IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

Benchmarking Parallelization Models through Karmarkar’s Interior-point method

This is a pre print version of the following article:

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/1964571 since 2024-03-25T13:51:27Z

Publisher:

IEEE

Published version:

DOI:10.1109/PDP62718.2024.00010

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

28 April 2024

Benchmarking Parallelization Models through Karmarkar's Interior-point method

Marco Edoardo Santimaria
Computer Science Dept.
University of Torino, Italy
marcoedoardo.santimaria@unito.it

Samuele Fonio
Computer Science Dept.
University of Torino, Italy
samuele.fonio@unito.it

Giulio Malenza
Computer Science Dept.
University of Torino, Italy
giulio.malenza@unito.it

Iacopo Colonnelli
Computer Science Dept.
University of Torino, Italy
iacopo.colonnelli@unito.it

Marco Aldinucci
Computer Science Dept.
University of Torino, Italy
marco.aldinucci@unito.it

***Abstract*—Optimization problems are one of the main focus of scientific research. Their computational-intensive nature makes them prone to be parallelized with consistent improvements in performance. This paper sheds light on different parallel models for accelerating Karmarkar's Interior-point method. To do so, we assess parallelization strategies for individual operations within the aforementioned Karmarkar's algorithm using OpenMP, GPU acceleration with CUDA, and the recent Parallel Standard C++ Linear Algebra library (PSTL) executing both on GPU and CPU. Our different implementations yield interesting benchmark results that show the optimal approach for parallelizing interior point algorithms for general Linear Programming (LP) problems. In addition, we propose a more theoretical perspective of the parallelization of this algorithm, with a detailed study of our OpenMP implementation, showing the limits of optimizing the single operations.**

***Index Terms*—Optimization problems, stdblas, PSTL, GPU programming, parallel computing, Linear programming.**

I. Introduction

Linear optimization problems have been a flourishing research area for computer scientists since their introduction [13], [10]. Determining the optimal value of a linear objective function with linear constraints within computationally feasible time constitutes the main goal of the development of algorithms for LP problems, which are prevalent in diverse industrial domains with numerous practical scenarios [29], [23], [27], [30], [11]. Despite its wide range of applications, the research in LP problems is built upon two main algorithms. The first one, introduced in 1947, is the Simplex method. Despite its mathematical simplicity, it has been the leading method for many years, and still today specific variations of this algorithm have shown leading performances in different types of problems [13]. However, when it comes to large-scale problems, the simplex method tends to perform poorly because of its exponential worst-case complexity [21]. To overcome this issue in 1984 it's been introduced the second main algorithm: Karmarkar's Interior point method [19].

Karmarkar's algorithm outperformed the state-of-the-art of that time, such as the Ellipsoid method [20], and yielded a primary competitor to the Simplex method, serving as the foundational method for extensive further investigations and studies [15], [5]. The key point of Karmarkar's algorithm is to redefine linear problems as non-linear, which in return enables progress toward the minimum without testing each possible solution. In fact, Interior points methods are used both for linear optimization [28], [31] and for non-linear optimization [7].

The key distinction from the Simplex method is that each interior-point iteration is computationally expensive but makes significant progress toward the solution, whereas 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.

It's been widely studied in the literature that LP solvers can benefit from parallelized operations, especially on GPUs. In fact, parallel versions of the Simplex method are already available [26], [22], as well as for the Revised Simplex Method [9], [6]. Interior point methods benefit consistently from parallelization since their structure involves more computationally expensive operations. In fact, GPU implementations of the interior-point methods have already been proposed [14], [24], [18], confirming that GPU acceleration boosts consistently the performances for Karmarkar's algorithm and in general interior-point methods. However, different parallel programming models yield different performances, depending also on the available hardware. In this work, we want to analyze the parallel performance of Karmarkar's algorithm using OpenMP, CUDA, and the C++ parallel standard template linear algebra library (PSTL-stdblas) [17]. In addition, to the best of our knowledge and effort, we were not able to find any parallel, open-source implementation of this algorithm.

As a consequence, the contribution of this work is twofold:

- We provide a comparison between different parallelization schemes when applied to Karmarkar's LP solver: OpenMP, GPU-acceleration with CUDA and PSTL;

- we provide detailed results about the scaling behavior for the OpenMP implementation, providing insights on the theoretical perspective of the parallelization of Karmarkar's algorithm.

With this work, we want to bring attention to the comparison between different possible parallel implementations that may require various levels of effort and consequently bring different levels of performance improvements.

The paper is structured in the following way: after a section of related works, we present in detail Karmarkar's algorithm with all the operations involved. Then, we explain our parallelized version of the algorithm with OpenMP, CUDA, and PSTL. Consequently, we show the results of a simulated large-scale problem, involving a study on the scaling property of the OpenMP implementation.

II. Related Works

Linear Programming research is a flourishing field, and the parallelization of algorithms for solving linearly constrained problems is very important from a practical and theoretical point of view.

The literature related to the Simplex method is particularly rich. Our work presents some similarities with [32], where it is shown that single operations involved in the simplex method can be consistently improved with sophisticated programming frameworks. However, while they introduce a detailed study and a novel implementation of a single task (Basis Update) in the Simplex method, we focus on the whole implementation of Karmarkar's algorithm.

In a similar way, but always related to the Simplex method, in [25] the authors focus on different parallel schemes implementations of the Simplex method enabling GPU acceleration and hybrid CPU/GPU approach. In [22] they provide a multi-gpu implementation of the Simplex method.

Taking inspiration from these works, our main focus is Karmarkar's interior-point method, for which different operations are involved and represent a more reliable solution to large-scale problems. Parallel implementations can boost the performances thanks to recent framework implementations [32], and our goal is to empirically show that this is possible through modern parallelization techniques.

In [1], an efficient implementation of Karmarkar's algorithm is presented, describing a family of interior point power series affine scaling algorithms, that are variants of Karmarkar's original algorithm. In particular, they were able to reach the performances of the simplex method on publicly available datasets. Their implementation was based on FORTRAN and built on specific code intended primarily for the solution of constrained non-linear programming problems. We mainly focus on the original algorithm rather than its variants, since we focus on simulated problems as in [22] and concentrate on the comparison between different parallel schemes. A parallel implementation was already available in 1988 [34], but was developed in Occam2 and at the best of our effort, we could not find the source code. Our work builds upon C++ with a specific implementation for GPU accelerators programmed with CUDA and PSTL.

To conclude, with respect to the existing literature, we fill the gap in benchmarking parallelization techniques for solving LP problems through Karmarkar's interior-point method, which enables consistent improvement in performance but is not deeply studied in the existing literature and lacks an open-source implementations.

III. Algorithm

In this section, we are going to present Karmarkar's interior-point method [19]. Karmarkar's algorithm is an interior-point method for which 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 [33], [12]. To be solved with this algorithm, a problem must be in *canonical form*, that is: for a given dimension n we have $c, x \in \mathbb{R}^n$, $A \in \mathbb{R}^{m,n}$, $\mathbf{1} = (1, \dots, 1)^t$, $\mathbf{e} = (1/n, \dots, 1/n)^t$:

1) Constraints are the following:

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

2) $v = 0$;
3) $A\mathbf{1} = 0$;
4) The rows of A are linearly independent.

Karmarkar's algorithm operates in the n-dimensional unit simplex. A n-dimensional unit simplex S is the set of points $(x_1, x_2, \dots, x_n)$ satisfying:

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

Karmarkar's algorithm wants to generate a sequence of points $x^0, x^1, x^2, \dots, x^k$ 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 creation of a direct path towards the minimum, optimizing the objective function and weighting properly the length of the steps. To do so 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.

A. Projection

As we said, the goal is to generate a feasible point x^1 from another feasible point x^0, that for a fixed vector c, will have a smaller value than $c^t x^0$. The direction towards

x^1, $d = (d_1, d_2, \ldots, d_n)$ is identified in the solution of the following optimization problem:

$$\begin{aligned} \min_{d} \quad & c^t d, \\ Ad &= 0, \\ d_1 + d_2 + ...d_n &= 0 \\ \|d\| &= 1. \end{aligned}$$

The direction d that solves this optimization problem is the projection of c onto X satisfying $Ax = 0$ and $x_1 + x_2 + ... + x_n = 0$ and is given by solving $BB^t w = B\bar{c}$;, 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.

B. Karmarkar's centering transformation

If x^k is a point in S, where S is the simplex, the centering transformation $f(\cdot|x_k) : S \to S$ is defined as follows:

$$f(x|x_k)_j = y_j = \frac{\frac{x_j}{x_j^k}}{\sum_{r=1}^{n} \frac{x_r}{x_r^k}}.$$

The properties of the centering transformation are the following:

1) x^k is mapped into the center of the transformed unit simplex;
2) it is a one-on-one mapping from S to S;
3) A point $x \in S$ will satisfy $Ax = 0$ if $AD_k f(x\|x^k) = 0$, where $D_k = diag(x^k)$.

To recap, this centering transformation transforms the working space centering the current solution. Thanks to property 2 we can transform y^{k+1} back into x^{k+1}, and x^{k+1} will be feasible for the original LP. Furthermore, property 4 is very useful to define clearly the constraints of feasibility.

C. Karmarkar's potential function

Once we detect the right direction and after providing the centering transformation, the problem becomes the length of the step towards the solution. At this end, Karmarkar introduced the **Karmarkar's potential function**:

$$\psi(x) := n \log(c^t x) - \sum_i \log(x_i).$$

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 $k = 0, 1, 2, \ldots$

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

This means that the decreasing difference is bounded, and it is crucial when choosing the length of the step α for a feasible solution, which becomes:

$$\hat{\alpha} = \arg\min_{\alpha} \bar{\psi}(e + \alpha d), \tag{2}$$

where $\bar{\psi}(x) = n \log(\bar{c}^t x) - \sum_i \log(x_i)$, $\bar{x}$ is the current solution, $\bar{c} = cdiag(\bar{x})$ and $\bar{A} = Adiag(\bar{x})$. Thanks to this α we are able to detect the correct step for not exiting the solution space. However, due to simplicity, we decided not to optimize this α and to keep a constant step of:

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

where r is the ray of the maximum sphere included in the affine space of 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,ϵ);
  Let $x := \mathbf{e}$
  $\hat{\alpha} := \alpha r = \frac{n-1}{3n} \frac{1}{\sqrt{n(n-1)}}$;
  while $c^t x > \epsilon$ **do**:
    $\bar{c} := c^t diag(\bar{x})$;
    $\bar{A} := Adiag(\bar{x})$;
    $B^t := (\bar{A}^t, \mathbf{1}^t)$;
    solve $BB^t w = B\bar{c}$;
    $\bar{c}' = \bar{c} - B^t w$
    $d = -\bar{c}' / \|\bar{c}'\|$;
    $y := \mathbf{e} + \hat{\alpha} d$;
    $x' := diag(x)y / (\mathbf{1}^t diag(x)y)$;
    $x := x'$;
  end while
  output(x);

As we can see there are many parts of the algorithm that can be optimized. For 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 $\mathcal{O}(n^{3.5} L)$ operations on $\mathcal{O}(L)$-digit numbers, as compared to $\mathcal{O}(n^6 L)$ such operations for the ellipsoid algorithm [20].

IV. Parallelization

Thanks to recent research and development, many parallelization techniques are available on every kind of hardware and in every programming language, empowering the capacity to use the total computation capability of a computer. As a consequence, the optimization of a code should be the main focus of a good programmer, as the demand for greater computational power keeps growing [35]. Our work mainly builds upon C++ implementations, providing a flexible framework for Karmarkar's Interior-point method.

In particular, for what regards our experience with this algorithm, at first glance, it appears not to be parallelizable. This is due to the dependent iterations, and thus it is impossible to start multiple iterations simultaneously. If we look more closely at the algorithm, we can see that most of its steps are actually extremely prone to parallelization. As a consequence, in our implementation, we focused not on running multiple iterations at the same time, but on parallelizing each step of the algorithm.

This methodology is in line with [2] and [8], where it is explained how to parallelize an end-to-end application. In particular, loops are numerous in Algorithm 1, and we want to compare different techniques to parallelize them, both with huge and minimal code-redesigning, namely CUDA implementation and OpenMP/PSTL implementation.

For each parallelization model, we implemented Algorithm 1 and every operation involved, using when possible functions provided in the available libraries, as we will explain in this section.

A. OpenMP

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 on github[1]. At this end, we developed from scratch the namespace `Algebrakit`with all the operations involved.

2) OpenMP Implementation: The strategy we followed for the implementation with OpenMP, has been to parallelize the for loops whenever there was no data dependency. To this end, every operation had to be implemented from scratch. This strategy allowed us to maintain a single codebase for the sequential implementation and the OpenMP implementation. In fact, due to OpenMP's use of precompiler directives, we were able to seamlessly adapt the code used for the sequential algorithm. 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 several parameters. However, we found that the default values consistently delivered the best performance results.

The implementation with OpenMP was as simple as it was effective. In fact, we will show the overall good performances reached, and focus on the resulting weak scaling and strong scaling behavior. In addition, it allowed us to study in detail the scaling behavior both for Ahmdal's [4] and Gustavfson's law [16]. Despite it's been useful for studying the parallelization of the algorithm, to boost consistently the performances leveraging maintained libraries is necessary.

B. CUDA

CUDA [2], short for Compute Unified Device Architecture, is a framework developed by NVIDIA. It serves as a platform for creating Single Instruction, Multiple Thread (SIMT) programs designed to run on NVIDIA GPUs. CUDA acts as an extension of C++, fully compatible with C (similar to OpenMP). The main elements of CUDA are the kernels: and functions to be executed on CUDA-enabled devices. NVIDIA also provides additional frameworks built on top of CUDA to simplify programming tasks. Notably:

- CuBLAS[3](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.

In our project, we leveraged these frameworks extensively. Specifically, we employed CuBLAS for efficient and fast matrix multiplication kernels and CuSOLVE for solving linear equation systems. We turned to CUDA to develop custom kernels when neither CuBLAS nor CuSOLVE offered the required functionality. The complexity of our implementation, for each iteration, is $\mathcal{O}(n^3)$, and in general is:

$$8\mathcal{O}(n) + 3\mathcal{O}(n^2) + 3\mathcal{O}(n^3) = \mathcal{O}(n^3)$$

The cubic component of the complexity of our algorithm is given by the LEVEL 3 BLAS: two calls are done to `cusolverDnDpotrs` and a single call is done to `cublasDgemm`. Some of the level 2 BLAS functions that we used are `cublasDdot` and `cublasDgemv`, while some of the level 1 blas functions that we used are `cublasDscal`, `cublasDdot` and `cublasDnrm2`.

We show here the code for the body of the main iteration of the algorithm written in CUDA.

```
sym_rank1<<< ... >>>(c_vec_dev,x_vec_dev,Ncol);
sym_rank2<<< ... >>>(A_vec_dev,x_vec_dev,Nrow,Ncol);
cudaMemcpy(B_vec_dev, A_vec_dev, ... );
cublasDgemv( ... ,Ncol,Nrow+1,&ONE,B_vec_dev,Ncol,
    c_vec_dev,1,&ZERO,c_vec1_dev,1);
cublasDgemm( ... ,CUBLAS_OP_N,m,n,k,&ONE,B_vec_dev,k
    ,B_vec_dev,k,&ZERO,BB_vec_dev,m);
cusolverDnDpotrf( ... , n, BB_vec_dev, n, buffer,
    bufferSize, info);
cudaMemcpy(w_vec_dev, c_vec1_dev,  ... );
cusolverDnDpotrs( ... , uplo, n, 1, BB_vec_dev, n,
    w_vec_dev, n, info);
cublasDgemv( ... ,Ncol,Nrow+1,&MONE,B_vec_dev,Ncol,
    w_vec_dev,1,&ONE,c_vec_dev,1);
cublasDnrm2(handle, Ncol, c_vec_dev, 1, &norm);
double tmp=-alpha/norm;
dsum<<< ... >>>(d_vec_dev,1.0/Ncol,Ncol);
cublasDaxpy(...,Ncol,&tmp,c_vec_dev,1,d_vec_dev,1);
cublasDdot(...,Ncol,x_vec_dev,1,d_vec_dev,1,&
    norm_dot);
sym_rank1<<<...>>>(x_vec_dev,d_vec_dev,Ncol);
norm_dot=1/norm_dot;
cublasDscal(...,Ncol,&norm_dot,x_vec_dev,1);
cublasDdot(...,Ncol,co_vec_dev,1,x_vec_dev,1,&res);
cudaMemcpy(x_vec.data(),x_vec_dev, ...);
```

Listing 1. "..." means code has been omitted.

C. C++ PSTL

The C++ Parallel Standard Template library (PSTL) was officially introduced for the first time in the 2017. This standard allowed the parallel execution of algorithms specifying an execution policy:

- `std::execution::seq` → The sequenced policy (since C++17). This policy forces the execution of an algorithm to run sequentially on the CPU.

[1] Github code repository
[2] https://developer.nvidia.com/cuda
[3] https://developer.nvidia.com/cublas

- `std::execution::unseq` → The unsequenced policy (since C++20). With this policy, the calling algorithm is executed using vectorization on the calling thread.
- `std::execution::par` → The parallel policy (since C++17). This policy tells the compiler that the algorithm could be run in parallel.
- `std::execution::par_unseq` → The parallel unsequenced policy (since C++20). This policy allows the algorithm to be run in parallel on multiple threads each able to vectorize the calculation.

Proposal [17] aims to include in the standard C++26 a linear algebra layer of abstractions that allow user to use Basic Linear Algebra Subprograms (BLAS) directly from C++. In this work, we used an experimental version of stdblas provided by NVIDIA in the `hpc_sdk` toolkit [4]. We re-wrote the CUDA code using C++ parallel algorithms like `std::transform`,`std::copy` and stdblas linear algebra algorithms with `mdspan`. In particular, we used wrappers to BLAS level 3 functions like `matrix_product`, `triangular_matrix_vector_solve`, wrappers to BLAS level 2 function `matrix_vector_product` and wrappers to BLAS level 1 function `vector_norm2`, `dot`. The library can be used to offload C++ and linear algebra algorithms on GPU setting the flags `-stdpar=gpu -cudalib=cublas` at compile time. In this way the CuBLAS library is used as a back-end for linear algebra algorithms. The same code can be executing on multicore CPU adding the flags `-stdpar=multicore` and linking the OpenBLAS library[5] `-lblas`. For our experiments we used the `par` policy which allows for very intuitive and easy parallelization of the code, and the broad range of linear algebra built-in functions enables an easy usage of this framework. The complexity of our implementation is $\mathcal{O}(n^3)$, and in particular the complexity of out implementation is:

$$9\mathcal{O}(n) + 3\mathcal{O}(n^2) + 3\mathcal{O}(n^3) = \mathcal{O}(n^3)$$

The cubic component of the complexity of our algorithm is given by the two calls that are done to `stdex::linalg::matrix_product`, and by the cholesky solver.

V. Results

In this section, we are going to illustrate our results. In particular, we will comment on the weak and strong scalability of our algorithm with OpenMP and on the performances at various problem sizes for the CUDA and PSTL implementations. The experiments were run on an EpiTO [3] ARM machine, which consists of:

- Ampere Altra Q80-30 CPU (80-core Arm Neoverse N1),
- 512GB of memory,
- 2 x NVIDIA A100 GPU (40GB vram).

We compile the code with the following library versions: `gcc 12.2.0`, `nvhpc 23.5` and `CUDA 11.8`.

[4] https://developer.nvidia.com/hpc-sdk
[5] https://www.openblas.net/

Similarly to [22], we chose to benchmark the numerical aspect of our implementation rather than a real problem, by generating random positive definite matrices with Python. Moreover, from Algorithm 1, we had to choose $m < n$, otherwise the problem becomes a closed solution of a linear system.

We are going to present separately the OpenMP results and the others since the OpenMP code is not aligned with the other implementations. On the contrary, the other implementations are aligned and comparable. In addition, OpenMP is more suitable for a theoretical treatment of the parallel algorithm, since every operation was implemented from scratch, providing a naive overview of the algorithm. The tests were performed by executing 10 iterations and then computing the average time required per iteration. Furthermore, we conducted a sanity check on small problems to ensure the correctness of our implementations.

A. Sequential and OpenMP implementation

OpenMP results can provide insights for the scalability of LP problems when the parallelization involves single operations within the algorithm. Analyzing Figure 1, we have the average time per iteration, with an increasing problem size and number of processors. To increase the number of processors we used `OMP_NUM_THREADS` environment variable to control how many threads were used.

The average time per iteration in Figure 1 shows that when the number of processors grows, the time increases consistently. In this case, the number of columns was fixed at 10000 and the number of rows grew linearly: $number\ of\ threads \times 100$.

We decided to compare the OpenMP implementation with the PSTL implementation for CPU, which allows the limitation of threads used. We can observe that using existing libraries definitly improves the performances in terms of weak scalability. In particular, if the times grow consistently with a higher number of threads (and a higher dimension of the problem), for PSTL-CPU the time increases very slowly.

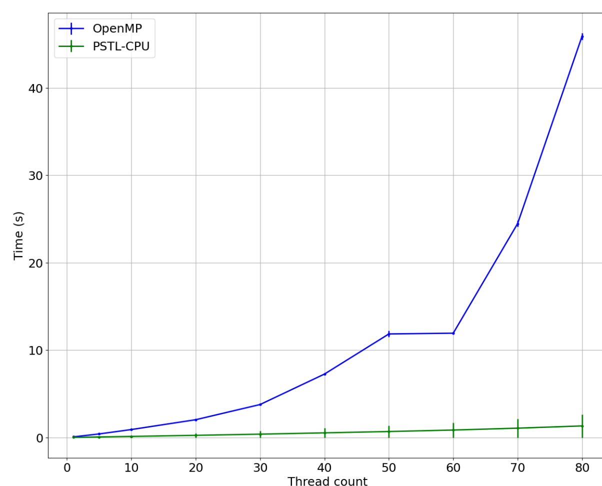

Figure 1. Weak scaling analisys of OpenMP implementation.

Since the OpenMP implementation is entirely from scratch, it presents limited results. The parallelization of the single operations instead of the whole algorithm is a limit. In fact, as we can see from Figure 1, after 10 threads the performances start to decrease.

However, we can notice from Figure 2 that the improvement in performance is not negligible with respect to using a single processor (sequential implementation), with an improvement of up to 35 times for OpenMP and 50 for PSTL-CPU.

Figure 2 shows the strong scaling behavior with a fixed problem size of 4000×8000, increasing the number of processors. Ahmdal's law evaluates whether a constant problem size impacts a growing number of processors. It is important to state that the OpenMP implementation does not use BLAS libraries. This leads to some loss in performance in comparison with the PSTL executed with multicore and OpenBLAS library.

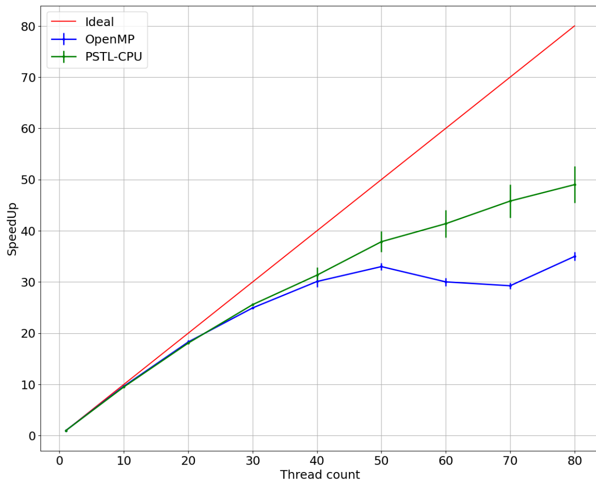

Figure 2. Strong scaling with OpenMP.

From Figure 2 it is evident the difference between our two implementations that run on CPU. In particular, the OpenMP implementation stops scaling after 50 threads, while PSTL-CPU keeps improving its performance.

From this comparison, there are several important things to highlight. Implementing from scratch the operations of this algorithm allowed us to have an overview of how it worked, with the possibility of assessing the OpenMP parallelization techniques. However, this strategy has resulted in being less performant than using PSTL, which is an advanced library that operates in the most optimized way. Consequently, it is not fair to compare OpenMP and PSTL, since the two implementations are not perfectly aligned, but we will further detail this concept in the following.

To conclude, the parallelization of this algorithm with OpenMP brought a good improvement in performance with respect to the sequential code, reaching a 35x speed up. However, the weak scaling behavior has shown only limited performances, probably due to the parallelization of the single operations instead of a whole restructured algorithm.

B. OpenMP and PSTL implementation results

In this section, we are going to discuss the overall performance of our implementations. This comparison is at the core of this study and wants to give initial insights into the impact of different parallel models when used for Interior-point methods and for Karmarkar's algorithm in particular. In our testing, we managed to use the implementation on CPU up to a matrix size of 16000 rows by 32000 columns. We could not use OpenMP on bigger matrices, as the time required to get a result wasn't feasible.

On the one hand, comparing the OpenMP implementation with PSTL on multi-core is unfair since the first one is implemented from scratch and the other one is optimized. However, we decided to put the overall results according to whether it was run on a multi-core CPU or GPU. On the other hand, as we can see from the comparison between Table I and Table II(namely plotted in Figure 3 and Figure 4), when the matrix size is small (less than $4k \times 8k$) the PSTL on CPU (PSTL-CPU) shows competitive results with its GPU counterpart (PSTL-GPU). This is due to data movement and cache effects, which is an interesting fact highlighting the possibility of reaching performances similar to GPUs if the problem is small enough and, most of all, using the same code base.

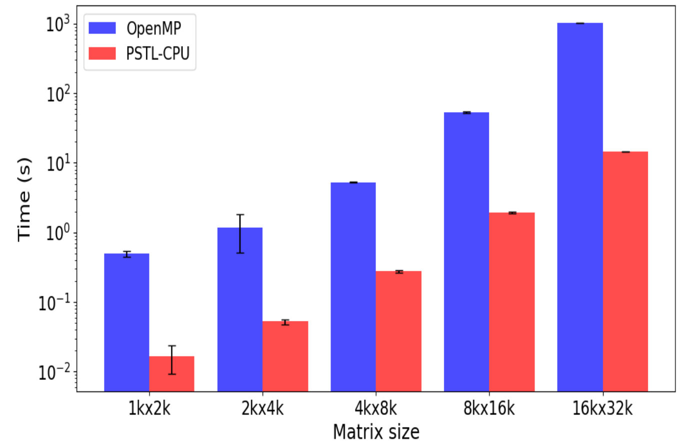

Figure 3. Results of Table I.

A fair comparison between different parallel models is not trivial since it depends on the alignment of the implementations: different implementations should imply the same operations and the same optimization level. While the first is achievable, the second represents an interesting challenge. The results shown can be considered comparable (except for OpenMP, which does not use BLAS routines), as we put much effort into aligning PSTL and CUDA code. However, for future work, our goal is to optimize our implementations further.

C. PSTL and CUDA native implementation results

In this section, we discuss the performance of the PSTL implementation against a native CUDA implementation. From the results shown in Table II, we can clearly see that the CUDA

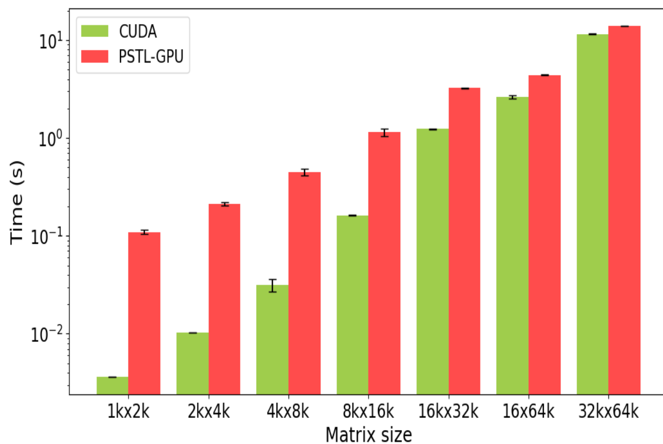

Figure 4. Results of Table II.

native version yields better results than PSTL (though only marginally). However, it must be noted that PSTL is still an experimental library and more performance improvements are to be expected in the future.

In particular, figure 4 highlights an overhead between CUDA and PSTL when the matrices are small and do not fill entirely the GPU. However, when the size of the matrices grows, the bottleneck becomes the computation of level 3 BLAS, which are used in both CUDA and PSTL code, and consequently, performance is similar.

Table I
FINAL TIMES OF OUR IMPLEMENTATIONS ON CPU.

Matrix size	OpenMP (s)	PSTL-CPU (s)
1k×2k	0.4947 ± 0.0492	0.0165 ± 0.0071
2k×4k	1.1702 ± 0.6588	0.0524 ± 0.0043
4k×8k	5.2870 ± 0.0894	0.2767 ± 0.0132
8k×16k	53.4493 ± 0.9770	1.9395 ± 0.0565
16k×32k	1022.0564 ± 10.6253	14.6499 ± 0.0659

Table II
FINAL TIMES OF OUR IMPLEMENTATIONS ON GPU.

Matrix size	CUDA (s)	PSTL-GPU (s)
1k×2k	0.0036 ± 0.0	0.1095 ± 0.0052
2k×4k	0.0102 ± 0.0	0.2115 ± 0.0074
4k×8k	0.0314 ± 0.0046	0.452 ± 0.0339
8k×16k	0.1627 ± 0.0017	1.1491 ± 0.0988
16k×32k	1.2372 ± 0.01	3.2508 ± 0.0387
16k×64k	2.6471 ± 0.0934	4.4214 ± 0.0375
32k×64k	11.5167 ± 0.0934	13.9629 ± 0.1048

On the other hand, it is surprising that through PSTL we get competitive performances with CUDA native implementation. This is very important from a programming point of view. In fact, given a sequential code, the CUDA implementation requires a complete restructuring of the code, and most of the time is not trivial to achieve. With PSTL the code base needed only marginal adjustments and reached competitive performances. Of course, the CUDA version performs better, but it must be noted that PSTL implementation is way more portable and flexible.

To conclude, we stopped at dimension 32k×64k since we reached the full GPU capacity with the biggest matrices (40GB). We see this result as a possible step towards a unified, user-friendly, and flexible language for optimizing the code on GPUs for Linear Algebra operations, providing a valid and feasible alternative to CUDA.

VI. CONCLUSIONS

So far we have discussed Karmarkar's Interior-point method and its parallelization through modern frameworks. In particular, we presented results that leveraged OpenMP, CUDA, and PSTL, enabling GPU acceleration within our implementation. Through the OpenMP implementation (every operation was developed from scratch), we were able to assess the difficulty in parallelizing this family of algorithms in an efficient and scalable way. To produce valuable results, the use of BLAS is needed and GPU acceleration can definitively boost the performance. In particular, leveraging PSTL we were able to develop a code base that could run both on multi-core and on GPU, with minimal effort with respect to the sequential version. On the other hand, through CUDA and the use of BLAS, it is possible to improve consistently the performances, reaching the best overall results.

PSTL is a library in development, so further improvements are expected in the future. Nevertheless, the possibility of having GPU acceleration with the same code base as the sequential version paves the way to a unified and portable programming language, and it needs to be considered as an important step towards more friendly solutions with respect to the development of code with CUDA, that most of the time hinges the GPU-acceleration for non-expert programmers [2].

VII. FUTURE WORKS

This work might be considered a good starting point for many possible future research directions. It provides a valid open-source code base to be improved with more Interior-point algorithms. In addition, Section IV-A has shown the limitations of leveraging the parallelization of this algorithm only through single operations optimizations. From an algorithmic point of view, this work represents a benchmark in optimizing Karmarkar's Interior-point method, and we hope to develop a parallel version of the whole algorithm, enabling more comparisons and further alternatives to compute-intensive parallelization models.

ACKNOWLEDGMENT

This work has been supported by ICSC – Centro Nazionale di Ricerca in High Performance Computing, BigData and

Quantum Computing, funded by European Union – NextGenerationEU.

REFERENCES

[1] Ilan Adler, Mauricio GC Resende, Geraldo Veiga, and Narendra Karmarkar. An implementation of karmarkar's algorithm for linear programming. *Mathematical programming*, 44:297–335, 1989.

[2] Marco Aldinucci, Valentina Cesare, Iacopo Colonnelli, Alberto Riccardo Martinelli, Gianluca Mittone, Barbara Cantalupo, Carlo Cavazzoni, and Maurizio Drocco. Practical parallelization of scientific applications with openmp, openacc and mpi. *Journal of parallel and distributed computing*, 157:13–29, 2021.

[3] Marco Aldinucci, Sergio Rabellino, Marco Pironti, Filippo Spiga, Paolo Viviani, Maurizio Drocco, Marco Guerzoni, Guido Boella, Marco Mellia, Paolo Margara, Idillio Drago, Roberto Marturano, Guido Marchetto, Elio Piccolo, Stefano Bagnasco, Stefano Lusso, Sara Vallero, Giuseppe Attardi, Alex Barchiesi, Alberto Colla, and Fulvio Galeazzi. HPC4AI, an AI-on-demand federated platform endeavour. In *ACM Computing Frontiers*, Ischia, Italy, May 2018.

[4] 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.

[5] Earl R Barnes. A variation on karmarkar's algorithm for solving linear programming problems. *Mathematical programming*, 36:174–182, 1986.

[6] Jakob Bieling, Patrick Peschlow, and Peter Martini. An efficient gpu implementation of the revised simplex method. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[7] Immanuel M Bomze, Vladimir F Demyanov, Roger Fletcher, Tamás Terlaky, Imre Pólik, and Tamás Terlaky. Interior point methods for nonlinear optimization. *Nonlinear Optimization: Lectures given at the CIME Summer School held in Cetraro, Italy, July 1-7, 2007*, pages 215–276, 2010.

[8] Valentina Cesare, Iacopo Colonnelli, and Marco Aldinucci. Practical parallelization of scientific applications. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 376–384. IEEE, 2020.

[9] George Bernard Dantzig. *Alternate Algorithm for the Revised Simplex Method: Using a Product Form for the Inverse*. Rand, 1953.

[10] Dick den Hertog and Cees Roos. A survey of search directions in interior point methods for linear programming. *Mathematical Programming*, 52:481–509, 1991.

[11] Jonathan Eckstein, İ İlkay Boduroğlu, Lazaros C Polymenakos, and Donald Goldfarb. Data-parallel implementations of dense simplex methods on the connection machine cm-2. *ORSA Journal on Computing*, 7(4):402–416, 1995.

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

[13] Robert M Freund and Shinji Mizuno. *Interior point methods: current status and future directions*. Springer, 2000.

[14] Nicolai Fog Gade-Nielsen. Interior point methods on gpu with application to model predictive control. 2014.

[15] David M Gay. A variant of karmarkar's linear programming algorithm for problems in standard form. *Mathematical Programming*, 37(1):81–90, 1987.

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

[17] Mark Hoemmen, DS Hollman, and Christian Trott. P1674r1: Evolving a standard c++ linear algebra library from the blas. 2022.

[18] Jin Hyuk Jung and DIANNE P O'Leary. Implementing an interior point method for linear programs on a cpu-gpu system. *Electronic Transactions on Numerical Analysis*, 28(174-189):37, 2008.

[19] 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.

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

[21] Victor Klee and George J Minty. How good is the simplex algorithm. *Inequalities*, 3(3):159–175, 1972.

[22] Mohamed Esseghir Lalami, Didier El-Baz, and Vincent Boyer. Multi gpu implementation of the simplex algorithm. In *2011 IEEE International Conference on High Performance Computing and Communications*, pages 179–186. IEEE, 2011.

[23] Zhang Liu and Lieven Vandenberghe. Interior-point method for nuclear norm approximation with application to system identification. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1235–1256, 2010.

[24] Marco Maggioni. *Sparse convex optimization on GPUs*. PhD thesis, University of Illinois at Chicago, 2016.

[25] Basilis Mamalis and Marios Perlitis. A hybrid parallelization scheme for standard simplex method based on cpu/gpu collaboration. In *Proceedings of the 20th Pan-Hellenic Conference on Informatics*, pages 1–6, 2016.

[26] Xavier Meyer, Paul Albuquerque, and Bastien Chopard. A multi-gpu implementation and performance model for the standard simplex method. In *Proceedings of the 1st International Symposium and 10th Balkan Conference on Operational Research (BALCOR 2011), 22-25 September 2011, Thessaloniki, Greece*. 22-25 September 2011, 2011.

[27] K Ponnambalam, A Vannelli, and TE Unny. An application of karmarkar's interior-point linear programming algorithm for multi-reservoir operations optimization. *Stochastic Hydrology and Hydraulics*, 3:17–29, 1989.

[28] Florian A Potra and Stephen J Wright. Interior-point methods. *Journal of computational and applied mathematics*, 124(1-2):281–302, 2000.

[29] Victor H Quintana, Geraldo L Torres, and Jose Medina-Palomo. Interior-point methods and their applications to power systems: a classification of publications and software codes. *IEEE Transactions on power systems*, 15(1):170–176, 2000.

[30] Christopher V Rao, Stephen J Wright, and James B Rawlings. Application of interior-point methods to model predictive control. *Journal of optimization theory and applications*, 99:723–757, 1998.

[31] Cornelis Roos, Tamás Terlaky, and J-Ph Vial. Interior point methods for linear optimization. 2005.

[32] Usman Ali Shah, Suhail Yousaf, Iftikhar Ahmad, Safi Ur Rehman, and Muhammad Ovais Ahmad. Accelerating revised simplex method using gpu-based basis update. *IEEE Access*, 8:52121–52138, 2020.

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

[34] Johannes GG van de Vorst. A parallel implementation of the karmarkar algorithm using a parallel linear algebra library. In *Conference Organized by Koninklijke/Shell-Laboratory, Amsterdam*, pages 127–135. Springer, 1988.

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