A GPU-Accelerated Interior Point Method with Applications in Radiation Therapy Optimization

Felix Liu[1][0000−0001−6865−9379], Albin Fredriksson[1], and Stefano Markidis[2]

[1] RaySearch Laboratories, Stockholm, Sweden
felix.liu@raysearchlabs.com
[2] KTH Royal Institute of Technology, Stockholm, Sweden

Abstract. Optimization plays a central role in modern radiation therapy, where it is used to determine optimal treatment machine parameters in order to deliver precise doses adapted to each patient case. In general, solving the optimization problems that arise can present a computational bottleneck in the treatment planning process, as they can be large in terms of both variables and constraints. As high precision is often sought, second-order optimization algorithms, such as sequential quadratic programming (SQP) and/or interior point methods (IPM) are commonly used. Existing implementations of these algorithms often use direct linear solvers internally, and are typically intended to run on CPUs. Utilizing iterative linear solvers instead is an active research topic in the optimization community, and one which carries the potential to enable efficient GPU acceleration for these types of optimization problems. Numerical stability issues make this a difficult problem for optimization solvers targeting problems from a wide range of application areas, however. In this paper, we develop and implement a GPU-accelerated interior point method for optimization problems from radiation therapy using iterative linear algebra. We utilize a so called doubly augmented formulation of the Karush-Kuhn-Tucker linear systems, together with a Jacobi-preconditioned conjugate gradient solver, which is able to find sufficiently accurate search directions while running on GPU. By evaluating our solver on real optimization problems from a commercial treatment planning system for radiation therapy, we show that our method can accelerate the aggregated time-to-solution by 1.4 and 4.4 times, respectively, for two patient cases.

1 Introduction

Optimization problems arise in a number of applications areas, including machine learning [3], operations research [23], radiation therapy planning [5,31] and many more. In many applications, the problems are large and challenging in terms of the number of variables and constraints, and the computational performance of the optimization solver is key. The focus in this work will be on interior point methods (IPMs) [22] for optimization, which are well known for their polynomial time complexity and good practical performance.

The specific application we have in mind for this work is optimization of radiation therapy treatment plans. In treatment planning, optimization is used

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11

to find control parameters for the treatment machine to deliver a precise dose that is concentrated to the tumor volume, thus achieving the desired killing of tumor cells while sparing surrounding healthy tissue as much as possible. The computations required in this process can be time-consuming, which makes computational speed a crucial factor. GPU computing is already widely used in radiation treatment planning [13], such as for dose calculation [8,17] and various image processing workloads [12]. With the advent of deep neural networks and machine learning, GPUs have also found uses in neural network based automatic segmentation algorithms [25].

One part of the computational workflow that has not yet benefited from GPU acceleration is the optimization algorithm itself. There may be many reasons for this, not least that other computational components, such as dose calculation [8], previously dwarfed the time required for optimization, a situation that is inevitably changing as large performance gains are realized in those areas. Another reason may be algorithmic in nature, in that current algorithms used for precise optimization may be inherently challenging to parallelize to the degree required to use GPUs efficiently.

An active research topic in the literature on IPMs is utilizing iterative linear solvers (e.g. Krylov subspace methods) as the method for solving linear systems internally. Traditionally, IPMs often rely on direct linear solvers, motivated in large part by numerical stability issues and inherent ill-conditioning of the linear systems involved. The move to iterative linear solvers, while challenging in terms of stability and preconditioning, may be crucial for the performance of IPMs on large-scale problems [10]. Another potential benefit of moving to iterative linear solvers, and one of the primary motivations for the method presented in this paper, is better suitability for massively parallel computing hardware such as GPUs,

In this paper, we present a GPU-accelerated IPM implementation for quadratic optimization problems, based on previous work on using Krylov subspace solvers to solve linear systems [16]. Furthermore, while our focus lies on interior point methods for quadratic optimization problems, we mainly consider the case where the quadratic problems are solved as part of a sequential quadratic programming (SQP) algorithm for general nonlinear optimization problems. We show that our solver can outperform existing, clinically used CPU-based solvers on real problems. To the best of our knowledge, this is the first of its kind in GPU accelerated IPMs for radiation therapy optimization.

2 Related Work

GPU accelerated optimization algorithms are already widely used in many contexts, especially for problems where first-order gradient-based algorithms (which do not require Hessian information) are used. Prominent examples include algorithms based on gradient descent for training deep neural networks and similar. For second-order methods with Hessian information such as IPMs and SQP, GPU accelerated solvers do not appear to be as widespread.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11

For linear programming, GPU accelerated IPMs have been studied previously by Smith et al. and Gade-Nielsen [27,9], which are both based on a matrix-free method proposed by Gondzio [11]. Notably, Gondzio's matrix-free method also uses a preconditioned conjugate gradient method, with regularization in the IPM itself as well as a custom preconditioner. GPU accelerated IPMs have also been studied for other types of optimization problems such as quadratic programming for training support vector machines [14], as well as more general nonlinear optimization [6]. The paper by Cao et al. [6] is similar to ours in that it uses a preconditioned conjugate gradient method with Jacobi preconditioning as well. However, they consider mainly equality constrained optimization problems and use a different formulation of the Karush-Kuhn-Tucker (KKT) system.

GPU acceleration for IPMs using direct linear solvers has also been studied previously, see [19], where the KKT-system is condensed into a dense form, which is more amenable to GPU accelerated factorization. In [30], a refactorization approach is considered, where pivots from previous factorizations are reused, since the sparsity pattern of matrices often remains the same between IPM iterations. Approaches combining inexact factorization and iterative refinement have also been considered [29], as well as hybrid methods combining factorization and iterative methods to solve KKT systems [24], with promising performance results demonstrated on optimal power flow problems, another application area where large-scale optimization problems are frequently encountered. An example of a software package for IPM with support for GPU acceleration is HiOp [20], which has also been used for optimal power flow problems [21].

For first-order optimization methods for quadratic optimization problems, GPU acceleration has been explored in for example the alternating direction method of multipliers (ADMM) based solver OSQP [28]. The GPU porting of OSQP is described in [26]. As a general rule, first-order methods trade achievable accuracy in favor of computational speed, which may be a very worthwhile trade-off for many applications, but may not be the most suited for radiation therapy where a high degree of accuracy is sought.

3 Background

The optimization algorithm used in this work is based on the method described in [16]. Our contribution in this work is to port the algorithm to GPU accelerators, address related challenges in performance optimization, and evaluate the performance compared to existing solvers on a set of realistic problems, in order to evaluate the performance compared to state-of-the-art. We give a brief overview of the optimization method used for completeness, but refer to [16] for more details.

3.1 Interior Point Methods

IPMs are commonly used for many types of constrained continuous optimization problems, including linear, quadratic, nonlinear and semidefinite programming.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11

Our interest in this paper is in IPMs for quadratic programming (i.e. optimization problems with quadratic objective function and linear constraints). Generally those problems are of the form

$$\begin{aligned} \text{min.} \quad & \frac{1}{2}x^T Hx + p^T x \\ \text{s.t.} \quad & l \leq Ax \leq u, \end{aligned} \tag{1}$$

where H is the $n \times n$ Hessian of the objective function, $p \in \mathbb{R}^n$ are linear coefficients of the objective function and A is an $m \times n$ matrix with coefficients for the linear inequality constraints. A common trick in optimization solvers is to introduce slack variables s_l, s_u for inequality constraints, thereby transforming them into equality constraints and a simple positivity constraint for the slack variables instead. The positivity constraints for the slack variables are handled by replacing them with a logarithmic barrier term in the objective, yielding a problem of the form

$$\begin{aligned} \text{min.} \quad & \frac{1}{2}x^T Hx + p^T x - -\mu \sum_i \log((s_l)_i) - \mu \sum_i \log((s_u)_i) \\ \text{s.t.} \quad & Ax - s_l - l = 0 \\ & -Ax - s_u + u = 0. \end{aligned} \tag{2}$$

With $s_l, s_u \geq 0$ handled implicitly. The intuition is that the logarithmic terms in the objective tend towards infinity as the boundary of the feasible region is approached from within, or in this case, when s_l, s_u become close to zero. μ is known as the *barrier parameter*, and its value can be chosen by the solver. IPMs proceed by solving the barrier problem while successively decreasing the value of the barrier parameter μ towards 0.

It can be shown that there exists *Lagrange multipliers* λ such that solutions x to the barrier problem (2) satisfy the following system of equations:

$$\begin{aligned} r_H &\coloneqq Hx + p - A^T\lambda_l + A^T\lambda_u = 0 \\ r_l &\coloneqq Ax - s_l - l = 0 \\ r_u &\coloneqq -Ax + s_u + u = 0 \\ r_{c_1} &\coloneqq (\lambda_l)_i(s_l)_i - \mu = 0, \quad i \in \{1, ..., m_l\}, \\ r_{c_2} &\coloneqq (\lambda_u)_i(s_u)_i - \mu = 0, \quad i \in \{1, ..., m_u\}, \end{aligned} \tag{3}$$

where λ_l denotes the multipliers for the lower bounds and λ_u for the upper bounds. The slack variables s are subscripted in the same way.

A popular approach is a so called primal-dual [22] approach, which is based on solving the system of equations (3) directly using Newton's method. Newton's method applied to (3) gives a linear system to solve of the form

$$\begin{pmatrix} H & -A^T & A^T & & \\ A & & & -I & \\ -A & & & & -I \\ & S_l & & \Lambda_l & \\ & & S_u & & \Lambda_u \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta\lambda_l \\ \Delta\lambda_u \\ \Delta s_l \\ \Delta s_u \end{pmatrix} = - \begin{pmatrix} r_H \\ r_l \\ r_u \\ r_{c_1} \\ r_{c_2} \end{pmatrix}, \tag{4}$$

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11

where Λ, S are diagonal matrices with the Lagrange multipliers and slack variables on the diagonal, respectively, and e is an appropriately sized vector of ones. Newton's method does not take into account the implicit condition that the slack variables and Lagrange multipliers remain positive throughout. This is accounted for by some line search method instead, where the search direction is scaled by some step length α such that the slacks and multipliers remain positive.

3.2 Sequential Quadratic Programming

Sequential quadratic programming (SQP) [4] is an optimization algorithm for solving nonlinear optimization problems with constraints. The basic idea is to solve, in each SQP iteration, a quadratic subproblem consisting of a quadratic approximation of the objective function or Lagrangian and linear approximation of the constraints. To give a concrete example, consider a problem of the form:

$$\begin{aligned} \text{min.} \quad & f(x) \\ \text{subject to} \quad & g(x) \leq 0, \end{aligned} \tag{5}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function and $g : \mathbb{R}^n \to \mathbb{R}^m$ are the constraints. We assume both $f(x)$ and $g(x)$ to be three times continuously differentiable. We define the Lagrangian of the problem as

$$\mathcal{L}(x, \lambda) = f(x) - \lambda^T g(x). \tag{6}$$

In SQP, we find search directions to iteratively solve problem (5) from the quadratic sub-problem

$$\begin{aligned} \min_d \quad & d^T \nabla^2_{xx} \mathcal{L}(x, \lambda) d + d^T \nabla f(x) \\ \text{subject to} \quad & d^T \nabla g(x) + g(x) \leq 0, \end{aligned} \tag{7}$$

where d is the search direction for the current iteration. These QPs solved in an SQP solver will often be referred to as *QP subproblems* in the remainder of the paper. For many practical problems, the Hessian of the Lagrangian may be too expensive to compute exactly. In such cases, it is common to use quasi-Newton type approximations of the Hessian instead. This is the approach used in the treatment planning optimization problems considered later in this paper, where a Broydon-Fletcher-Goldfarb-Shanno (BFGS) [15] type quasi-Newton approximation for the Hessian is used. The solution of the QP subproblems is a major computational burden in SQP solvers. Thus, SQP solvers both rely on an efficient solver fo quadratic programs internally, and also provide a way to extend a solver for QPs to the nonlinear setting. In this paper, we use our GPU accelerated IPM solver for the QP-subproblems.

4 Implementation

Solving the system (4) is the computational core of our method. As is common in practical implementations, we reduce the size of the system through block-row

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11

Algorithm 1 Interior Point Method

for $i \leftarrow 1$ to N **do**
  Solve (9) using preconditioned conjugate gradient (PCG) (GPU)
  Assemble full search direction from solution to (9) (CPU)
  Compute maximum step length α_x, α_λ (CPU)
  $x \leftarrow x + \alpha_x \Delta x$ (CPU)
  $\lambda \leftarrow \lambda + \alpha_\lambda \Delta\lambda$ (CPU)
  $s \leftarrow s + \alpha_x \Delta s$ (CPU)
  Update diagonal D in KKT system (CPU / GPU)
  Compute residuals r (CPU)
  if $||r|| < \mu$ **then**
    if $\mu \leq \mu_{tol}$ **then**
      Return solution
    end if
    $\mu \leftarrow \mu/10$
  end if
end for

elimination for efficiency reasons. Furthermore, it is common that our optimization problems will include bound constraints on the variables (of the form $a \leq x \leq b$). In the more general formulation (1), these are handled implicitly in the linear constraints. For computational efficiency however, it is beneficial to separate the rows of the constraint matrix A corresponding to such bound constraints. The result of these reductions gives us a system to solve of the form

$$\begin{pmatrix} Q & -B^T \\ B & D \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta\lambda_A \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \end{pmatrix}, \tag{8}$$

where

$$Q = H + S_{l_x}^{-1}\Lambda_{l_x} + S_{u_x}^{-1}\Lambda_{u_x}, \quad B = \begin{pmatrix} A \\ -A \end{pmatrix},$$
$$D = \begin{pmatrix} \Lambda_{l_A}^{-1} S_{l_A} & \\ & \Lambda_{u_A}^{-1} S_{u_A} \end{pmatrix}, \quad \Delta\lambda_A = \begin{pmatrix} \Delta\lambda_{l_A} \\ \Delta\lambda_{u_A} \end{pmatrix}.$$

S denotes diagonal matrices with the slack variables on the diagonal, and Λ denotes diagonal matrices with the Lagrange multipliers on the diagonal. They are subscripted based on the type of constraint they correspond to: l_x and u_x for lower and upper bounds on the variables, respectively, and l_A and u_A for lower and upper bounds on the linear constraints, respectively. A more detailed derivation of the block reductions leading to the formulation above can be found in [16].

To symmetrize system (8), we consider a *doubly augmented* formulation due to Forsgren and Gill [7]

$$\begin{pmatrix} Q + 2B^T D^{-1} B & B^T \\ B & D \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta\lambda_A \end{pmatrix} = \begin{pmatrix} r_1 + 2B^T D^{-1} r_2 \\ r_2 \end{pmatrix}. \tag{9}$$

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11

A high-level algorithmic overview of our method is shown in Algorithm 1 (adapted from [16]). The doubly augmented matrix in (9) is positive definite when Q is, which enables the use of a conjugate gradient (CG) solver. Furthermore, the ill-conditioning of the system arises in part due to the poor scaling of the diagonal block D, making Jacobi preconditioning a natural choice, which has the advantage of being very cheap to apply.

The main part of the computation that we have ported to GPU is the solution of the doubly augmented linear system (9) on line 2, while the remainder of the algorithm is run on CPU. The data transfer required in each iteration is not large, as we keep the doubly augmented KKT system on the GPU throughout the optimization, only updating the diagonal D, and the diagonal term of the Hessian block block each iteration. More concretely, the data transfer between CPU and GPU in each iteration consists of:

- The residuals which form the basis of the RHS of (9)
- The solution $(\Delta x, \Delta\lambda_A)$ from the PCG solver
- The diagonal matrix D
- The diagonal terms $S_{l_x}^{-1}\Lambda_{l_x} + S_{u_x}^{-1}\Lambda_{u_x}$ of the Hessian block.

4.1 GPU Acceleration

The most time-consuming part in the optimization algorithm is the preconditioned CG solver used to solve the KKT system at each iteration, which makes it a natural target for GPU acceleration. There are essentially three components to this, computing the matrix-vector products with the KKT-matrix on the GPU, the Jacobi preconditioner, and then general performance considerations for CG on GPUs.

Doubly Augmented KKT Matrix-Vector Multiplications Multiplications with the doubly augmented matrix are relatively straightforward to implement efficiently on GPU, and we always work with the matrix in unassembled form by computing the products with different sub-blocks of the matrix separately. In the doubly augmented matrix, the Hessian H is stored on the GPU exclusively, as are the diagonal block D and the diagonal terms in Q. The constraint matrix B is stored on both CPU and GPU. Since the constraint matrix remains constant throughout the optimization, the copy to GPU is done when the problem is initialized and no further data transfer between CPU and GPU is needed for the constraint matrix in the solver. We store the sub-blocks of the B matrix in CSR format, and we also pre-compute and store their transposes. This enables us to use transpose-free SpMV kernels for all of our sparse matrix-vector products, which improves performance.

The C++ code for our solver is written to allow execution on both CPU and GPU for the solver. As an example of design choices to accommodate this, the class representing the doubly augmented matrix is templated on the dense vector type used. This is to allow both CPU and GPU execution. For the GPU accelerated case, we provide the template argument `CudaDenseVector`

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11

(representing a GPU dense vector implemented in CUDA) and for the CPU case, we use the corresponding CPU class `DenseVector`. The matrix-free approach enabled by the use of Krylov solvers is also seen in the code, as the multiplication is done by accumulating results from each separate block and term of the matrix separately. No fully assembled representation of the entire doubly augmented matrix is ever required, we only store the components of the full matrix (e.g. Hessian H and constraint matrix). The code makes extensive use of C++20's `std::span`, to extract views over contiguous blocks of arrays. This is especially useful in our application, as the doubly augmented matrix is naturally divided into blocks, and performing matrix-vector multiplications (or other operations) using it naturally means needing to perform calculations on sub-sections of the vector. `std::span` enables us to perform operations on chunks of the vector easily, without the use of costly additional copies. Being essentially a thin wrapper around a raw pointer and size, `std::span` is also usable for both CPU and GPU arrays, which we use to simplify code and interfaces for code meant for both CPU and GPU calculations.

Computing the Jacobi Preconditioner For the Jacobi preconditioner, one needs to compute the diagonal elements of the KKT-matrix. For the bottom right block, consisting of the D matrix, this is trivial, since the matrix is already diagonal, so one can simply extract the elements directly. The top left block is different, since it is not explicitly formed in the solver.

The computation of the diagonal of the Hessian is relatively straightforward, and this calculation only needs to be performed once, since the Hessian does not change throughout the optimization. For the remainder of the optimization, we cache the pre-computed diagonal of the Hessian and provide that directly whenever required.

For the diagonal of the $2A^T D^{-1} A$ term in the top left block, the situation is more complicated. First, this term changes each IPM iteration, since the diagonal matrix D does, and secondly, computing the diagonal in parallel on the GPU is less straightforward. We address this by keeping an extra copy of the constraint matrix on the CPU, which is used to compute $\text{diag}(A^T D^{-1} A)$. This presents some overhead in data transfer between CPU and GPU, but since the diagonal only needs to be recomputed once per IPM iteration (of which there are typically fewer than 100), this trade-off was found to be acceptable.

5 Experimental Setup

In the following we describe the experimental setup, in terms of the hardware and software used as well as the source of the test problems.

5.1 Hardware and Software Environment

The following test systems were used to conduct the performance evaluations in this work.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11

- **Bluedog** is a local workstation equipped with an AMD Ryzen 9 7900X CPU, and 64 GB of DDR5 RAM @ 5200 MT/s. The GPU is an Nvidia GeForce RTX 4080 with 16 GB of GDDR6X RAM.
- **RS_WKS** is a local Windows workstation with RayStation version 2024A installed. The system is equipped with an Intel Core i9-7940x CPU and 64 GBs of DDR4 RAM @ 2666 MT/s.
- **NJ** is a local server at KTH equipped with an AMD EPYC 7302p 16 core CPU. The GPU is an Nvidia A100 with 40GB of HBM2 memory.

We evaluate the performance of our method in multiple ways. We measure the impact of GPU acceleration by evaluating the performance improvement compared to the CPU version of the solver. The CPU version of the solver is capable of parallel execution through the use of multithreaded BLAS. We use OpenBLAS 0.3.24 for the CPU version of the solver, with the default value used for the number of threads. Our SpMV implementation on the CPU is parallelized using OpenMP. These computational kernels (BLAS and SpMV) should occupy the majority of computational time in the solver. The GPU version of the solver utilizes the cuBLAS and cuSPARSE library heavily for dense and sparse linear algebra kernels. We also analyze the performance of our solver on a range of different GPUs, to see how the performance varies across GPUs for our problem case. Finally, to give an idea of how the GPU accelerated solver may improve solution times for radiation treatment planning in practice, we compare our optimization solver to the one implemented in RayStation, which is a commercial treatment planning system used in clinics around the world. RayStation's QP solver is capable of multi-threaded execution in many cases, but the degree of parallelization varies substantially between cases.

5.2 Test Problems from Radiation Treatment Planning

Problem	Vars.	Lin. cons.	Bound cons.
Proton H&N	77373	0	77373
Proton H&N (after spot filtering)	33531	0	33531
VMAT H&N	13425	68618	13425

Table 1: Dimensions of the optimization problems used in the performance analysis in terms of number of variables, linear constraints and bound constraints. The proton case is shown before and after spot filtering, which occurs after 100 SQP iterations.

The optimization problems we use are quadratic programming subproblems exported directly from the RayStation SQP solver. These are the problems the SQP method solves to find search directions in each iteration, and represent the main computational burden. We consider two cases, one for cancer in the head and neck region treated using protons, and one head and neck case treated

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11

using photons with a treatment technique known as Volumetric Modulated Arc Therapy (VMAT) [18]. For the proton case, the SQP solver performs so-called spot filtering after 100 SQP iterations, which reduces the size of the optimization problem by eliminating variables that are close to zero. Spots, in this case, are intensities of the proton beam in discrete points along the scanning path which can be controlled to achieve the desired dose. Dimensions of the QP subproblems for our test problems are shown in Table 1. The total number of SQP iterations used were 200 and 33, for the proton and photon VMAT case, respectively. This also corresponds to the number of QP subproblems for the different cases.

We expect the QP subproblems to become more expensive to solve in later SQP iterations due to the quasi-Newton Hessian becoming larger for each iteration, since each iteration adds two terms to the BFGS Hessian approximation, which makes the (dense) matrix of update vectors two columns larger. Spot filtering resets the quasi-Newton Hessian approximation, which is another important factor in reducing computational cost after filtering.

5.3 Matrix dimensions and Computational Cost

The size of the matrices in each QP subproblem varies slightly from iteration to iteration. The largest matrix components of the doubly augmented KKT linear system are the quasi-Newton Hessian $H = H_0 + UWU^T$, and the linear constraint matrix B. The remaining blocks and terms are diagonal matrices, and thus comparatively cheap to perform calculations with. In the quasi-Newton Hessian, H_0 is a square diagonal matrix (corresponding to the initial guess for the BFGS Hessian) with the dimension equal to the number of variables in the optimization problem, which can be found in Table 1. W is a diagonal $k \times k$ matrix, where k is the twice the SQP iteration that the current QP-subproblem corresponds to (since two new rank one updates are added to the BFGS Hessian each SQP iteration). Finally, U is a dense $n \times k$ matrix with the BFGS update vectors as columns.

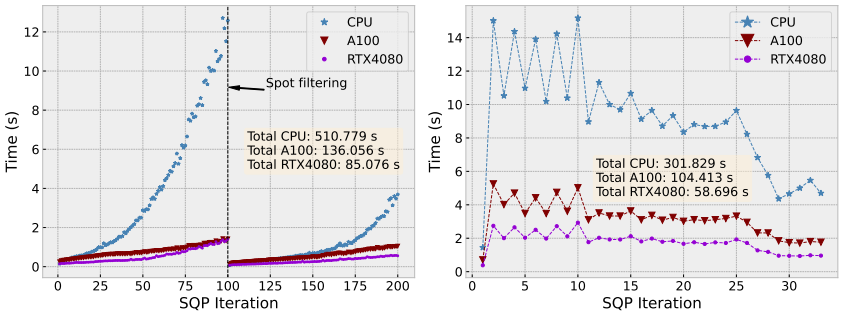
6 Results

This section presents our results from benchmarking the CPU and GPU performance, as well as comparison with the RayStation solver on a set of realistic problems.

6.1 GPU and CPU comparison

We begin by measuring the execution time of our solver on different GPUs and on the CPU to see the impact of GPU acceleration. Figure 1 shows some results from this comparison. We see that the GPU acceleration brings significant performance benefits to our solver, as we would expect, with an approximately 6× speedup of the total execution time when comparing the CPU baseline with the RTX4080 results for the proton head and neck case and approximately 5.1× for the VMAT

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11



(a) Head and Neck proton arc problem. Spot filtering after 100 SQP iterations, where some variables that are close to zero are pruned from the problem.

(b) VMAT Head and Neck problem

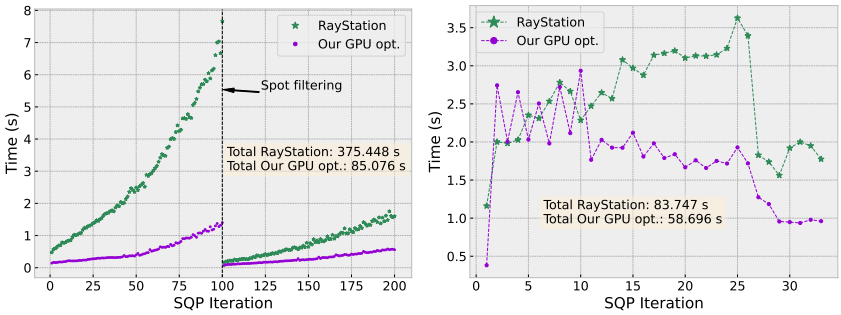
Fig. 1: Comparison of performance for our solver on different GPUs and on the CPU. The CPU and RTX4080 benchmarks were run on Bluedog. The A100 benchmark was performed on NJ.

head and neck case. The execution time of early SQP iteration for the VMAT case shows relatively large oscillations, which seem to be due to numerical differences between the problems causing slower convergence speed (i.e. more IPM or and/or Krylov iterations). The reason why the QP-subproblems alternate between being easier and more difficult is not fully understood. Interestingly, the solver performs better on the RTX4080 system (Bluedog) than on the A100 system (NJ), despite the peak throughput in both memory bandwidth and floating point operations being higher for the A100. Further profiling and analysis suggests that relatively small kernel sizes and larger latency for the A100 may be a large contributing factor to the performance deficit. Merging multiple smaller computational kernels into larger ones (and consequently relying less on cuBLAS for smaller kernels) may alleviate this issue [1]. Further performance engineering in that direction is left for future work.

6.2 Performance Comparison on Realistic Cases

Figure 2 shows the solution time comparison between our solver running on **Bluedog** (with the Nvidia RTX 4080 GPU) and RayStation running on **RS_WKS**. The times shown are solution times for QP subproblems in the SQP solver. The runs using our GPU accelerated optimizer are performed on exported QP subproblems from RayStation. The timing for the RayStation solver is isolated to only include the solution of the QP subproblems, in order to make a fair comparison. Other work in each SQP iteration is not included in the RayStation solution times. In total, we see that our optimization solver outperforms RayStation's optimizer by 4.4× for the proton problem and roughly 1.4× for the photon VMAT problem. For the proton case, the dashed vertical line in the plot

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11



(a) Head and Neck proton arc problem. Spot filtering occurs after 100 SQP iterations, marked with a dashed vertical line, where some variables that are close to zero are pruned from the problem.

(b) VMAT Head and Neck problem.

Fig. 2: Comparisons of solution times for QP subproblems between RayStation's QP solver and our GPU accelerated solver. Only solution times for QP subproblems is measured, and does not include updating the quasi-Newton Hessian or other operations. Total solution time (on all subproblems) for the RayStation optimizer and our solver is shown in the text box.

shows the point where *spot filtering* occurs, which is an intermediate step in the SQP optimization where variables that are close to zero are pruned from the problem. The reason, we believe, for the relatively larger improvement for the proton case is twofold. First, the proton case is a bound constrained problem, and tended to require fewer CG iteration in each IPM iteration to converge. Secondly, the main computations in the VMAT case are sparse matrix-vector products, which may relatively speaking benefit less from GPU porting compared to the dense matrix-vector operations for the proton case. While a completely fair comparison between a CPU and GPU implementation is challenging — and is made more complicated by the fact that the algorithms used also differ (direct versus iterative linear solvers) — we emphasize that the RayStation optimization algorithm is originally developed for CPU only, and may not benefit from direct porting to GPU at all. The comparison above is rather intended to give an idea of the speedup obtainable in practice by shifting to the GPU based optimization solver instead.

7 Conclusions and Future Work

In this paper, we presented our GPU accelerated interior point method implementation which is tailored for solving optimization problems from treatment planning for radiation therapy. The move to GPU was enabled by a shift from using direct linear solvers to iterative linear solvers internally, based on the method proposed in [16]. Much research in the optimization literature has been

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11

conducted on the use of iterative linear solvers for IPMs, however, most optimization packages still use direct linear solvers. We showed in this paper that a GPU accelerated implementation based on iterative linear algebra can outperform existing approaches on real problems from radiation therapy optimization.

Some aspects of the IPM implementation used in this work are still rather crude and could be further investigated and improved. An example is the rather conservative method to update the barrier parameter μ, where we decrease its value only when the current barrier subproblem is solved somewhat accurately. Further work on improving the selection of barrier parameter values, which should likely take into account the inexactness of the search direction due to the iterative linear solver [2], and similar may push the performance even further. Additionally, our comparison of performance between GPU models suggests that there is still room for further performance optimizations in the implementation. For example, kernel launch overhead for small computational kernels could be addressed by merging smaller computational kernels into larger ones, thus relying less on cuBLAS for computations. However, it is encouraging that even with a relatively simple IPM implementation, the use of iterative linear solvers and GPUs is able to improve the overall time-to-solution compared to the clinically used solver in RayStation.

Overall, our GPU accelerated solver was able to improve total optimization times by 1.4× and 4.4× respectively, when compared to an optimization solver from a clinically used treatment planning system on realistic problems. Furthermore, enabling the use of more powerful computational hardware may provide future proofing in a field where demands for computational efficiency are ever increasing.

References

1. Anzt, H., Tomov, S., Luszczek, P., Sawyer, W., Dongarra, J.: Acceleration of GPU-based Krylov solvers via data transfer reduction. The International Journal of High Performance Computing Applications **29**(3), 366–383 (2015)
2. Bellavia, S.: Inexact interior-point method. Journal of Optimization Theory and Applications **96**, 109–121 (1998)
3. Bennett, K.P., Parrado-Hernández, E.: The interplay of optimization and machine learning research. The Journal of Machine Learning Research **7**, 1265–1281 (2006)
4. Boggs, P.T., Tolle, J.W.: Sequential quadratic programming. Acta Numerica **4**, 1–51 (1995)
5. Bortfeld, T., Thieke, C.: Optimization of treatment plans, inverse planning. In: New Technologies in Radiation Oncology, pp. 207–220. Springer (2006)
6. Cao, Y., Seth, A., Laird, C.D.: An augmented Lagrangian interior-point approach for large-scale NLP problems on graphics processing units. Computers & Chemical Engineering **85**, 76–83 (2016)
7. Forsgren, A., Gill, P.E., Griffin, J.D.: Iterative solution of augmented systems arising in interior methods. SIAM Journal on Optimization **18**(2), 666–690 (2007)
8. Fracchiolla, F., Engwall, E., Janson, M., Tamm, F., Lorentini, S., Fellin, F., Bertolini, M., Algranati, C., Righetto, R., Farace, P., et al.: Clinical validation of a GPU-based Monte Carlo dose engine of a commercial treatment planning system for pencil beam scanning proton therapy. Physica Medica **88**, 226–234 (2021)

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11

9. Gade-Nielsen, N.F.: Interior point methods on GPU with application to model predictive control. Ph.D. thesis, Technical University of Denmark (2014)
10. Gondzio, J.: Interior point methods 25 years later. European Journal of Operational Research **218**(3), 587–601 (2012)
11. Gondzio, J.: Matrix-free interior point method. Computational Optimization and Applications **51**, 457–480 (2012)
12. Gu, X., Pan, H., Liang, Y., Castillo, R., Yang, D., Choi, D., Castillo, E., Majumdar, A., Guerrero, T., Jiang, S.B.: Implementation and evaluation of various demons deformable image registration algorithms on a GPU. Physics in Medicine & Biology **55**(1), 207 (2009)
13. Jia, X., Ziegenhein, P., Jiang, S.B.: GPU-based high-performance computing for radiation therapy. Physics in Medicine & Biology **59**(4), R151 (2014)
14. Li, T., Li, H., Liu, X., Zhang, S., Wang, K., Yang, Y.: GPU acceleration of interior point methods in large scale SVM training. In: 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications. pp. 863–870. IEEE (2013)
15. Liu, D.C., Nocedal, J.: On the limited memory BFGS method for large scale optimization. Mathematical Programming **45**(1), 503–528 (1989)
16. Liu, F., Fredriksson, A., Markidis, S.: Krylov solvers for interior point methods with applications in radiation therapy and support vector machines. In: Computational Science – ICCS 2024. pp. 73–77. Springer (2024)
17. Liu, F., Jansson, N., Podobas, A., Fredriksson, A., Markidis, S.: Accelerating radiation therapy dose calculation with Nvidia GPUs. In: 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 449–458. IEEE (2021)
18. Otto, K.: Volumetric modulated arc therapy: IMRT in a single gantry arc. Medical Physics **35**(1), 310–317 (2008)
19. Pacaud, F., Shin, S., Schanen, M., Maldonado, D.A., Anitescu, M.: Accelerating condensed interior-point methods on SIMD/GPU architectures. Journal of Optimization Theory and Applications pp. 1–20 (2023)
20. Petra, C.G.: A memory-distributed quasi-Newton solver for nonlinear programming problems with a small number of general constraints. Journal of Parallel and Distributed Computing **133**, 337–348 (2019)
21. Petra, C.G., Aravena, I.: Solving realistic security-constrained optimal power flow problems. arXiv preprint arXiv:2110.01669 (2021)
22. Potra, F.A., Wright, S.J.: Interior-point methods. Journal of Computational and Applied Mathematics **124**(1-2), 281–302 (2000)
23. Rais, A., Viana, A.: Operations research in healthcare: a survey. International Transactions in Operational Research **18**(1), 1–31 (2011)
24. Regev, S., Chiang, N.Y., Darve, E., Petra, C.G., Saunders, M.A., Świrydowicz, K., Peleš, S.: HyKKT: a hybrid direct-iterative method for solving KKT linear systems. Optimization Methods and Software **38**(2), 332–355 (2023)
25. Samarasinghe, G., Jameson, M., Vinod, S., Field, M., Dowling, J., Sowmya, A., Holloway, L.: Deep learning for segmentation in radiation therapy planning: a review. Journal of Medical Imaging and Radiation Oncology **65**(5), 578–595 (2021)
26. Schubiger, M., Banjac, G., Lygeros, J.: GPU acceleration of ADMM for large-scale quadratic programming. Journal of Parallel and Distributed Computing **144**, 55–67 (2020)
27. Smith, E., Gondzio, J., Hall, J.: GPU acceleration of the matrix-free interior point method. In: Parallel Processing and Applied Mathematics: 9th International

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11

Conference, PPAM 2011, Torun, Poland, September 11-14, 2011. Revised Selected Papers, Part I 9. pp. 681–689. Springer (2012)
28. Stellato, B., Banjac, G., Goulart, P., Bemporad, A., Boyd, S.: OSQP: An operator splitting solver for quadratic programs. Mathematical Programming Computation **12**(4), 637–672 (2020)
29. Świrydowicz, K., Koukpaizan, N., Alam, M., Regev, S., Saunders, M., Peleš, S.: Iterative methods in GPU-resident linear solvers for nonlinear constrained optimization. arXiv preprint arXiv:2401.13926 (2024)
30. Świrydowicz, K., Koukpaizan, N., Ribizel, T., Göbel, F., Abhyankar, S., Anzt, H., Peleš, S.: GPU-resident sparse direct linear solvers for alternating current optimal power flow analysis. International Journal of Electrical Power & Energy Systems **155**, 109517 (2024)
31. Wedenberg, M., Beltran, C., Mairani, A., Alber, M.: Advanced treatment planning. Medical Physics **45**(11), e1011–e1023 (2018)

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97629-2_11