---

**Article:**
Moreno, J.J. orcid.org/0000-0002-2194-2318, Puertas-Martín, S., Redondo, J.L. et al. (2 more authors) (2025) Where high-performance computing meets radiotherapy for enhanced intensity-modulated radiation therapy planning. Concurrency and Computation: Practice and Experience, 37 (12-14). e70133. ISSN 1532-0626

https://doi.org/10.1002/cpe.70133

---

*Concurrency and Computation: Practice and Experience*

WILEY

SPECIAL ISSUE PAPER

# Where High-Performance Computing Meets Radiotherapy for Enhanced Intensity-Modulated Radiation Therapy Planning

Juan José Moreno[1] | Savíns Puertas-Martín[1,2] | Juana L. Redondo[1] | Pilar M. Ortigosa[1] | Ester M. Garzón[1]

[1]Informatics Departament, CEINSA, University of Almería, Almería, Spain | [2]Information School, University of Sheffield, Sheffield, UK

**Correspondence:** Juan José Moreno (juanjomoreno@ual.es)

**Received:** 7 February 2025 | **Revised:** 30 April 2025 | **Accepted:** 13 May 2025

**Funding:** Ministerio de Ciencia, Innovación y Universidades, European Regional Development Fund.

**Keywords:** evolutionary optimization | high performance computing | IMRT | radiotherapy

## ABSTRACT

Intensity Modulated Radiotherapy (IMRT) employs radiation beams with varying angles and intensities to precisely target cancerous tissues while sparing healthy organs. Planning methods based on the generalized Equivalent Uniform Dose (gEUD) metric achieve excellent Planning Target Volume coverage. However, computing these plans requires extensive parameter adjustments and multiple model evaluations, making the process resource-intensive and time-consuming. This study aims to enhance the computational efficiency of radiotherapy plans by automating the adjustment of gEUD parameters, reducing solution times, and facilitating clinical integration. We introduced a novel approach that combines Gradient Descent algorithms with evolutionary optimization to explore the gEUD parameter space. This hybrid methodology generates radiation plans that meet clinical constraints. To address the high computational costs, we implemented parallelization and batching strategies, leveraging multicore servers to accelerate the optimization process and enable real-time clinical applications. Benchmarking was conducted on three multicore platforms with distinct micro-architectures, testing various batch sizes and thread configurations. Using a dataset of three Head and Neck IMRT patients treated with nine beams, our approach demonstrated substantial computational speed-ups. Results confirmed the ability of the method to consistently produce high-quality radiation therapy plans that meet clinical constraints. By effectively exploiting multicore servers, this approach overcomes the computational challenges of gEUD parameter tuning, enabling its integration into clinical practice. This advancement reduces planning times, supports medical physicists, and ultimately enhances patient care in radiotherapy.

## 1 | Introduction

External radiation therapy treatments are extremely important for addressing cancer treatments. In particular, Intensity Modulated Radiation Therapy (IMRT) is a highly effective cancer treatment technique that accurately delivers radiation to cancerous tissues while preserving the surrounding healthy organs. IMRT linear accelerators deliver radiation beams to patients from different angles with different intensities within a beam. To manage radiation dose deposition, each beam is decomposed into a regular grid of beamlets whose radiation intensity can be individually controlled. Therefore, every plan is defined by the specific intensities of all beamlets on all beams, called the fluence map. When preparing an RT (Radiation Therapy) plan, the goal is twofold.

On the one hand, plans are sought that ensure the minimal prescribed doses in tumorous cells (the target regions, commonly referred to as PTV (Planning Target Volume) and, on the other hand, reduce the radiation on highly sensitive organs (Organ(s) At Risk, OAR(s)). These two goals are contradictory and, as such, must be traded off, which means that effective and safe patient plans require significant effort from planners.

The design of effective plans is very complex because of the high dimensions of the fluence maps that define such plans and the difficulties of expressing different types of clinical prescriptions in optimization models.

Numerous optimization models with different statistical and biological criteria have been proposed in the literature to express different medical prescriptions [1, 2]. However, each model focuses on limited criteria, and the plans computed by the software require manual adjustment with all clinical criteria to get feasible clinical plans. Beyond the precision of the plans, the design time is crucial in radiotherapy planning, and high-performance computing (HPC) techniques have been used to develop planning software. Therefore, there are many works focused on exploitation of the parallelism involved in planning on several platforms, for example, multicore and vector units on the CPU [3], on GPUs [4, 5], on reconfigurable hardware architectures (FPGAs) [6], or on distributed clusters [7].

Planning techniques based on the generalized Equivalent Uniform Dose (gEUD) biological metric provide radiation plans with excellent PTV coverage. The gEUD concept enforces the prescribed radiation doses in PTVs and OARs by a penalty function that uses a linear-quadratic cell survival model. For the gEUD-based optimization model, optimal solutions can be efficiently computed by gradient methods. However, for every PTV region and every OAR, the planner should set several gEUD parameters. To account for the additional statistical constraints needed for viable plans, imposed by oncology doctors, manual tuning of gEUD parameters is required from medical physicists. This is a time-consuming and complex task that consumes a relevant time in the clinical workflow daily. In [8], we proposed an evolutionary optimization strategy that eliminates manual gEUD parameter tuning and replaces it with automatic parameter selection, using an evolutionary optimization method to guide the search for the optimal plan when a gradient algorithm evaluates candidate solutions. Our proposal, called PersEUD, can significantly reduce the effort required for manual parameter selection and provide a more effective way to obtain optimal plans.

In the PersEUD approach, on the one hand, the optimal solution of the gEUD model with a given set of parameters is solved by a gradient algorithm. On the other hand, the evolutionary model is harnessed to explore the gEUD parameter space in the quest for promising parameter sets. Therefore, collections of optimal solutions of the gEUD model are independently computed with a gradient method at each evolutionary iteration.

However, the huge computational cost of this proposal prevents its application in clinical practice. In this work, we aim to overcome this challenge and apply HPC techniques to efficiently compute viable plans on the most common computing platforms in the clinical context. Considering the large computational power of modern multicore servers, our goal is to obtain optimal utilization of multicore platforms to get a fast PersEUD implementation that allows us to use it in the clinical context.

In the course of the optimization computations, the most time-consuming operation is to calculate the values of the objective functions, that is, the optimal solution of the gEUD model achieved with the Gradient Descent (GD) method. The gEUD model is a function of the radiation doses deposited in the voxels (3D mesh of the irradiated part of the patient's body). Such doses are computed as the products of deposition matrices and beamlet intensity vectors. The deposition matrices are large and sparse; therefore, the dose computations consume the most computational resources in the IMRT planner. The result is a process dominated by products of the deposition matrix and fluence maps as dense vectors. The acceleration of our proposal is focused on the efficient management of these operations on modern multicore systems. Our proposal harnesses the new planning tool that allows batching the products of the deposition matrix and several fluence maps into one matrix product. With this scheme, the multicore architecture can efficiently accelerate the computation of objective functions. High-performance servers are commonly used to accelerate the computing involved in planning. To name a few representative examples, Karbalaee et al. [4] and Liu et al. [5] describe approaches to accelerate the simulation dose on GPUs and [7] on distributed clusters. Despite the strong potential of GPUs to accelerate this type of model, our initial approach to improving the performance of the gEUD model focuses on the efficient use of multicore servers, as these are widely available in hospital computational environments.

The rest of the article is organized as follows. Section 2 introduces the gEUD model, explains the evolutionary method used to fine-tune its parameters, and provides an in-depth overview of our approach to speed up the automated generation of plans. Section 3 provides a computational evaluation of our implementation to exploit multicore systems. The evaluation is based on the data of three patients on three servers with several multicore processors. We present and discuss the results obtained from our experiments and analyses. Finally, the Section 4 summarizes the findings and conclusions drawn from this study, as well as suggesting potential avenues for future research in this domain.

## 2 | Methods

In this section, we introduce the basic foundations of our work: The gEUD model and the evolutionary method to adjust its parameters are introduced in Section 2.1, and the details of our proposal to accelerate the automatic generation of plans are described in Section 2.2.

### 2.1 | Problem Definition

In this section, we will delve into the parameters that are crucial in the gEUD model and explore the evolutionary method utilized for their adjustment. To help the reader, Table 1 summarizes the notation used in the paper.

**TABLE 1** | Notation and naming conventions.

| Notation | Meaning |
|---|---|
| $x$ | Fluence map |
| $D$ | Deposition matrix (translates $x$ to doses in voxels) |
| $D_j$ | $j$th row of $D$ |
| $d(x) = Dx$ | Vector of doses deposited in voxels |
| $d_j(x) = D_j x$ | Dose deposited in voxel $j$ |
| $T = \{t\}$ | Set of indices of PTVs |
| $M_t$ | Set of voxels in $t$th PTV |
| $EUD_t^0$ | Prescribed uniform dose for $t$th PTV |
| $R = \{r\}$ | Set of indices of OARs |
| $M_r$ | Set of voxels in $r$-th OAR |
| $EUD_r^0$ | Maximal uniform dose for $r$th OAR |

### 2.1.1 | Parameters to Adjust in the gEUD Model

The process of planning radiotherapy for a patient can be seen as an optimization procedure, where the search space is determined by feasible fluence maps of the linear accelerator. They are represented by vectors of non-negative numbers $\mathbf{x}$, which define the radiation intensities of individual beamlets. The deposition matrix $\mathbf{D}$ translates fluencies $\mathbf{x}$ into doses deposited in voxels so that the doses in voxels are computed as a product $\mathbf{Dx}$. The goal of the planning is to calculate the fluence map $\mathbf{x}$ that deposits prescribed doses in PTVs and admissible doses in OARs with acceptable 3D dose distributions.

gEUD is a biology-motivated criterion to estimate radiation effects, based on the concept of a uniform radiation dose delivered to a patient organ, which causes the same effect as a non-uniform dose [9, 10]. In the model, PTVs and OARs are referred to as structures or Region Of Interest (ROI) with index $s$. In this way, the gEUD model evaluates the radiation effects in an ROI by the following function that aggregates these effects on all voxels belonging to the structure $s$:

$$gEUD_s(\mathbf{x}, a_s) = \left( \frac{1}{|S_s|} \sum_{j \in S_s} d_j(\mathbf{x})^{a_s} \right)^{\frac{1}{a_s}} \tag{1}$$

where $S_s$ is the set of voxels of structure $s$; $d_j(\mathbf{x}) = D_j\mathbf{x}$ is the radiation dose deposited in the voxel $j$ of structure $s$ by the fluence map $\mathbf{x}$, with $D_j$ the $j$-th row of $\mathbf{D}$; the parameter $a_s$ represents the radiation effect on the structure. The model for PTVs generally uses large negative values of this parameter. In contrast, OARs such as the spinal cord and rectum typically use large positive values, while other OARs that exhibit a large-volume effect (e.g., lungs and parotids) are assigned small positive values [10]. We denote by the indices $t$ and $r$ the structures PTV and OAR, respectively.

According to [10], clinically meaningful RT plans can be obtained by computing the maximum built over the gEUD criterion:

$$\mathbf{x}^\star = \underset{\mathbf{x}}{\operatorname{argmax}} F(\mathbf{x}, \varphi) \tag{2}$$

with

$$F(\mathbf{x}, \varphi) = \prod_{t \in T} \frac{1}{1 + \left( \frac{EUD_t^0}{gEUD_t(\mathbf{x}, a_t)} \right)^{n_t}} \cdot \prod_{r \in R} \frac{1}{1 + \left( \frac{gEUD_r(\mathbf{x}, a_r)}{EUD_r^0} \right)^{n_r}} \equiv \prod_{t \in T} f_t(\mathbf{x}, p_t) \cdot \prod_{r \in R} f_r(\mathbf{x}, p_r) \tag{3}$$

where $EUD_t^0$ is the prescribed dose for $t$-th PTV, $EUD_r^0$ is the maximum uniform dose at $r$th OAR; $n_r$ and $n_t$ express the importance of the prescriptions for the corresponding structure; $\varphi$ represents the set of parameters involved in the $F$ definition. In the model, every OAR with index $r$ (resp. PTV with index $t$) involves the set of parameters $p_r \equiv \{a_r, n_r, EUD_r^0\}$ (resp. $p_t \equiv \{a_t, n_t\}$). Hence, all the parameters used in the gEUD model can be concatenated into the array $\varphi = (p_0, \ldots, p_s, \ldots, p_n)$, where $n$ denotes the total number of ROIs. However, the dimension of $\varphi$ ranges between $2n$ and $3n$, since the concatenated subarrays $p_s$ have a size of 2 or 3, depending on whether they correspond to OARs or PTVs, respectively. Moreover, Equation (3) denotes the factors of $F(\mathbf{x}, \varphi)$ as $f_t(\mathbf{x}, p_t)$ and $f_r(\mathbf{x}, p_r)$ to emphasize that each factor is associated with a specific ROI and involves its corresponding set of parameters, $p_s$.

With these parameters fixed, the GD method computes $x^\star(\varphi)$ defined in Equation (2) [11]. In practice, the values of these parameters are initially fixed from the literature and then adjusted by trial and error, where the GD method is involved to obtain a clinically feasible plan. In the following, a brief explanation of the GD algorithm to optimize the gEUD model is included.

### 2.1.2 | Gradient Descent Algorithm to Optimize gEUD Model

The objective function $F(\mathbf{x}, \varphi)$ is non-linear and differentiable, so a GD method can be used to explore possible plans that maximize $F(\mathbf{x}, \varphi)$. To facilitate the computation of the derivatives, we can transform the optimization model in terms of $\ln F(\mathbf{x}, \varphi)$.

The gradient function to look for the arguments $\mathbf{x}$ that maximize $F(\mathbf{x}, \varphi)$ can be decomposed by the gradients of $\ln f_r(\mathbf{x}, p_r)$ and $\ln f_t(\mathbf{x}, p_t)$ with respect to the vector $\mathbf{x}$. They are computed as:

$$\nabla_x \ln f_s(\mathbf{x}, p_s) = \mathbf{D}^T \begin{pmatrix} v_1^r(\mathbf{x}, p_s) \\ \cdots \\ v_m^r(\mathbf{x}, p_s) \end{pmatrix} \tag{4}$$

where the $i$th components of the vectors $\mathbf{v}^{\mathbf{s}}(x, p_s)$, of dimension $m$ (the number of voxels) are defined as follows, for the OAR with index $r$,

$$v_i^r(\mathbf{x}, p_r) = \frac{-n_r f_r(\mathbf{x}, p_r)}{\sum_{j \in S_r} d_j(\mathbf{x})^{a_r}} \left( \frac{gEUD_r(\mathbf{x}, a_r)}{EUD_r^0} \right)^{n_r} A_i^r(\mathbf{x}, a_r) \tag{5}$$

and for the PTV with index $t$

$$v_i^t(\mathbf{x}, p_t) = \frac{n_t f_t(\mathbf{x}, p_t)}{\sum_{j \in S_t} d_j(\mathbf{x})^{a_t}} \left( \frac{EUD_t^0}{gEUD_t(\mathbf{x}, a_t)} \right)^{n_t} A_i^t(\mathbf{x}, a_t) \tag{6}$$

with $A_i^s(\mathbf{x}, a_s) = d_j(\mathbf{x})^{a_s - 1}$ if $i \in S_s$ and $A_i^s(\mathbf{x}, a_s) = 0$ in other case, where $S_s$ represents the set of voxels of the structure $s$. Thus, Equations (5) and (6) are the keys to compute the gradient vector in $\mathbf{x}$.

The GD algorithm for determining the optimal fluence map $\mathbf{x}$, with fixed parameters $\varphi$, consists of a stepping process. At each step, the gradient is evaluated at each point $\mathbf{x}$ to guide its movement in the direction of the gradient. The size of these movements is called the **step** size. Subsequently, a new step is initiated, and this process is repeated until the movements of $\mathbf{x}$ become negligible or the maximum number of steps is reached.

A key aspect to consider is the computational complexity of every iteration of the GD algorithm due to the large dimension of the $\mathbf{x}$ fluence maps, the deposition matrix $\mathbf{D}$, and its transpose $\mathbf{D}^T$. This imposes a high computational burden, as the operations required to manipulate and process these large data sets are very demanding. This fact is especially evident here, given the presence of these operations throughout the whole GD algorithm.

### 2.1.3 | PersEUD: Evolutionary Method to Adjust the gEUD Parameters

To automatically tune the parameters of the gEUD, we consider the combination of a multi-objective optimization algorithm with a gradient search-based method [12]. In particular, for the current implementation, the MOEA/D algorithm has been selected [13].

In the optimization, every array $\varphi_i$ defines one individual, and collections of individuals denoted by $\Phi = \{\varphi_0, \ldots, \varphi_g\}$ are associated with the generations of populations computed at each evolutionary iteration. In our proposal, we define feasibility ranges for each parameter type to reduce the search space and generate clinically acceptable plans. The feasibility ranges start at a small number higher than zero to prevent division by zero errors in Equation (2) during the optimization process. All of these collections represent potential solutions and are collectively referred to as the population within the automation scheme.

The first step is an initialization stage of the population, $\Phi = \{\varphi_0, \ldots, \varphi_g\}$. Subsequently, an iterative process is carried out using both GD and evolutionary optimization methods in a two-level approach. The number of generations is specified as an input parameter and determines when the evolutionary process ends.

The first level of the iterative process is managed by the evolutionary optimization algorithm, which controls the population of candidate solutions by applying variation operators such as mutation, crossover, and selection [13]. At the second level, the new generation of solutions is evaluated using the custom-coded GD [14]. This evaluation takes patient data, deposition matrix, beamlet geometry, segmentation of the Region of Interest (ROI), and parameters $\varphi$ as inputs. Next, for each parameter vector or individual, $\varphi$, the corresponding optimal fluence $x^\star$ is computed, and the objective function $F(x^\star)$ is evaluated accordingly. Where $F(x^\star)$ expresses the physical prescriptions that physicists consider in their manual adjustments. Physical prescriptions or constraints on PTV and OAR are expressed in Table 2 with the corresponding definitions with $LB_s$ and $UB_s$ the lower and upper bounds for the dose in any voxel of the structure $s$ and $\overline{LB_s}, \overline{UB_s}$ the lower and upper bounds for the average dose in the structure $s$.

**TABLE 2** | Prescribed physical constraints.

| Definitions | PTV constraints | OAR constraints |
|---|---|---|
| $D_s^{\min}(\mathbf{x}) = \min_{j \in S_s} d_j(\mathbf{x})$ | $D_t^{min}(\mathbf{x}) \geq LB_t$ | Parallel OARs : |
| $\overline{D_s(\mathbf{x})} = \frac{1}{\lvert m_s \rvert} \sum_{j \in S_s} d_j(\mathbf{x})$ | $\overline{D_t(\mathbf{x})} \geq \overline{LB_t}$ | $\overline{D_r(\mathbf{x})} \leq \overline{UB_r}$ |
| $D_s^{\max}(\mathbf{x}) = \max_{j \in S_s} d_j(\mathbf{x})$ | $\overline{D_t(\mathbf{x})} \leq \overline{UB_t}$ | Serial OARs : |
| | $D_t^{\max}(\mathbf{x}) \leq UB_t$ | $D_r^{max}(\mathbf{x}) \leq UB_r$ |

According to the notation introduced, the constraint violations in structures, PTVs, and OARs are captured by the following functions:

$$
\begin{aligned}
C_s^{min}(\mathbf{x}) &= \begin{cases} LB_s - D_s^{\min}(\mathbf{x}) & \text{if } LB_s \text{ is defined } \& \; LB_s > D_s^{\min}(\mathbf{x}) \\ 0 & \text{otherwise} \end{cases} \\
\overline{C_s^{min}}(\mathbf{x}) &= \begin{cases} \overline{LB_s} - \overline{D_s(\mathbf{x})} & \text{if } \overline{LB_s} \text{ is defined } \& \; \overline{LB_s} > \overline{D_s(\mathbf{x})} \\ 0 & \text{otherwise} \end{cases} \\
\overline{C_s^{max}}(\mathbf{x}) &= \begin{cases} \overline{D_s(\mathbf{x})} - \overline{UB_s} & \text{if } \overline{UB_s} \text{ is defined } \& \; \overline{UB_s} < \overline{D_s(\mathbf{x})} \\ 0 & \text{otherwise} \end{cases} \\
C_s^{max}(\mathbf{x}) &= \begin{cases} D_s^{\max}(\mathbf{x}) - UB_s & \text{if } UB_s \text{ is defined } \& \; UB_s < D_s^{\max}(\mathbf{x}) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
\tag{7}
$$

Therefore, the following function $\mathcal{F}(\mathbf{x})$ aggregates constraint violations in all structures and can address the search for admissible clinical plans.

$$
\mathcal{F}(\mathbf{x}) = \sum_{s \in S} C_s^{\min}(\mathbf{x}) + \overline{C_s^{\min}}(\mathbf{x}) + \overline{C_s^{\max}}(\mathbf{x}) + C_s^{\max}(\mathbf{x}) \tag{8}
$$

Although the proposed methodology is effective and provides good plans, it comes at a significant computational cost. This is because it typically requires multiple iterations to converge to an optimal solution, with a large number of candidate solutions evaluated at each iteration. However, the evaluation process itself can be a bottleneck, as the custom-coded GD method is computationally intensive and can require significant time and memory resources. The next section is focused on the implementation approaches to efficiently compute a set of plans on multicore servers.

## 2.2 | Batched PersEUD: Automated gEUD Tuning on Multicore

Figure 1 provides a visual representation of the batched PersEUD structure, which is primarily made up of three key elements: A genetic algorithm (GA), a ZeroMQ router, and the batched gEUD method based on the GD, named gEUD-GD.

The GA plays a crucial role in the efficient exploration of the parameter space. It works with a population of individuals (each

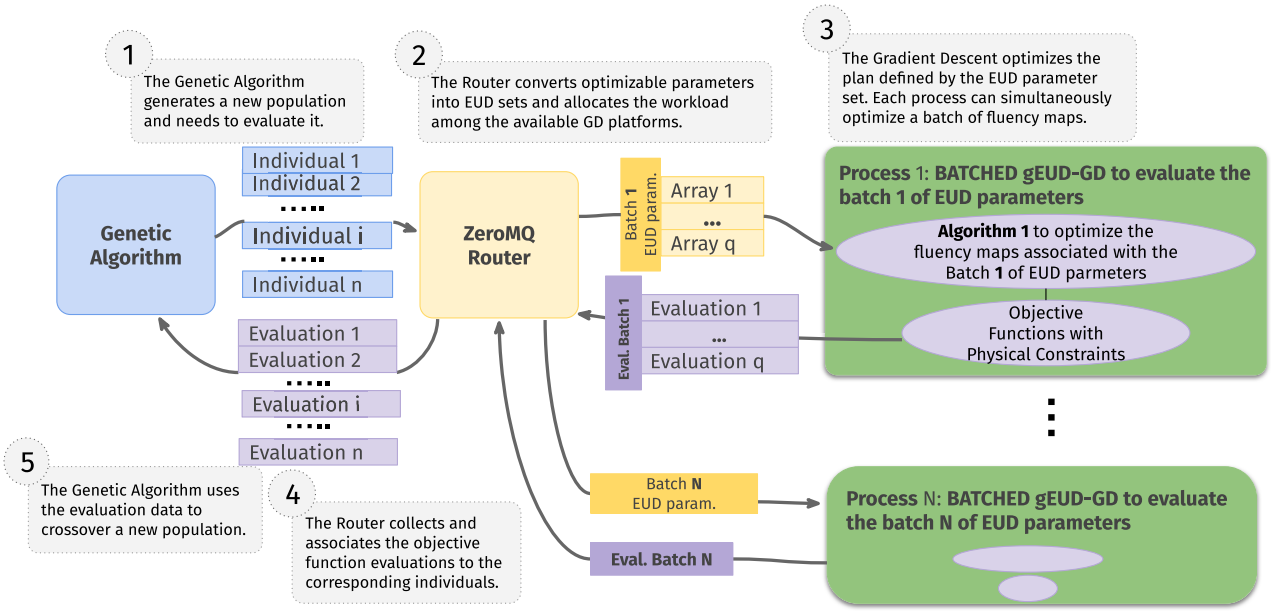



**FIGURE 1** | Scheme for the batched version of PersEUD.

representing a set of parameters $\Phi$), and aims to generate new candidate individuals through processes such as mutation or crossover (step 1). The ZeroMQ router enables communication and coordination between the GA and gEUD-GD. It converts the individuals into input for gEUD-GD (step 2). In step 3, all candidates are evaluated in a two-phase process: First, for every array of parameters gEUD-GD, the optimal fluence maps are computed, and then the objective function $\mathcal{F}(\mathbf{x})$ is evaluated. Once evaluated, the fluence maps are sent to the ZeroMQ router in step 4, which collects and sorts all the evaluations, associating each solution with the respective candidate individual that generated it. Finally, in step 5, the GA updates its records and continues the optimization process by generating a new set of candidate individuals.

As discussed in Section 2.1.2, step 3 consumes most of the huge computational cost of PersEUD. Algorithm 1 describes these steps. Each one is computationally demanding, as it includes products of the large deposition matrix and its transpose with the fluence maps and the matrix operations to compute the vectors involved in the gradients. Therefore, our main focus has been to accelerate gEUD-GD.

On the one hand, we store the sparse matrices $\mathbf{D}$ and $\mathbf{D}^T$ in compressed form to take advantage of the typically larger memory pools available on multicore servers. This maintains spatial locality in memory access and improves the runtime of both products with the deposition matrix and its transpose.

On the other hand, in the context of matrix computations, it is well known that operations involving matrices and vectors (Level 2 BLAS) generally suffer from limited memory management efficiency, which leads to memory-bound computations. In contrast, matrix–matrix operations (Level 3 BLAS) benefit from improved memory access locality and more effective use of cache hierarchies. As a result, these operations are significantly more efficient and can even become compute-bound in dense cases [15]. This point strongly impacts our application's performance due to the large dimensions of the matrices.

The structure of PersEUD allows us to translate the gEUD-GD iterations in terms of high-level matrix operations (2, 3 of BLAS). But, the development of a new batched version of the custom GD has been necessary. It is described in Algorithm 1 where several fluence maps are optimized in the same iteration or step. This scheme allows us to express the computation of the gradients of a batch of fluence maps in terms of higher-level matrix operations to improve the performance on multicore. We have defined the computation of optimal solutions of the gEUD model using batches of $q$ fluence maps, denoted as $\mathbf{x}_0, \ldots, \mathbf{x}_{q-1}$, which can be columns of dense matrices $\mathbf{X}$. This allows us to express their optimization in terms of matrix-matrix products, $\mathbf{DX}$, rather than a set of several matrix-vector products $\mathbf{Dx}_i$. The matrix of a batch of $q$ fluence maps, $\mathbf{X}$, has $q$ columns and $b$ rows, where $b$ is the total number of beamlets in the IMRT device.

The Algorithm 1 describes the implemented gEUD-GD in such a way that, at each step of the GD, it moves $q$ fluence maps $\mathbf{X} = (\mathbf{x_0}, \ldots, \mathbf{x_{q-1}})$ for $q$ arrays of gEUD parameters, $\Phi = (\varphi_0, \ldots, \varphi_{q-1})$. The goal of every step is to compute the gradient vectors of every fluence map $\mathbf{x}_i$ Equation (4) by means of the vectors $\mathbf{v^s}(\mathbf{x}, p_s)$ Equations (5) and (6). In this way, it starts with the calculation of volume doses, $\mathbf{DX}$ and also two magnitudes are computed for every ROI, $s$, the gEUD functions (Equation 1) and the vectors $\sigma_s^i = \sum_{j \in S_s} (d_j^i)^{a_s^i}$ with $i = 0, \ldots, q-1$. Next, for every ROI, the objective functions $f_s^i$ and their derivatives are computed. Then, we can compute the vectors $\mathbf{v^s}(i)$ as functions of the quantities already computed. They are the columns of the matrix $\mathbf{V}$, and we can compute the batch of $q$ gradients as the product of matrices $\nabla\mathbf{X} = \mathbf{D}^T\mathbf{V}$.

The most computational load of Algorithm 1 is related to the products $\mathbf{DX}$ and $\mathbf{D}^T\mathbf{V}$ (lines 4 and 35 in blue color in Algorithm 1). Both have a complexity of $m\,b\,q$, that is, it increases linearly with the number of voxels, beamlets, and batch size. Therefore, both operations consume relevant runtime due to the large dimensions of the problem. The rest of the computation to determine the matrix $\mathbf{V}$ (lines from 6 to 33 in Algorithm 1) has a lower complexity; however, it also consumes an appreciable run time. But the batched scheme defined in Algorithm 1 to execute gEUD-GD exhibits high parallelism levels that can be exploited by multicore architectures.

We have developed a parallel version of gEUD-GD to take advantage of the computational power of modern multicore. Every multicore processor includes libraries of optimized implementations to efficiently compute matrix operations, by enhancing memory management and utilizing computational resources like vector units and multiple cores to carry out matrix computations efficiently by the exploitation of several parallelism levels. For example, Intel supplies Math Kernel Libraries (MKL) for sparse and dense matrix operations[1]. Our scheme of batched gEUD-GD allows us to use these libraries to accelerate the computation of $\mathbf{DX}$ and $\mathbf{D}^T\mathbf{V}$. Furthermore, the computation of the matrix $\mathbf{V}$ involves a significant number of vector operations with high parallelism. They have been parallelized using OpenMP directives to compute the gradient for each structure in the body (PTV and OARs). Therefore, our batched implementation of PersEUD can take advantage of efficient memory management with the batched structure and the parallel computation provided on multicore servers.

**ALGORITHM 1** | Batched implementation of the gEUD-based Gradient Descent to optimize a batch of $q$ fluence maps $\mathbf{X} = [\mathbf{x_0}, \ldots, \mathbf{x_{q-1}}]$ for $q$ arrays of gEUD parameters, $\Phi = (\varphi^0, \ldots, \varphi^{q-1})$.

1: Initialize variables: The batch of fluences $\mathbf{X}$ and $q$ arrays of gEUD parameters in $\Phi$
2: **while** the algorithm is running **do**
3: Compute doses from fluences
4: $\mathbf{d^0} \cdots \mathbf{d^{q-1}} \leftarrow \mathbf{D}\,[\mathbf{x_0} \cdots \mathbf{x_{q-1}}] \equiv \mathbf{DX}$
5: **for** each ROI $s$ **do**
6: Compute parameters involved in the gradient of all ROIs
7: $\sigma_s^0 \cdots \sigma_s^{q-1} \leftarrow \sum_{j \in S_s} (d_j^0)^{a_s^0} \cdots \sum_{j \in S_s} (d_j^{q-1})^{a_s^{q-1}}$
8: $gEUD_s^0 \cdots gEUD_s^{q-1} \equiv gEUD_s(\mathbf{d^0}) \cdots gEUD_s(\mathbf{d^{q-1}})$
9: **end for**
10: **for** each OAR $r$ **do**
11: Calculate function $f_r$ and its partial derivative $\frac{\partial lnF}{\partial gEUD_r}$
12: $f_r^0 \cdots f_r^{q-1} \equiv f_r(gEUD_r^0) \cdots f_r(gEUD_r^{q-1})$
13: $\delta_r^0 \cdots \delta_r^{q-1} \equiv \frac{\partial lnF}{\partial gEUD_r}\Big|_{(gEUD_r^0)} .. \frac{\partial lnF}{\partial gEUD_r}\Big|_{(gEUD_r^{q-1})}$
14: **end for**
15: **for** each PTV $t$ **do**
16: Calculate function $f_t$ and its partial derivative $\frac{\partial lnF}{\partial gEUD_t}$
17: $f_t^0 \cdots f_t^{q-1} \equiv f_t(gEUD_r^0) .. f_t(gEUD_t^{q-1})$
18: $\delta_t^0 \cdots \delta_t^{q-1} \equiv \frac{\partial lnF}{\partial gEUD_t}\Big|_{(gEUD_t^0)} \cdots \frac{\partial lnF}{\partial gEUD_t}\Big|_{(gEUD_t^{q-1})}$
19: **end for**
20: **for** each ROI $s$ and voxel $i$ **do**
21: **if** the voxel $i$ belongs to the ROI **then**
22: Calculate the voxel-specific component of vector $\mathbf{V}$ Equations (5) and (6):
23: $v_i^s(0) \equiv v_i^s(gEUD_s^0, \sigma_s^0, f_s^0, \delta_s^0)$
24: $\cdots$
25: $v_i^s(q-1) \equiv v_i^s(gEUD_s^{q-1}, \sigma_s^{q-1}, f_s^{q-1}, \delta_s^{q-1})$
26: **else**
27: $v_i^s(0) \cdots v_i^s(q-1) \leftarrow 0 \cdots 0$
28: **end if**
29: **end for**
30: **for** each voxel $i$ and ROI $s$ **do**
31: Reduce partial gradients in the vectors:
32: $v_i^0 \cdots v_i^{q-1} \leftarrow \sum_{s \in S} v_i^s(0) \cdots \sum_{s \in S} v_i^s(q-1)$
33: **end for**
34: $\mathbf{v^k}$ vector of elements $v_i^k$ a column of matrix $\mathbf{V}$, find the gradient of the fluences:
35: $\nabla\mathbf{x^0} \cdots \nabla\mathbf{x^{q-1}} \leftarrow \mathbf{D^T}(\mathbf{v^0} \cdots \mathbf{v^{q-1}}) \equiv \nabla\mathbf{X} = \mathbf{D^T V}$
36: Move the fluences in the direction of the gradients:
37: $\mathbf{X} \leftarrow \mathbf{X} + \mathbf{step} \cdot \nabla\mathbf{X}$
38: Smooth the fluence using a convolution kernel:
39: $\mathbf{x^0} \cdots \mathbf{x^{q-1}} \leftarrow \mathbf{smooth}(\mathbf{x^0}) \cdots \mathbf{smooth}(\mathbf{x^{q-1}})$
40: **end while**

## 3 | Results

In this section, we present the computational experiments carried out to test the behavior and performance of the proposed scheme. First, in Section 3.1 we define the context of the computational experimentation, focusing on the real cases we are dealing with, as well as the hardware platforms where we are going to test our proposal. Next, in Section 3.2, we analyze how the batched scheme affects the execution time and speedup of Batched PersEUD. To do so, we consider different batch sizes but only one thread. Finally, in Section 3.3, we perform a series of experiments increasing the number of threads and taking advantage of multicore platforms.

### 3.1 | Study Cases and Hardware Platforms

For the experimental phase of this study, we have solved three real Head and Neck (H&N) IMRT cases. Consequently, the number of OARs remains consistent in all three cases, specifically, nine OARs for each case. Table 3 shows this particular detail. All the cases aim to fulfill physician dose prescriptions on the PTV while keeping the dose in the OAR below the physician-prescribed maximum (for serial organs) or average maximum (for parallel organs). Furthermore, to be able to generate the dose deposition, our optimizer uses a dose deposition model developed by researchers at the Warsaw University of Technology [16].

Table 3 shows the main parameters of the three cases. It is important to note that the computational needs of a patient can wildly vary, even in patients with tumors in the same region of the body, due to the geometries of the PTVs and the patient's organs. The number of nonzeros of the matrix $\mathbf{D}$ (which stores the interaction between beamlets and voxels) can be used to estimate the computational needs of each patient.

As described in Section 2.2, this work proposes two strategies to improve planning speed. In the first strategy, multiple plans are calculated at the same time by batching together their

**TABLE 3** | Plan specifications for each test case.

| Parameter | Patient A | Patient B | Patient C |
|---|---|---|---|
| Beam angles | 9 | 9 | 9 |
| Beamlets ($b$) | 25,298 | 33,911 | 30,265 |
| Voxels ($m$) | 145,965 | 160,786 | 94,647 |
| **D** nonzeros | 67,544,881 | 106,792,251 | 64,991,188 |
| Organs At Risk (OARs) | 9 | 9 | 9 |
| Planning Target Volumes (PTVs) | 3 | 2 | 3 |
| $PTV_0$ pr. dose (Gy) | 54.0 | 59.4 | 54.0 |
| $PTV_1$ pr. dose (Gy) | 60.0 | 66.0 | 60.0 |
| $PTV_2$ pr. dose (Gy) | 67.5 | — | 66.0 |

**TABLE 4** | Specifications of the testing platforms.

| Platform | CPU | Cores | RAM |
|---|---|---|---|
| Sandy | Intel Xeon E5-2650 | 16 (2 sockets) | 64 GB DDR3 |
| EPYC | AMD EPYC 7642 | 96 (2 sockets) | 512 GB DDR4 |
| Ryzen | AMD Ryzen 9 5950X | 16 (1 socket) | 32 GB DDR4 |

fluence vectors, transforming the usual sparse matrix–dense vector multiplication used for one plan into a sparse matrix–dense matrix multiplication. In the second strategy, we employ multi-threading programming techniques to improve the performance of the whole program by an efficient exploitation of current multicore architectures.

In an effort to make a generalizable analysis, two result-processing decisions have been made. Firstly, the reported times are related to a single step of Algorithm 1 to move only one fluence map $\mathbf{x_i}$. They can be obtained from the total running time of Algorithm 1 as follows:

$$t_{step} = \frac{\text{TotalRuntime}}{n_{\text{steps}} \cdot q} \tag{9}$$

where $n_{\text{steps}}$ represents the number of GD steps, each of which takes the same amount of time. Meanwhile, dividing by $q$ allows us to easily discern the performance effect of the batch size, without having to take into account the number of plans that are being processed during the same step. Secondly, to abstract the performance results from a specific test patient, we have calculated the average of all the times and accelerations across the three patients. Additionally, to demonstrate the influence of the ROI geometries on performance, we included standard deviations in all the subsequent figures.

On the hardware front, as this work is focused on the multi-threaded performance of PersEUD, we have selected three widely used multicore computational platforms. Table 4 lists the most relevant specifications of each one of them.

The three testing platforms comprise widely used server and desktop architectures. Platforms Sandy and EPYC contain server-segment processors based on the Sandy Bridge and Zen 2 micro-architectures, respectively. They are part of the High-Performance Computing cluster maintained by the Supercomputing–Algorithms research group at the University of Almería, running CentOS 8.2 (OpenHPC 2) and Intel oneAPI MKL 2023.0.0. Platform Ryzen contains a mid-range desktop processor based on the Zen 3 micro-architecture, running Ubuntu 23.04 and Intel oneAPI MKL 2023.0.0.

In the software front, the MOEA/D [13] implementation available in the JMetal 5.10 framework in Java was used [17]. Maven 3.6.3 and OpenJDK 17.0.5_8 were used to compile the project. The Router is implemented in Python 3.8.5 using the ZeroMQ (pyzmq) 24.0.1 library [18].

### 3.2 | Impact of the Batch Size on Performance

Our objective in this first analysis is to elucidate the performance improvement obtained by our fluence batching strategy. With this objective, we tested the program with different batch sizes $q$ and only one thread ($th = 1$). Table 5 depicts the time per GD step of the three patients across the three test platforms, considering batch sizes ($q$) between 1 and 64.

By examining the column $q = 1$ of Table 5, it can be seen that the execution time can vary between patients. This variability is attributed to the different number of nonzeros in the **D** matrix, which stores the interaction between beamlets and voxels (see Table 3). This discrepancy in runtimes provides insight into the different computational requirements of each patient. However, our aim is to evaluate performance independently of the specific patient under consideration, so we chose to calculate and include the mean values in the last row of the table. The reason for this choice is to provide a more generalized perspective of performance, ensuring that the analysis is not influenced by the individual characteristics of each patient. Therefore, from now on, we will rely on these mean values for the three patients, as we believe that this approach is more suitable for a comprehensive analysis of performance.

A closer examination of Table 5 reveals the same pattern regardless of the patient considered, that is, performance improves as batch size increases, up to a threshold of $q = 4$, on all platforms. However, beyond this point, increasing the batch size does not result in further improvement. This behavior is independent of the particular platform under study.

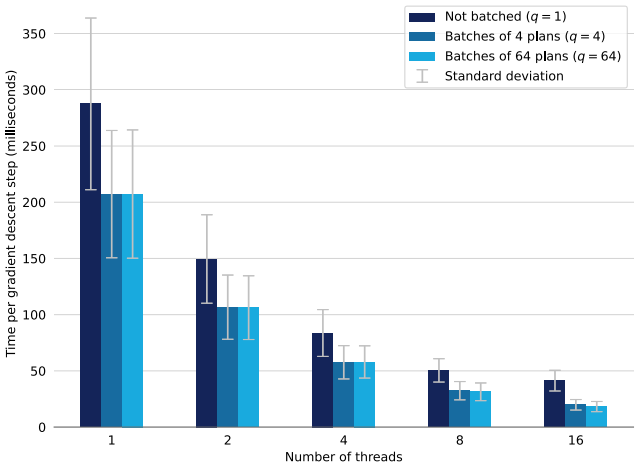
Conducting a more in-depth analysis of the performance across different platforms, we can see that platform Sandy, being the oldest, is, as expected, the slowest. Conversely, Platform Ryzen demonstrates the best single-threaded performance, surpassing even the pricier EPYC platform. This can be attributed to Ryzen's high-end desktop processor, emphasizing single-threaded performance, while EPYC server-line processors prioritize multitasking capabilities[2].

### 3.3 | Performance Analysis of the Parallel Versions on the Test Platforms

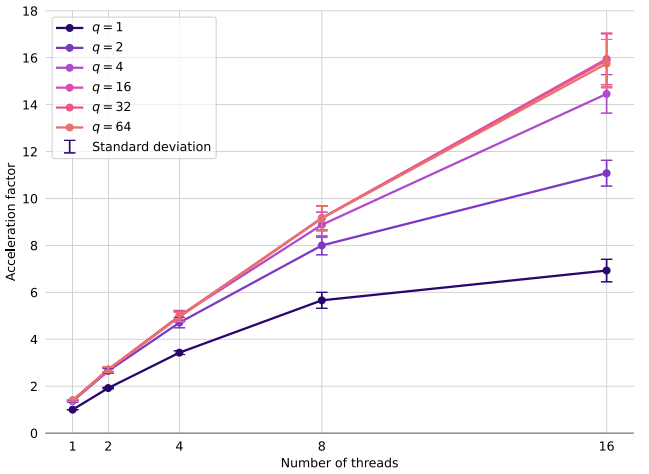
Evaluating $q$ fluence maps with only 1 thread does not fully exploit the parallelism level of PersEUD. Consequently, we will now analyze the effect of parallelizing the entire workload using

**TABLE 5** | $t_{step}$ (ms) defined in Equation (9) for different batch sizes using one thread ($th = 1$) on the three test platforms.

| | | $q = 1$ | $q = 2$ | $q = 4$ | $q = 16$ | $q = 32$ | $q = 64$ |
|---|---|---|---|---|---|---|---|
| Patient A | Sandy | 252.59 | 189.63 | 185.4 | 186.27 | 186.48 | 186.63 |
| | Zen2 | 105.47 | 74.29 | 70.07 | 69.93 | 69.77 | 69.83 |
| | Ryzen5 | 75.63 | 49.01 | 42.42 | 42.16 | 41.9 | 41.78 |
| Patient B | Sandy | 375.08 | 277.32 | 271.51 | 271.51 | 271.34 | 271.79 |
| | Zen2 | 154.86 | 106.83 | 99.09 | 99.29 | 99.25 | 99.18 |
| | Ryzen5 | 116.44 | 72.71 | 63.3 | 63.98 | 62.99 | 63.73 |
| Patient C | Sandy | 234.83 | 166.86 | 164.76 | 163.85 | 163.92 | 163.41 |
| | Zen2 | 96.16 | 65.9 | 60.07 | 59.92 | 59.94 | 60.01 |
| | Ryzen5 | 70.96 | 44.82 | 36.81 | 37.96 | 38.21 | 37.59 |
| Average | Sandy | 287.5 | 211.3 | 207.2 | 207.2 | 207.2 | 207.3 |
| | Zen2 | 118.8 | 82.3 | 76.4 | 76.4 | 76.3 | 76.3 |
| | Ryzen5 | 87.7 | 55.5 | 47.5 | 48.0 | 47.7 | 47.7 |




(a) Average of $t_{step}$ of the three patients considered in the evaluation with different $q$.



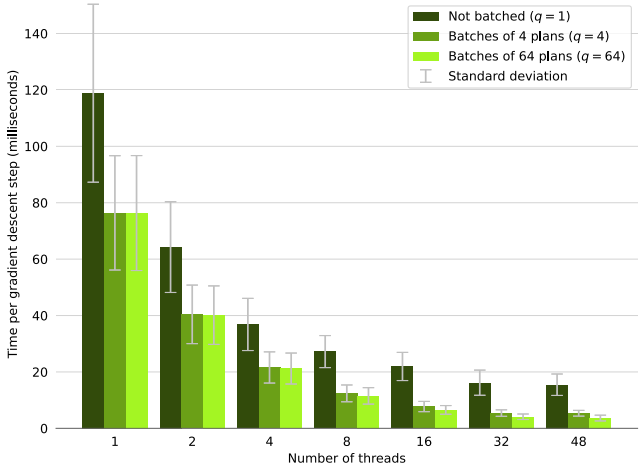

(b) Average multithreaded acceleration with the sequential runtime of the three patients for $q = 1$ as baseline.

**FIGURE 2** | Evaluation on the platform Sandy.

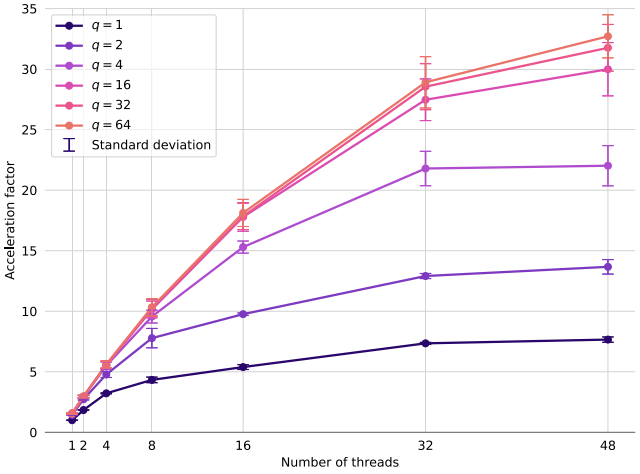
multi-threading, that is, with $th > 1$. To carry out the analysis of this section, the mean values of the three patients have been considered, since they provide a more complete and representative picture, smooth out individual patient variations, and allow a more generalized understanding of the parallel version performance. However, for completeness, we have included the statistical confidence interval in the graphs to show the variability of the different patients.

Figures 2–4 display the average run times on the left and the corresponding speed-up factors on the right for the three test platforms, evaluated across different batch sizes and thread counts. To obtain the acceleration factors, we have considered the sequential execution time for $q = 1$ as a baseline, allowing us to measure how much the parallel versions accelerate the sequential process on each platform with different batch sizes. As a consequence, we obtain acceleration 1 for ($th = 1, q = 1$) and higher than 1 for ($th = 1; q > 1$), as discussed in Section 3.2.

Figure 2 shows the results on the Sandy platform. We have tested the run-times and acceleration for a number of threads ranging from $th = 1$ to $th = 16$ (the number of cores on Sandy) with several values of batch sizes $q = 1, \ldots, 64$. To simplify the bar plot on the left, we have shown the results for only representative values of $q$, that is, $q = 1, 4, 64$. As can be seen, the run time decreases significantly as the number of threads increases. For a fixed thread count, the impact of batch size resembles that observed in the sequential version: As $q$ increases, the run time decreases until it stabilizes at a certain value. This stable run time is achieved at higher values of $q$ as the thread count increases. An analogous trend is observed for the acceleration factor. For example, with $th = 2$, the max acceleration is obtained for $q = 4$, while with $th = 16$, it is reached with $q = 16$. Analyzing the impact of batch size on parallel scalability reveals that, as $th$ increases, acceleration factors approach the ideal value (equal to the number of threads, $th$) only for larger values of $q$. Thus, batching substantially improves the parallel scalability of gEUD on the multicore Sandy platform.

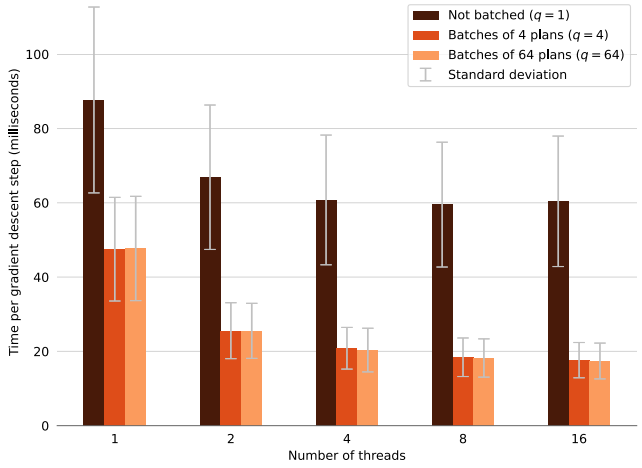



(a) Average of $t_{step}$ of the three patients considered in the evaluation with different $q$.

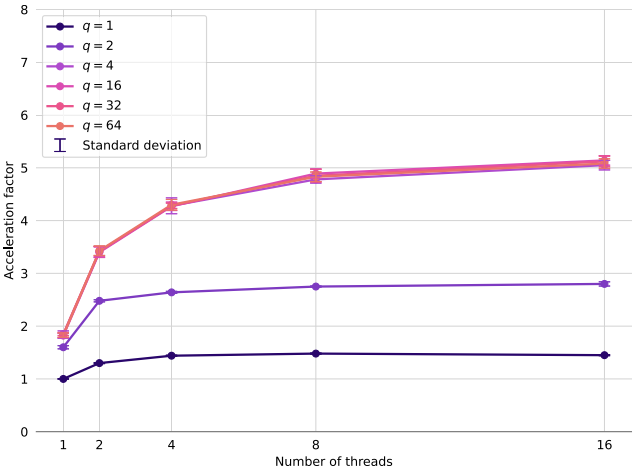



(b) Average multithreaded acceleration with the sequential runtime of the three patients for $q = 1$ as baseline.

**FIGURE 3** | Evaluation on the platform EPYC.




(a) Average of $t_{step}$ of the three patients considered in the evaluation with different $q$.




(b) Average multithreaded acceleration of the three patients with the sequential runtime for $q = 1$ as baseline.

**FIGURE 4** | Evaluation on the platform Ryzen.

Figure 3 shows the results on the EPYC platform, which has 96 cores across two sockets. We tested thread counts from $th = 1$ to $th = 96$, but observed minimal improvements in acceleration for $th > 48$. Therefore, to clarify the analysis, we plot only the results ranging from $th = 1$ to $th = 48$ in the figure. The trends in run-times and acceleration with respect to $th$ and $q$ are similar to the Sandy platform. Only notice that the experimental acceleration is far from the ideal for $th > 16$, even for the highest values of $q$. However, the highest acceleration is also obtained for the highest values of $q$, indicating that batching enhances parallel scalability on the EPYC platform as well.

The test results for the Ryzen platform are shown in Figure 4. We have 16 cores and hence, we have evaluated the performance for $th$ ranging from 1 to 16, similar to the analysis on the Sandy platform. However, the results differ significantly. In the case of Ryzen, the runtime is reduced as $th$ increases. Regarding the acceleration, it achieves a stable value when $th > 4$, although it is higher as $q$ increases. This behavior is due to the Ryzen's desktop-based architecture, which prioritizes single-threaded performance. As a result, the sequential time on Ryzen is the lowest among the three platforms, but its parallel scalability is limited. Even so, the batching impact is also notable in the acceleration achieved on Ryzen.

In general terms, it can be observed that on all platforms, the evaluation time is reduced considerably as the number of threads increases. This behavior is particularly pronounced on the EPYC and Sandy platforms, which are both high-performance server-based architectures. Focusing on the multi-threaded acceleration achieved by each platform, up to $th = 16$, the acceleration is linear for both the EPYC and Sandy platforms, especially with a higher number of plans per batch. Regarding the influence of the batch size, it is reassuring to note that, across the tested configurations with several numbers of threads on the test platforms, higher batch sizes consistently result in better performance and

**TABLE 6** | Mean speed-ups and standard deviations of the three patients. The baseline corresponds to the runtime with a single thread ($th = 1$) and a batch size of $q = 64$.

| Threads | Sandy | EPYC | Ryzen |
|---|---|---|---|
| 1 | $1.00 \pm 0.00$ | $1.00 \pm 0.00$ | $1.00 \pm 0.00$ |
| 2 | $1.95 \pm 0.03$ | $1.90 \pm 0.02$ | $1.87 \pm 0.01$ |
| 4 | $3.56 \pm 0.10$ | $3.60 \pm 0.08$ | $2.34 \pm 0.01$ |
| 8 | $6.59 \pm 0.23$ | $6.63 \pm 0.28$ | $2.61 \pm 0.03$ |
| 16 | $11.32 \pm 0.50$ | $11.62 \pm 0.48$ | $2.73 \pm 0.05$ |
| 32 | — | $18.54 \pm 0.92$ | — |
| 48 | — | $20.99 \pm 0.54$ | — |

parallel scalability. As the batch size increases, the matrix products involved in the gEUD method can be computed more efficiently, potentially resulting in acceleration factors exceeding the number of threads, particularly in configurations with lower thread counts, as shown in Figures 2b, 3b, and 4b. It is important to highlight that the reported accelerations demonstrate the benefits of combining multithreaded and batched implementations, as the non-batched configuration ($th = 1, q = 1$) is used as the baseline in this analysis. This super-linear behavior further confirms the efficiency gains achieved through batching, especially when combined with multithreading.

To better understand the individual contribution of multithreading, we conducted additional experiments using a fixed batch size of $q = 64$, which already benefits from optimized memory access patterns. Hence, the parallel scalability of the batched version is evaluated using the runtimes with $th = 1$ as the baseline (Table 5), and we obtain the results of Table 6. It reports the mean speed-ups and standard deviations observed across three hardware architectures: Sandy, EPYC, and Ryzen. These results confirm that, even when the batch size remains constant, increasing the number of threads alone leads to substantial gains. In particular, we observe near-linear scaling on the Sandy and EPYC platforms up to four threads, indicating efficient multithreaded performance. In contrast, the Ryzen architecture exhibits more limited scalability beyond two threads, likely due to its lower core count and architectural constraints. This table helps isolate the impact of multithreading from batching, demonstrating that while batching contributes most significantly to performance, multithreading also plays an important role, especially on platforms with higher core counts.

Translating these findings into a practical radiotherapy scenario highlights the importance of these time reductions. It's essential to note that the times analyzed here are for a single gradient iteration, but the overall computation time is considerably higher. Our methodology involves an evolutionary algorithm that requires multiple evaluations to achieve an optimal solution, which greatly increases the total time. To reach solutions of clinically acceptable quality, we required a population of 128 individuals evaluated over 50 generations and 2,000 gradient descent steps (see our companion paper [8]). For an idea of the speedup achieved, consider the EPYC platform, the most efficient of the three we tested. The sequential version required 422.51 h, but by evaluating batches of 64 individuals using 96 threads, this time has been reduced to just 12.55 h.

The key here is that we can now provide high-quality solutions in reasonable timeframes. This is a crucial advancement in radiotherapy, where treatment plans often need to be updated and tailored promptly to patient needs. This acceleration allows clinicians to optimize radiotherapy planning without prolonged waiting times, enhancing both treatment efficacy and patient outcomes.

## 4 | Discussion

This paper presents a method designed to take advantage of the capabilities of multicore servers to address IMRT radiation planning problems. The underlying sequential method, PersEUD, works as a two-level optimization scheme. At the top level, the method scans the parameter space for promising parameter combinations, guided by the values of objective functions derived from a multiobjective optimization model. At the lower level, a GD algorithm is employed to optimize the parameters. While PersEUD has been effective in generating high-quality solutions for medical physicists, its computational demands made it impractical for routine clinical use due to the high processing times. In this work, we have focused on significantly reducing the runtime of this method, enabling faster access to optimal solutions without sacrificing quality.

To achieve this, we introduced several optimizations. First, we exploited the PersEUD framework, which allows batch processing of multiple fluence maps simultaneously, combined with parallelization techniques. At each step of the GD method, sets of fluence maps have been adjusted. Then, each step involves products of the large and sparse deposition matrix and its transpose with dense matrices. In this way, the optimization of fluence maps can be expressed in terms of high-level matrix operations that allow us to achieve better performance on multicore processors. This strategic choice enabled us to define batches of individuals generated by the GA, leading to improved computational efficiency.

We further enhanced performance by developing a batched implementation tailored for multicore CPUs. This version integrates Intel Math Kernel Libraries (MKL) for sparse matrix operations and uses custom OpenMP-accelerated code to calculate gradients for each anatomical structure, including the Planning Target Volume (PTV) and Organs at Risk (OARs).

Our results indicate that this batched evaluation approach yields substantial speedups. Evaluating the full population across 50 generations and 2,000 GD steps, the runtime dropped from 422.51 h in the non-batched sequential configuration ($q = 1, th = 1$) to just 12.55 h when processing batches of 64 individuals with 96 threads ($q = 64, th = 96$). This acceleration not only meets the demanding computational requirements of IMRT planning but also brings high-quality, adaptive solutions within reach of routine clinical workflows, ultimately benefiting patient care through more responsive and flexible treatment planning.

As future work, we plan to extend PersEUD to heterogeneous computing environments. In particular, we aim to develop a GPU-batched version of the gEUD method, leveraging the strong potential of these platforms. Moreover, to better manage the computational burden of PersEUD, we will explore additional optimization techniques to further improve both the efficiency and precision of our approach.


### Acknowledgments

This work has been supported by projects PID2021-123278OB-I00 and TED2021-132020B-I00 (funded by MCIN/AEI/10.13039/501100011033/FEDER "A way to make Europe"). Savíns Puertas-Martín is a fellow of the "Margarita Salas" grant (RR_A_2021_21), financed by the European Union (NextGenerationEU). The authors wish to extend their sincere gratitude to: Paweł Kukołowicz and Anna Zawadzka from the Department of Medicine Physics, Maria Sklodowska-Curie National Research Institute of Oncology, Poland, for their invaluable assistance with data acquisition and methodological guidance. Jacek Starzyński and Robert Szmurło from the Faculty of Electrical Engineering, Warsaw University of Technology, Poland, for providing access to their excellent stand-alone dose deposition calculation software. Ignacy Kaliszewski and Janusz Miroforidis from the Systems Research Institute, Polish Academy of Sciences, Poland, for their pivotal role in leading the plan optimization efforts and for generously sharing their extensive expertise in model design.


### Disclosure

The authors have nothing to report.

### Conflicts of Interest

The authors declare no conflicts of interest.

### Data Availability Statement

The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

### Endnotes

[1] https://www.intel.com/content/www/us/en/docs/onemkl/get-started-guide/2025-0/overview.html.

[2] https://cpubenchmark.net/compare/4336vs3862/AMD-EPYC-7642-vs-AMD-Ryzen-9-5950X.

### References


1. B. Cho, "Intensity-Modulated Radiation Therapy: A Review With a Physics Perspective," *Radiation Oncology Journal* 36, no. 1 (2018): 1–10, https://doi.org/10.3857/roj.2018.00122.

2. S. Breedveld, D. Craft, v. R. Haveren, and B. Heijmen, "Multi-Criteria Optimization and Decision-Making in Radiotherapy," *European Journal of Operational Research* 277, no. 1 (2019): 1–19, https://doi.org/10.1016/j.ejor.2018.08.019.

3. P. Ziegenhein, C. Kamerling, M. F. Fast, and U. Oelfke, "Real-Time Energy/Mass Transfer Mapping for Online 4D Dose Reconstruction," *Scientific Reports* 8, no. 1 (2018): 3662, https://doi.org/10.1038/s41598-018-21966-x.

4. M. Karbalaee, D. Shahbazi-Gahrouei, and B. TM, "A Novel GPU-Based Fast Monte Carlo Photon Dose Calculating Method for Accurate Radiotherapy Treatment Planning," *Journal of Biomedical Physics & Engineering* 10, no. 3 (2020): 329–340, https://doi.org/10.31661/jbpe.v0i0.716.

5. F. Liu, N. Jansson, A. Podobas, A. Fredriksson, and S. Markidis, "Accelerating Radiation Therapy Dose Calculation With Nvidia GPUs," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (IEEE, 2021), 449–458, https://doi.org/10.1109/IPDPSW52791.2021.00076.

6. P. Ziegenhein, L. Vermond, J. Hoozemans, et al., "Towards Real Time Radiotherapy Simulation," *Journal of Signal Processing Systems* 92, no. 9 (2020): 949–963, https://doi.org/10.1007/s11265-020-01548-9.

7. F. Liu, M. I. Andersson, A. Fredriksson, and S. Markidis, "Distributed Objective Function Evaluation for Optimization of Radiation Therapy Treatment Plans," in *Parallel Processing and Applied Mathematics. Lecture Notes in Computer Science (13826)*, R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, eds. (Springer, 2023), 383–395.

8. J. J. Moreno, S. Puertas-Martín, J. L. Redondo, et al., "Bi-Level Optimization to Enhance Intensity Modulated Radiation Therapy Planning," *Informatica* 36, no. 1 (2025): 99–124, https://doi.org/10.15388/24-INFOR560.

9. A. Niemierko, "Reporting and Analyzing Dose Distributions: A Concept of Equivalent Uniform Dose," *Medical Physics* 24, no. 1 (1997): 103–110, https://doi.org/10.1118/1.598063.

10. Q. Wu, R. Mohan, A. Niemierko, and R. Schmidt-Ullrich, "Optimization of Intensity-Modulated Radiotherapy Plans Based on the Equivalent Uniform Dose," *International Journal of Radiation Oncology, Biology, Physics* 52, no. 1 (2002): 224–235, https://doi.org/10.1016/s0360-3016(01)02585-8.

11. J. Snyman and D. Wilke, "Practical Mathematical Optimization: Basic Optimization Theory and Gradient-Based Algorithms," in *Springer Optimization and Its Applications University of Pretoria* (Springer International Publishing, 2018).

12. J. Moreno, S. Puertas-Martín, J. Redondo, L. Casado, P. Ortigosa, and E. Garzón, "A New Hybrid Optimization Algorithm to Combine Physical and Biological Criteria to Compute IMRT Planning," in *19th Workshop on Advances in Continuous Optimization (EUROPT 2022) Universidade Nova de Lisboa* (University Nova de Lisboa, 2022).

13. Q. Zhang and H. Li, "MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition," *IEEE Transactions on Evolutionary Computation* 11, no. 6 (2007): 712–731, https://doi.org/10.1109/TEVC.2007.892759.

14. J. Moreno, J. Miroforidis, I. Kaliszewski, and E. Garzón, "Parallel EUD Models for Accelerated IMRT Planning on Modern HPC Platforms," in: *14th International Conference on Parallel Processing and Applied Mathematics (PPAM 2022) Czestochowa University of Technology* (Springer International Publishing, 2023), 139–150, https://doi.org/10.1007/978-3-031-30445-3_12.

15. J. J. Dongarra, L. S. Duff, D. C. Sorensen, and H. A. V. Vorst, *Numerical Linear Algebra for High Performance Computers* (Society for Industrial and Applied Mathematics, 1998).

16. J. Starzyński, R. Szmurło, B. Chaber, and Z. Krawczyk, "Open Access System for Radiotherapy Planning," in *2015 16th International Conference on Computational Problems of Electrical Engineering (CPEE)* (IEEE, 2015), 204–206.

17. J. J. Durillo and A. J. Nebro, "jMetal: A Java Framework for Multi-Objective Optimization," *Advances in Engineering Software* 42, no. 10 (2011): 760–771, https://doi.org/10.1016/j.advengsoft.2011.05.014.

18. ZeroMQ, "ØMQ - The Guide," (2025), https://zguide.zeromq.org/.