality (for the solution). They converge to the final point, with the barrier function vanishing, indicating the discovery of a solution that optimizes the objective function while satisfying the constraints.

One key distinction from the simplex method is that each interior-point iteration is computationally expensive but makes significant progress toward the solution. In contrast, the simplex method requires a larger number of inexpensive iterations as it works its way around the boundary of the feasible polytope, testing vertices until it finds the optimal one. Interior-point methods approach the boundary of the feasible set only in the limit, either from the interior or the exterior, but they never actually reside on the boundary of this region. These methods have revolutionized the field of optimization, especially for solving large-scale linear programming problems efficiently.

Before the raise of interior point methods, the time required to solve large linear programs was exponential in the size of the problem in the worst case scenario. Karmarkar's projective algorithm [7], introduced in the 1980s, brought polynomial complexity to linear programming, offering good practical performance with respect to the state-of-the-art at that time, the ellipsoid method [8]. This breakthrough resulted in a variety of related methods build upon Karmarkar's original algorithm.

An example of how the interior point methods work, is shown graphically below 1, for a problem in 2 dimensions. In blue we can see the constraints equations, while in red, we can see the objective function evaluation.

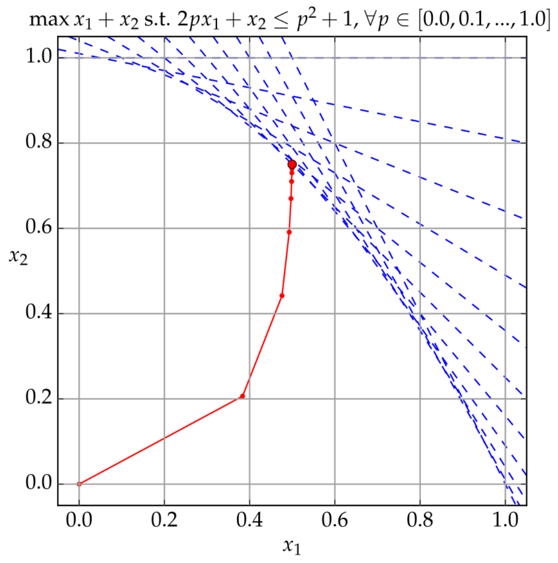

Figure 1: Interior point solution example from wikipedia

In this report, we are going to present Karmarkar's algorithm and show how we developed a parallel version of it, enabling further optimization in the solution of linear programming problems.

2 Algorithm

In this section we are going to present Karmarkar's algorithm [7]. Karmarkar's algorithm is an interior-point method since the current guess for the solution does not follow the boundary of the feasible set (as in the simplex method), but follows a central path through the interior of the feasible region, improving the approximation of the optimal solution and converging to that [11, 5].
To be solved with this algorithm, a problem must be in *canonical form*[1], that is: for a given dimension n we have $c, x \in \mathbb{R}^n$, $A \in \mathbb{R}^{m,n}$, $\mathbf{1} = (1, \ldots, 1)^t$, $\mathbf{e} = (1/n, \ldots, 1/n)^t$:

1. Constraints are the following:

$$\begin{aligned} v :=& min \;\; c^t x \\ & Ax = 0 \\ & \mathbf{1}x = 1 \\ & x \geq 0 \end{aligned} \tag{1}$$

2. $v = 0$;

3. $A\mathbf{1} = 0$;

4. The rows of A are linearly independent.

As stated in the previous chapter, Karmarkar's algorithm has outperformed the state-of-the-art of that time, which was represented by the simplex method, that is exponential in complexity, and the ellipsoid method [8], which is polynomial in complexity. However, Karmarkar's algorithm operates in the n-dimensional unit simplex just like its competitors. A n-dimensional unit simplex S is the set of points $(x_1, x_2, \ldots, x_n)$ satisfying:

$$\begin{aligned} x_1 + x_2 + \ldots + x_n &= 1 \\ x_j &\geq 0 \\ j &= 1, 2, \ldots, n \end{aligned}$$

Karmarkar's algorithm wants to create a sequence of points $x^0, x^1, x^2, \ldots, x^k$ having decreasing values of the objective function in 1. In particular, at the k-th step, the point x^k is brought into the center of the simplex by a projective transformation, which enables to create a direct path towards the minimum, optimizing the objective function and weighting properly the length of the steps.

To do so these there are three main concepts to be discussed:

- Projection of a vector onto the set of X satisfying $AX = 0$;
- Karmarkar's centering transformation;
- Karmarkar's potential function.

[1] A note on the canonical form, our algorithm implementation, just like other implementations, does not implement the conversion from standard to canonical form.

2.1 Projection

We want to move from a feasible point x^0 to another feasible point x^1, that for some fixed vector v, will have a smaller value than v^tX. If we choose to move in direction $d = (d_1, d_2, \ldots, d_n)$ that solves the optimization problem:

$$\begin{aligned} min \;\; v^t d, \\ Ad = 0, \\ d_1 + d_2 + ...d_n = 0 \\ \|d\| = 1 \end{aligned}$$

then we will be moving in a feasible direction that maximizes the decrease in v^tx per unit length moved. The direction d that solves this optimization problem is given by the projection of v onto X satisfying $Ax = 0$ and $x_1+x_2+...+x_n = 0$ and is given by solving $BB^tw = B\bar{c}$;, where $B^t := (\bar{A}^t, \mathbf{1}^t)$ and $\bar{c} = cdiag(\bar{x})$. A crucial aspect in our implementation of the algorithm is that BB^t is symmetric and positive definite, which means that the solution to this linear system can be performed using the Cholesky factorization.

2.2 Karmarkar's centering transformation

If x^* is a point in S, then the centering transformation $f(x_1, x_2, \ldots, x_n \| x_k)$ transforms a point $(x_1, x_2, \ldots, x_n)$ in S into a point $(y_1, y_2, \ldots, y_n)$ in S, where:

$$y = \frac{\frac{x_j}{x_j^k}}{\sum_{r=1}^{n} \frac{x_r}{x_r^k}}.$$

The properties of the centering transformation are the following:

1. $f(x^k \| x^k) = (\frac{1}{n}, \ldots, \frac{1}{n})$: Which mean that $f(\cdot \| x^k)$ maps x^k into the center of the transformed unit simplex;
2. $f(x\|x^k) \in S$ and for $x \neq x', f(x\|x^k) \neq f(x'\|x^k)$: any point in S is transformed into a point in the transformed unit simplex, and it is a one-one mapping (two points can't be transformed into the same point).
3. $\forall (y_1, \ldots, y_n) \in S$, there is a unique point $(x_1, \ldots, x_n) \in S$ satisfying $f((x_1, \ldots, x_n)\|x^k) = (y_1, \ldots, y_n)$: consequently, f is a one-one onto mapping from s to S.
4. A point $x \in S$ will satisfy $Ax = 0$ if $AD_k f(x\|x^k) = 0$ (with $D_k = diag(x^k)$)

To recap, we move from the center of the transformed unit simplex in a direction opposite to the projection of cD_k onto the transformation of the feasible region (the set of y satisfying $AD_k y = 0$). This ensures that we maintain feasibility in the transformed space and move in a direction that maximizes the rate of decrease of cD_k. In addition, notice that thanks to the

Karmarkar's centering function property of invertibility (property 4), when we transform y^{k+1} back into x^{k+1}, the definition of projection implies that x^{k+1} will be feasible for the original LP.

2.3 Karmarkar's potential function

Given the current solution c we don't project onto the transformed space c, but cD^t. However, this may result in non minimizing the objective function:

$$cx^{k+1} > cx^k.$$

This problem is overcome by the introduction of the **Karmarkar's potential function**:

$$\psi(x) := n\log(cx) - \sum_i \log(x_i).$$

Since the optimal value is 0, this potential function goes to $-\infty$ when $x \to x*$. Karmarkar showed that if we project cD^t onto the feasible region in the transformed space, then for some $\delta > 0$, it will be true that for $k = 0, 1, 2, \ldots$

$$f(x^k) - f(x^{k+1}) \geq \delta.$$

This means that each iteration of Karmarkar's algorithm decreases the potential function by an amount bounded away from 0. This is crucial when choosing the length of the step α, which becomes:

$$\hat{\alpha} = \min_{\alpha} \bar{\psi}(e + \alpha d),$$

where $\bar{\psi} = n\log(\bar{c}x) - \sum_i \log(x_i)$, and $\bar{c}$ is the vector in the transformed space to be minimized:

$$\begin{aligned} &min \;\; \bar{c}^t y \\ &\bar{A}y = 0 \\ &y \in S, \end{aligned} \tag{2}$$

where $\bar{c} = c{*}diag(\bar{x})$, $\bar{A} = A{*}diag(\bar{x})$ and $\bar{x}$ is the current solution. The minimization of the potential function is usually performed with a line search. However, due to simplicity, we decided not to optimize this α and to keep a constant step of:

$$\hat{\alpha} = \alpha * r = \frac{n-1}{3n} \frac{1}{\sqrt{n(n-1)}},$$

where r is the ray of the maximum sphere included in the affine space aff(S) centered in e and inside S. This ensures that y^{k+1} will remain in the interior of the transformed simplex. The resulting algorithm is the following:

Algorithm 1 Karmarkar's algorithm

```
Require: (A,c,$\epsilon$);
  Let $x := \mathbf{e}$
  $\hat{\alpha} := \alpha r = \frac{n-1}{3n}\frac{1}{\sqrt{n(n-1)}}$;
  while $c^t x > \epsilon$ do:
      $\bar{c} := c^t diag(\bar{x})$;
      $\bar{A} := A diag(\bar{x})$;
      $B^t := (\bar{A}^t, \mathbf{1}^t)$;
      solve $BB^t w = B\bar{c}$;
      $\bar{c}' = \bar{c} - B^t w$
      $d = -\bar{c}'/\|\bar{c}'\|$;
      $y := \mathbf{e} + \hat{\alpha} d$;
      $x' := diag(x)y/(\mathbf{1} diag(x)y)$;
      $x := x'$;
  end while
  output($x$);
```

As we can see there are many parts of the algorithm that can be optimized. For instance, the solution of the linear system can be done using the Cholesky factorization, which was implemented from scratch.
For what concerns the complexity of Karmarkar's algorithm, denoting n as the number of variables and L as the number of bits of input to the algorithm, Karmarkar's algorithm requires $O(n^{3.5}L)$ operations on $O(L)$-digit numbers, as compared to $O(n^6 L)$ such operations for the ellipsoid algorithm [8].

3 Parallelization

As the world has become data-driven, the optimization of a code should be the main focus of a good programmer as the demand for greater computational power keeps growing [12]. The parallelization of an algorithm is one possible way to optimize an algorithm, and it can be done with many paradigms. There are multiple software and libraries in every programming language that allow you to reach the maximum capacity of your hardware. However, the theoretical part of parallel computing is not negligible and fundamental in order to optimize properly an algorithm.
For what regards our experience with this project, at first glance, the algorithm would appear not to be parallelizable. This is because every iteration is not independent from each other, and thus it is not possible to start multiple iterations at the same time. If we look more closely at the algorithm, we can see that most of the steps of the algorithm are actually extremely prone to parallelization. In our implementation, we then focused not on running multiple iterations at the same time, but on parallelizing each step of the algorithm.

3.1 Ahmdal's law

The ideal situation when parallelizing an algorithm would be for all the available processors to operate simultaneously. However, there are some operations that must be executed sequentially. As a consequence, Ahmdal tried to figure out how to find the maximum speedup given a fraction of code that can't be parallelized [1]. If the fraction of the computation that cannot be divided into concurrent tasks is f, and there is no overhead when the computation is divided in multiple processors, the time to perform the computation with N processors is given by

$$ft_s + (1-f)\frac{t_s}{N},$$

where t_s is the sequential time. Hence, the speedup factor is given by

$$speedup = \frac{N}{1+(N-1)f}.$$

In our case, by looking at the steps of the algorithm, we can see that there is 1 operation that needs to be done sequentially (the evaluation of α) before the start of the iterations, and one that can be done sequentially. Looking at the main iteration body, there are around 8 operations that can be executed in parallel. However the true fraction of calculation parallelizable is quite tricky [2], we came up with $f = 0.02$. This number has been empirically estimated by analizing the way in wich we implemented the parallelized

[2]This is due to the fact, that some algebraic operations, were parallelized with a mix of sequential and parallel code, which has a result of making it harder to estimate accurately the proportion of sequential and parallel code.

version of the algorithm, and by looking on how each function had been parallelized.
Now if we suppose that the number of cores available is 80, we obtain that we can have a theoretical speedup of

$$speedup = \frac{N}{1 + (N-1)f} = \frac{80}{1 + (80-1) * 0.02} = \frac{80}{2.58} = 31.0078 \approx 31$$

3.2 Gustafson's law

Gustafson in 1988 [6] argued that Amdahl's law was not enough to determine the potential speedup limits of an algorithm. In particular, Amhdal's law focuses on the scaling ability of the algorithm at a fixed size of the problem, and a growing number of processors. Gustafson observes that, in practical scenarios, larger multiprocessors are able to accommodate problems of bigger size within reasonable execution times. Consequently, the choice of problem size often hinges on the available processor count, rather than assuming a fixed problem size. Instead of assuming a constant problem size, one can equally validly assume a fixed parallel execution time. This change of paradigm aims at measuring the scaling ability while keeping the load for each processor constant, varying the number of processors and, consequently, the size of the problem, which is distributed among the processors in Ahmdal's law. With an increase in system size, the problem size scales up to uphold a consistent parallel execution time. Gustafson further contends that the serial portion of the code generally remains unchanged and doesn't expand alongside the problem size. Due to these reasons, we could affirm that this algorithm might be part of the family of embarrassingly parallel computation algorithms. The resulting speedup is called *scaled speedup factor*:

$$S_s(N) = N + (1-N)ft_s$$

where t_s is the sequential time and f is the fraction of code that can't be parallelized. This formula implies two main assumptions: the parallel execution time is constant, and the part that must be executed sequentially, ft_s, is also constant and not a function of N. According to the aforementioned formula, the theoretical maximum scaled speedup factor should be (on an EpiTO node using all 80 cores and a $f = 0.02$):

$$S_s(80) = 80 + (1-80) * 0.02 = 80 - 79 * 0.02 = 80 - 1,48 = 78,42 \approx 78$$

3.3 An embarrassingly parallel computation

As previously mentioned, it may appear that the algorithm is not particularly well-suited for parallelism. However, it's worth noting that a significant portion of the algorithm's operations are highly amenable to parallelization. Many of these operations can be parallelized independently, without the need for communication between parallel threads. Examples of such operations include matrix multiplication and matrix transposition. Nevertheless, there are certain computations that do require communication, such as when calculating the Euclidean norm. In this case, a mapping operation must first be performed to compute the squares of the elements, followed by a reduction operation. However, it's important to highlight that these communication-intensive parts of the code are not the most computationally expensive ones.

4 OpenMP

4.1 Sequential implementation

Since we were unable to locate an open-source implementation of the Karmarkar algorithm, our initial step was to create our own version of the algorithm, which can be found in detail in 1. This process took some time as we wanted to ensure its functionality and accuracy. To accomplish this, we established a dedicated namespace named "AlgebraKit." This namespace was organized to prevent any conflicts with the existing CUDA implementation of the algorithm. Specifically, for the sequential component, we implemented each operation required by the algorithm. 1

4.2 Implementation with OpenMP

OpenMP is a framework that empowers programmers to parallelize a sequential codebase, starting from its original sequential implementation. This is achieved through the utilization of precompiler directives, which are ignored by the compiler if certain compiler parameters are left out. This flexibility enables the maintenance of a unified codebase that caters to both the sequential and parallel versions of an algorithm.
The implementation of the algorithm using OpenMP is relatively straightforward. Due to OpenMP's use of precompiler directives, we were able to seamlessly adapt the existing code used for the sequential algorithm. In most cases, it merely required the addition of the following preprocessor directive:

```
#pragma omp parallel for
```

Which is the directive to tell OpenMP that the following code section should be parallelized. In particular, it is possible to see the parallelized sections of code in the following methods:

- AlgebraKit::cholesky_decomposition()
- AlgebraKit:.u_solve()
- AlgebraKit::ut_solve()
- AlgebraKit::diag()
- AlgebraKit::e()
- AlgebraKit::init_b()
- AlgebraKit::product_by_scalar()
- AlgebraKit::ones()
- AlgebraKit::euclidean_norm()

All these operations, developed alongside basic matrix operations, allowed us to have a direct link with the operations involved. For example, the Cholesky solver is way more efficient than a classical linear solver, and we developed from scratch `cholesky_decomposition()` ,`u_solve` and ut_solve in order to accomplish this task. As a consequence, the parallelization with OpenMP has resulted to be more straightforward and capillary. OpenMP offers a wide range of configurable parameters that can be adjusted by adding options to the precompiler instructions. In our pursuit of optimizing performance, we conducted experiments with various parameters, including num_threads and the shared()[3] option, among others. Surprisingly, we found that the default values consistently delivered the best performance results.
It's worth noting that there's an essential function not mentioned in the previous list, namely AlgebraKit::sumReduce(). This function serves the purpose of reducing an array by applying a summation operation to all its elements. OpenMP provides a dedicated directive designed for parallelizing such operations:

```
#pragma omp parallel for reduction(+:sum)
```

The reduction() option takes two arguments, separated by colon: the first one can be one of

```
{ +, -, *, &&, ||, &, |, ^}
```

and the second option is the variable, in which the result is stored.
Synchronization is another thing missing from the code. That is because OpenMP implicitly synchronize with a barrier at the end of each parallel section of the code, hence no barrier were required to be explicitly declared.

Overall the implementation with OpenMP was straightforward and did not require a lot of effort (that has been very different from the CUDA implementation, which will be the topic of future sections).

4.3 Conclusions on OpenMP implementation

To conclude, the implementation of the algorithm with OpenMP was very fast to achieve, tough due to two main problems, it is likely not the most optimized. this is due to two main reasons:

1. The algorithm has been implemented so that each operation defined inside AlgebraKit, allocates a new object that stores the result of the operation. This is not the best, because the new and the delete operators are not super-efficient. It has later been added the option to overwrite the original matrix with the result, instead of returning a new object. This has proven in some test to be slightly faster but, due to the fact that the improvement of the overall performance were marginal, the algorithm still uses the old version[4].

[3] used to define a shared segment of memory between all threads in OpenMP
[4] Although it is still present the option to use dynamic programming.

2. OpenMP does not use a thread pool, and instead spawns new threads every time it reaches a new parallel section. The overhead of thread spawning is significant, and a thread pool approach might be more suited to gain more performance.

5 Cuda

CUDA, short for Compute Unified Device Architecture, is a framework developed by NVIDIA. It serves as a platform for creating Single Instruction, Multiple Data (SIMD) programs designed to run on NVIDIA GPUs. CUDA acts as an extension of C++, fully compatible with C (similar to openMP). With CUDA, programmers must define kernels that execute on CUDA-enabled devices. NVIDIA also provides additional frameworks built on top of CUDA to simplify programming tasks. Notably:

- CuBLAS (CUDA Basic Linear Algebra Subsystem): CuBLAS offers a collection of routines that implement basic, highly efficient linear algebra operations.
- CuSOLVE: CuSOLVE is a set of routines built on CUDA, providing tools to solve linear equation systems. One of the solvers in CuSOLVE utilizes Cholesky decomposition, similar to the OpenMP version.

In our project, we leveraged these frameworks extensively. Specifically, we employed CuBLAS for efficient and fast matrix multiplication kernels and Cu SOLVE for solving linear equation systems. We turned to CUDA to develop custom kernels when neither Cabals nor Cu SOLVE offered the necessary functionality. Regarding the CUDA code, it resides within the CudaAlgebraKit, where we implemented the following operations:

- **e(double *d_sol, int n)**: this method creates a vector of length n, in which all elements are initialize to $\frac{1}{n}$;
- **diag(double *d_diag, double *d_sol, int n)**: this method converts a vector of size n, to a matrix of size $n \times n$, for wich the following equation is valid: $\forall (0 < i < n), d\_diag[i][i] = d\_sol[n]$
- **ones(double *x, int x_len)**: This method creates a vector of length n, in which all elements are set to 1;
- **diff(double *x, double *y, int x_len)**: This methot performs the difference between two array, in the following manner: $y = x - y$;
- **transpose(double *odata, double *idata, int width, int height)**: This method performs the transpose of matrix idata, and stores it inside odata;

5.1 Shared Memory

Though shared memory is extremely important to increase the performance of the kernel, it has not been used in our defined kernels, except for the transpose kernel. There are two reasons for not using shared memory:

- Frequency of kernel invocation: most of the kernels that we defined are called a single time, in the warm up phase. That is they are executed only a single time during the problem resolution.

- Difficulty in choosing the size of the shared memory to allocate: since we do not have prior knowledge of the size of the arrays / matrices, it has been difficult to efficiently design kernels that uses shared memory. Our tests failed miserably and so we choose not to implement shared memory inside them. For the transpose kernel, for which we had examples, and for which we knew the size of the tiles that were used, we implemented shared memory.

5.2 cublasDgemm

cuBLAS DGEMM stands for CUDA Basic Linear Algebra Subprograms for Double-precision General Matrix-Matrix multiplication. It is an optimized library for performing matrix-matrix multiplications with double-precision (64-bit) floating-point numbers on NVIDIA GPUs (Graphics Processing Units). Matrix-matrix multiplication, often denoted as:

$$C = (\alpha * A) \cdot B + \beta * C,$$

is a fundamental operation in linear algebra and it's particularly suitable for GPUs. In CUDA and GPU programming in general, the choice of data storage formats for vectors is the column-major format. While this format has some advantages, it can be counterintuitive due to cache constraints, especially because in languages like C++ the more familiar row-major format is used. In fact, in the column-major format, a two-dimensional array or matrix is stored in memory by storing each column sequentially. As a consequence, elements which are supposed to be contigous in memory, in the reality they are not contigous. The column-major format has been very counterintuitive for us, since the other implementation was thought and implemented in row-major format. However, it's been fundamental to enable the Cublas acceleration on GPUs. Notice that, in addition to using a data storage different from the common one, the CublasDgemm flags must be arranged taking into account the implicit transpose of the matrices, and each dimension must be set accurately taking into accounts many aspects. The deployment of this operation, despite the documentation available, has not been straightforward, and took us most of two weeks to fully understand.

6 Results

In this section, we are going to talk about our results. In particular, we will comment on the weak and strong scalability of our algorithm with openMP and weak scalability for CUDA.
The experiments were run on the EpiTO cluster, which consists of

- 80 ARM cores,
- 512GB RAM,
- 2 NVIDIA A100 GPUs with 40GB of memory,

and the part of interest of the architecture, is summarized in the figure 2 obtained with the command `lstopo`

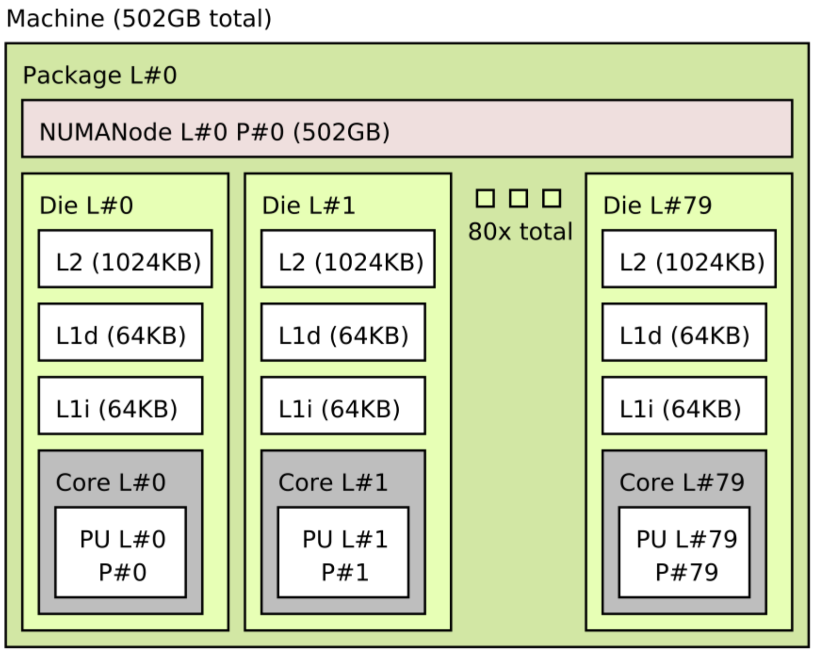

Figure 2: Structure of the main CPU of an EpiTO node.

To generate the random matrix we used MatLab, randomizing the entries and considering a limited number of rows to decrease the size of the problem. Notice that from 1, we had to choose $m < n$, otherwise the problem becomes a closed solution of a linear system. The correctness of our implementation was tested on a toy problem present at this link, and by checking the result of other small problems, with the results delivered by scilab karmarkar implementation (discussed further in this report).

6.1 OpenMP

First of all, let's comment on the OpenMP implementation. Analyzing Figure 3, on the left we have the average time per iteration, with an increasing dimension of the problem as well as the number of processors. To increase the number of processors we used an OpenMP flag that activates a precise number of threads.

Processors	Dimension of the matrix	Mean Average time per iteration
1	78×5000	4.9808 ± 0.0087
2	156×5000	4.9785 ± 0.0037
5	312×5000	4.8127 ± 0.0233
10	625×5000	5.1922 ± 0.0370
20	1250×5000	6.1954 ± 0.1140
40	2500×5000	8.2032 ± 0.2040
80	4999×5000	22.3701 ± 0.6638

Table 1: Result of the parallelization with OpenMP.

On the right part of Figure 3, the efficiency is simply the following:

$$Eff(N) = \frac{t(1)}{t(N)}$$

where N is the number of processors. However, as you can see from Table 3, the dimensions of the matrix are halved at every different number of processors. On the other hand, from 5 to 2 the computational resources are not halved. To overcome this issue that results in having $Eff(5) > 1$, we weighted both $t(1)$ and $t(N)$ with the scaling factor $\frac{N}{rows}$. This resulted in the scaling plot in the right part of Figure 3.

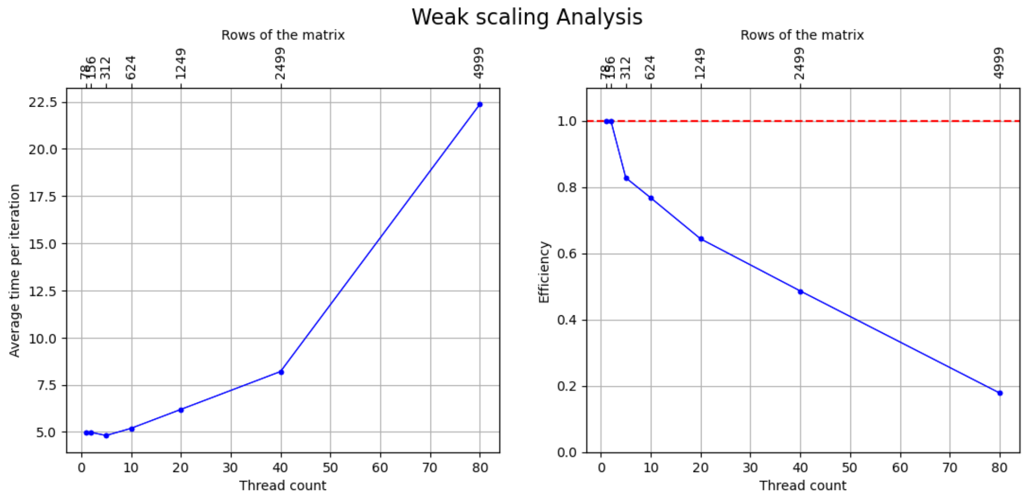

Figure 3: Weak scaling with OpenMP.

As we said, the scaling behavior does not seem to perform very well. This might be due to the parallelization of the single operations, which is probably a sub-optimal solution with respect to the parallelization of the whole

algorithm. However, being an iterative procedure, this is the best we could get.
On the other hand, in Figure 4, we tested Ahmdal's law with a fixed problem size of 4999×5000, increasing the number of processors. The number of processors follows the powers of 2 until 64, obtaining the following results:

Processors	Dimension of the matrix	Speedup
1	4999×5000	$633.2196 \pm 5,7423$
2	4999×5000	303.7588 ± 4.3670
4	4999×5000	140.8122 ± 0.7553
8	4999×5000	64.9885 ± 0.1790
16	4999×5000	32.1253 ± 0.7268
32	4999×5000	21.6984 ± 1.4756
64	4999×5000	22.8599 ± 0.5564

Table 2: Result of the parallelization with OpenMP.

Where the speedup is the same as the efficiency in the weak scaling analysis.

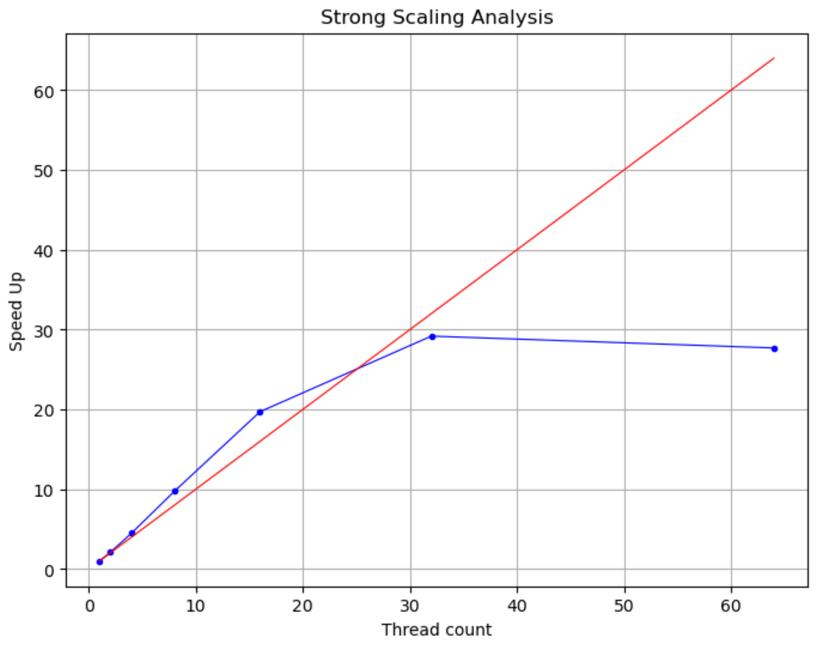

Figure 4: Strong scaling with OpenMP.

As we can see clearly from Figure 4, the behavior of this algorithm is super linear. This might be due to two factors:

- A sub-optimal implementation of the sequential algorithm;
- Hardware characteristics that might improve performance.

Since the codebase for both sequential and OpenMP is the same we are not prone to think that the cause is the sub-optimality of the sequential implementation. We are more likely to think that cache memory is to blame: it is possible to observe in Figure 2, EpiTO nodes have a lot of cache memory, which means that fewer memory accesses need to be done by the CPU, hence avoiding the memory access overhead that happens when a serial implementation is used. This conclusion is aligned with others present in the literature [10], and superlinear behaviors have also been observed in other studies [9].
To conclude, the parallelization of this algorithm with OpenMP brought a good strong scaling, enabling an almost 30x speed up. However, the weak scaling behavior has shown only decent performances, probably due to the parallelization of the single operations instead of a whole restructured algorithm, due to the iterative nature of Karmarkar's algorithm.

6.2 CUDA

For what concerns the performances with CUDA, we tested only weak scaling behavior since strong scaling is not straightforward to be checked [4] and there are many recent studies about this topic [2, 3].
For this study, we kept constant the grids and only increased the dimensions of the problem, resulting in the following performances:

Dimension of the matrix	Average time per iteration
10×5000	2.3658 ± 0.2402
18×5000	2.3654 ± 0.2401
39×5000	2.3656 ± 0.2403
78×5000	2.3660 ± 0.2407
156×5000	2.3663 ± 0.2411
312×5000	2.3675 ± 0.2425
625×5000	2.3690 ± 0.2419
1250×5000	2.3724 ± 0.2423
2500×5000	2.3814 ± 0.2405
4999×5000	2.4105 ± 0.2391

Table 3: Result of the parallelization with CUDA.

As we can see the absolute time is way less than the best performances shown so far. The time is almost constant also with the growing size of the problem, which confirms that GPUs are very efficient even with a huge increase in the problem size. The Efficiency shown is indeed almost 1 even with the biggest size of the problem, as shown in Figure 6.

With our analysis we can highlight empirically a well-known behavior of the GPUs: they are very efficient when the size of the problem is big enough so that the communication time is a small portion of the total computation time. If the problem is not big enough, then the GPU spends most of the time doing data transfer, and at this point, it is more efficient to compute the result with CPUs. This can be seen in Figure 5 which shows computation/communication time over the size of the problem.

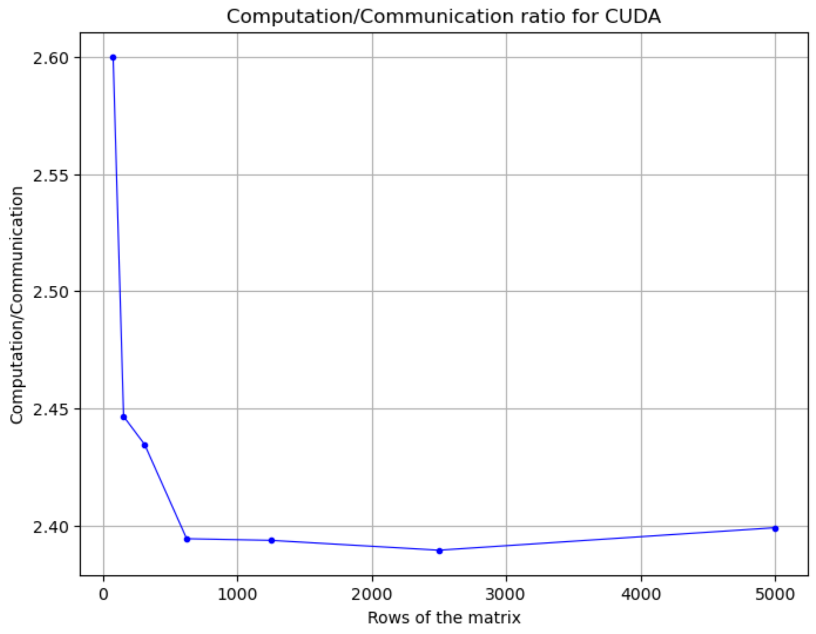

Figure 5: $\frac{Computation\ time}{communication\ time}$ over dimension of the matrix

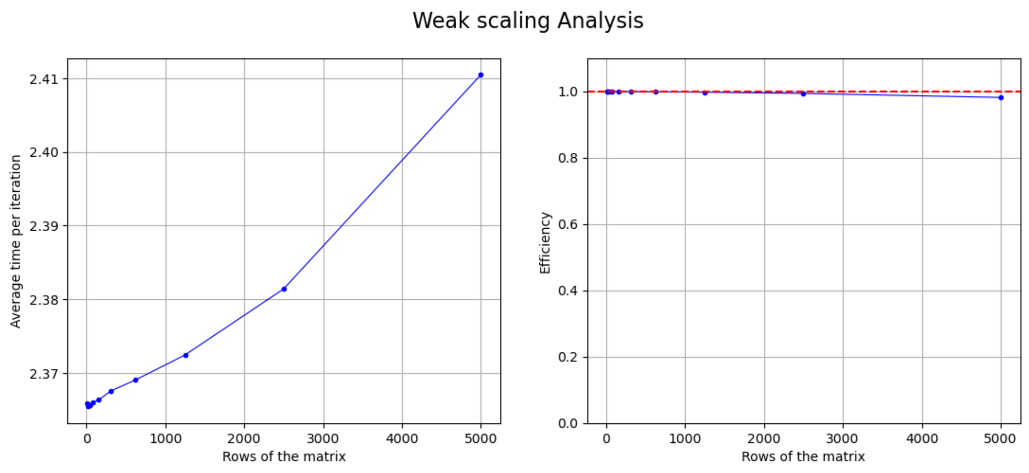

Figure 6: Weak scaling with CUDA.

Looking more accurately at Figure 6, we can see that for small problems the performances decrease:

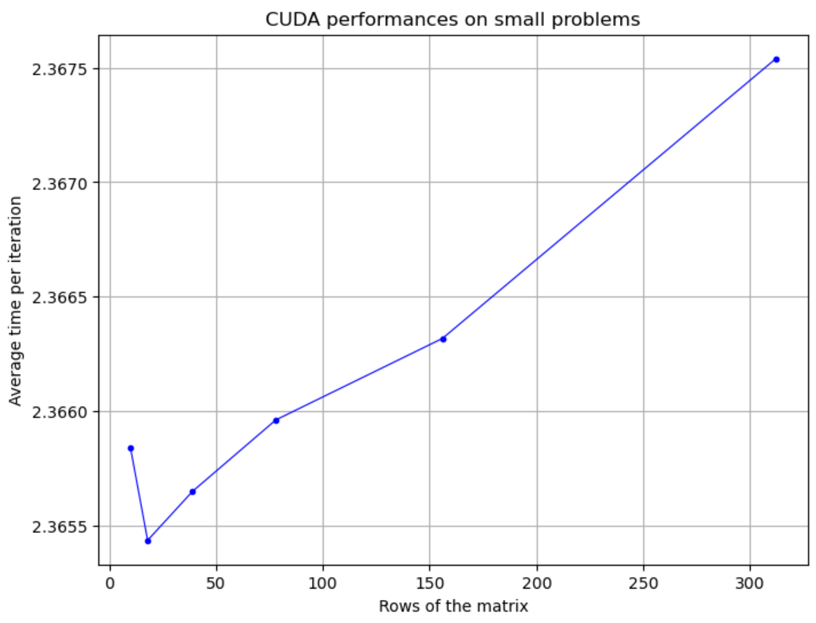

Figure 7: Weak scaling CUDA for small dimensional problems.

As we can see from 7, when the problem is very small using a GPU does not affect the computation, and may result even in a decrease in the performance. In fact, from Figure 7 the first three points on the left have almost the same time. The growing dimension of the problem affects negatively the computation, which suggests that working with GPUs on small-size problems is not recommended.
To conclude this section, we put the overall results[5]:

	times (seconds)	speedup
Sequential	633.22 ± 5.74	-
OpenMP	22.37 ± 0.66	≈ 28
CUDA	2.41 ± 0.24	≈ 260

Table 4: Final results on the biggest matrix (4999 rows and 5000 columns) in the best settings for OpenMP (80 cores) and CUDA.

We took the best performances on the biggest matrix resulting in a first SpeedUp of 30x with the OpenMP version with respect to the sequential version, and a final SpeedUP of 300x when using CUDA with respect to the sequential version. This suggests that, when the problem is big enough the use of a GPU is justified.

[5]Maximum speedup for Ahmdal's law is 31

7 Comparison with Scilab

While searching for other implementation of karmarkar algorithm, we found that scilab [6] had a sequential implementation of the algorithm. We run an experiment with the same matrix used in our experiment and we found that, on average, it took 35 ± 2 minutes for each iteration. It needs to be shown that this test has not been run on the same hardware in wich our tests were run (EpiTO), and that after few iterations, the program was terminated due to resource exhaustion. Another drawback of the implementation of scilab was ram usage, wich was constantly above 8GB It was unfortunately impossible to compare the parallel version of the algorithm, as we did not found a parallel implementation of the algorithm. In comparison our sequential implementation, ran on the same hardware of scilab, took 20 minutes for iteration, and ram usage was below 3GB.

Implementation	Time per iteration (minutes)	Ram Usage
Scilab	35 ± 2	8GB
Our(sequential)	20 ± 1	3GB

Table 5: Comparison wit Scilab implementation.

8 Conclusions

OpenMP parallelization, yields a significant improvement, even tough it is not as high as what can be achieved with dedicated GPUs. This is useful as it allows to execute the algorithm in a reasonable time even on devices with "limited" resources, tough it has to be noted that the speedup achieved in comparison with the CUDA is way smaller for problems of big size. Another useful application of the OpenMP algorithm is for those problems that on one hand are not big enough to make sense being optimized for a GPU, but on the other hand are not enough small to be run sequentially in reasonable time. OpenMP allows for a very straightforward way to parallelize the execution and optimize the performances, especially for detailed algorithms like the one we treated in this report. In fact, since it was made of many single operations that could be optimized, without the possibility to rethink the algorithm in a complete parallelized way, OpenMP provided the best opportunity to optimize the performances (taking into account the possible unavailability of a GPU). In particular, OpenMP also allows for the algorithm to be run on small laptops, and have discrete and tangible benefits.
On the other hand, CUDA has revealed to be an excellent choice to parallelize the algorithm. Due to design choiche made by NVIDIA in CUDA, develop code using CUDA is not straightforward, but the fact that the problems treated with the algorithm might be extremely large, the difficulty of

[6]SCILAB official website

programming with CUDA, is an acceptable drawback, in obtaining high performance increase.
To conclude, our parallelized versions of the Karmarkar's algorithm have showed important improvements with respect to the sequential version, which were discussed in this report, alongside a more theoretical treatment of the problem both from the algorithm and a parallel computing point of view.

References

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[2] Jaemin Choi, David F Richards, and Laxmikant V Kale. Improving scalability with gpu-aware asynchronous tasks. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 569–578. IEEE, 2022.

[3] Preyesh Dalmia, Rohan Mahapatra, Jeremy Intan, Dan Negrut, and Matthew D Sinclair. Improving the scalability of gpu synchronization primitives. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):275–290, 2022.

[4] David Eberius, Philip Roth, and David M Rogers. Understanding strong scaling on gpus using empirical performance saturation size. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 26–35. IEEE, 2022.

[5] Shu-Cherng Fang and Sarat Puthenpura. *Linear optimization and extensions: theory and algorithms*. Prentice-Hall, Inc., 1993.

[6] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[7] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311, 1984.

[8] Leonid Genrikhovich Khachiyan. A polynomial algorithm in linear programming. In *Doklady Akademii Nauk*, volume 244, pages 1093–1096. Russian Academy of Sciences, 1979.

[9] Gregor Kosec and Matjaz Depolli. Superlinear speedup in openmp parallelization of a local pde solver. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 389–394. IEEE, 2012.

[10] Sasko Ristov, Radu Prodan, Marjan Gusev, and Karolj Skala. Superlinear speedup in hpc systems: why and when? In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 889–898. IEEE, 2016.

[11] Gilbert Strang. Karmarkar's algorithm and its place in applied mathematics. *The Mathematical Intelligencer*, 9(2):4–10, 1987.

[12] Philip Wilkinson. *Parallel programming: Techniques and applications using networked workstations and parallel computers, 2/E*. Pearson Education India, 2006.