Towards Real Time Radiotherapy Simulation

Voss, Nils; Ziegenhein, Peter; Vermond, Lukas; Hoozemans, Joost; Mencer, Oskar; Oelfke, Uwe; Luk, Wayne; Gaydadjiev, Georgi

DOI
10.1007/s11265-020-01548-9

Publication date
2020

Document Version
Final published version

Published in
Journal of Signal Processing Systems

Journal of Signal Processing Systems (2020) 92:949–963
https://doi.org/10.1007/s11265-020-01548-9

Towards Real Time Radiotherapy Simulation

Nils Voss[1,2] · Peter Ziegenhein[3] · Lukas Vermond[2,4] · Joost Hoozemans[2] · Oskar Mencer[2] · Uwe Oelfke[3] · Wayne Luk[1] · Georgi Gaydadjiev[1,2,4]

Received: 2 December 2019 / Revised: 10 March 2020 / Accepted: 5 May 2020 / Published online: 27 June 2020

Abstract

We propose a novel reconfigurable hardware architecture to implement Monte Carlo based simulation of physical dose accumulation for intensity-modulated adaptive radiotherapy. The long term goal of our effort is to provide accurate dose calculation in real-time during patient treatment. This will allow wider adoption of personalised patient therapies which has the potential to significantly reduce dose exposure to the patient as well as shorten treatment and greatly reduce costs. The proposed architecture exploits the inherent parallelism of Monte Carlo simulations to perform domain decomposition and provide high resolution simulation without being limited by on-chip memory capacity. We present our architecture in detail and provide a performance model to estimate execution time, hardware area and bandwidth utilisation. Finally, we evaluate our architecture on a Xilinx VU9P platform as well as the Xilinx Alveo U250 and show that three VU9P based cards or two Alevo U250s are sufficient to meet our real time target of 100 million randomly generated particle histories per second.

Keywords Monte Carlo simulation · FPGA acceleration · Radiotherapy · Dataflow · Dose calculation

1 Introduction

Radiotherapy is a commonly used treatment for various cancer types. High doses of radiation are used to kill cancer cells. Modern radiotherapy relies on an intensity modulation technique that aims to deliver high dose gradients to cancerous tissues while sparing the surrounding healthy organs as much as possible. This is achieved by setting up a therapy treatment plan which takes into account the anatomy as well as the clinical case and dose delivering machine. In order to validate and optimise such therapy plans, the expected spatial dose distribution within the patient has to be simulated before the actual treatment. This is often implemented by Monte Carlo methods which simulate the pathway of millions of radiation particle trajectories as they enter the patient body. These simulations are very accurate. On the other hand, they require relatively long computation times.

Historically, these long computation times were not a problem. However, modern treatment machines in addition to radiation delivery, also allow imaging of the patient during treatment [12]. Real time dose simulation would allow patient treatment adjustments in real time. This is advantageous since, e.g., in the case of prostate or lung cancer target tissue might significantly move between imaging and treatment or even within one treatment session. The usage of real time imaging techniques will enable doctors to adapt to these changes and facilitate accurate radioactive dose delivery. This would minimise dose accumulation in healthy tissue and therefore reduce the damage caused. Additionally, it will be possible to significantly reduce the number of treatments per patient by delivering a higher dose delivery at shorter time due to more targeted radiation. While this would decrease the overall treatment costs and improve treatment quality for the patient, it is crucial to ensure very high accuracy to compensate for the high dosage delivery. As a result of this, the simulation has to be repeated regularly based on new measurements. According to medical experts, the time required for the simulation of the system has to stay below one second to facilitate real time updates.

To solve the computational challenge of real time dose simulation, different technologies have been proposed

✉ Nils Voss
n.voss16@ic.ac.uk

[1] Department of Computing, Imperial College London, London, UK

[2] Maxeler Technologies, London, UK

[3] Joint Department of Physics at The Institute of Cancer Research and The Royal Marsden NHS Foundation Trust, London, UK

[4] Delft University of Technology, Delft, The Netherlands

which utilise Central Processing Units (CPUs) or Graphics Processing Units (GPUs) on local or cloud based systems. However, in the case of CPUs and GPUs the size of the machine required to meet the realtime target is prohibitive. In the case of cloud based systems privacy concerns, bandwidth requirements and latency issues as well as the need to guarantee service quality during treatment provide major challenges for practical deployment.

In this paper, we will discuss the usage of Field-Programmable Gate Arrays (FPGAs) to address these problems in order to build the first real time radiotherapy simulation systems. There is a long history of accelerating Monte Carlo simulations using FPGAs. The inherent parallelism of Monte Carlo simulations allows very high speedups on FPGAs. Additionally, FPGA implementations are highly predictable making them especially suited for real time applications. Finally, the compute density of FPGA based systems is typically superior, allowing for placement directly in the medical facility. As a result, FPGAs are an excellent fit for the problem of real time dose simulation.

Programmability of FPGAs, however, is still a major challenge. Especially for this use case, it is crucial that medical domain experts can fine tune the FPGA design to their needs. To ease the programming we adopted the static dataflow abstraction and Maxeler's MaxCompiler. This provides a higher level of abstraction for the underlying hardware.

The main contributions of this paper are as follows:

- A dataflow architecture for Monte Carlo based dose accumulation simulation;
- An analytical model to estimate hardware usage and accurately assess performance; and
- Evaluation of the architecture and model using implementations based on a Xilinx VU9P FPGA and the Xilinx Alveo U250.

The remainder of the paper is organised as follows. In Section 2 we will discuss the background of radiotherapy and dataflow computing. Section 3 will present related work. Afterwards in Section 4 we will present the architecture used for the FPGA implementation. The performance will be modelled in Section 5. In Section 6 we will present and evaluate our implementation. Finally, Section 7 will conclude the paper and present possible directions for future work.

2 Background

2.1 Monte Carlo Based Dose Simulation for Radiotherapy

Using Monte Carlo simulations to calculate the dose distribution in radiotherapy is widely considered to be the most accurate method. This process relies on simulating individual particles and their trajectories through material representing the patient. The software simulates particle interactions and calculates the dose deposition along the trajectories following fundamental physics laws. However, this accuracy comes at a cost, since a significant amount of particles need to be simulated to achieve statistically significant results.

In our work we will focus on the Dose Planning Method (DPM) [18] implementation of a Monte Carlo technique that simulates the dosimetric effect of high energy photons in organic materials. This algorithm is specifically optimised for the radio therapy use case. DPM provides implementations for all relevant photon-matter and electron-matter interactions that occur in radiotherapy. High efficiency is achieved by optimising the physical interaction description as well as their implementation on modern processors. The authors distinguish between hard interaction processes which have to be calculated analogously and soft interactions which can be accumulated and only simulated once over a certain distance. Especially the latter reduces the simulation time of electron interactions significantly.

The DPM implementation uses a patient cube to store details on the patient as well as the accumulated dose. The cube consists of voxels, which can represent different materials, e.g., bone, tissue or water. In each dimension the cube has a configurable number of voxels, which divide the patient cube volume into equally sized parts. To achieve statistically significant results 100 million particles have to be generated and for real time operation the simulation needs to finish within one second according to medical experts [4]. The real time requirement to finish the simulation within one second is the result of the need to respond to the movement of the cancer cells. These are relative slow movements, e.g., breathing, in the case of lung cancer or the influence of rectal filling in the case of prostate cancer. All these movements happen within a few seconds and thus a single second is sufficient to adapt the treatment accordingly.

The DPM algorithm calculates the dosage by simulating the particle interactions along their individual trajectories. Each generated particle is initialised with energy and fuel values. The fuel values are used to determine the interaction point within the voxel cube and are used up as the particle travels through the cube. Depending on which fuel runs out, an interaction occurs. These interactions can either be a hard inelastic scattering interaction modelled according to Moller, an elastic particle scattering using the Hinge theory or the bremsstrahlung interaction which is the result of the acceleration or deceleration of a charged particle. Dependent on the interaction, different simulation subroutines are used to calculate the resulting dose as well as the changes to the particle direction, velocity, energy and

fuel. If the energy of the particle drops below a threshold it gets absorbed leading to a further dosage accumulation. The dose is deposited in the individual voxel in which the interaction occurs or the particle gets absorbed.

2.2 Dataflow

Streaming dataflow graphs provide a good abstraction for hardware structures. Each node of the dataflow graph represents a hardware unit and each edge represents the wires connecting these hardware units. Maxeler MaxCompiler uses this dataflow concept as main abstraction for the programmer.

MaxCompiler uses the notion of *Kernel*, which represents a single dataflow graph with inputs and outputs. The dataflow graph is scheduled automatically and deeply pipelined to help with timing closure. Due to the dataflow graph description, the inherent parallelism is fully exposed to the compiler. The control logic for the kernel is auto generated and stalls the kernel when either an input is empty or an output becomes full. As a result, the kernel abstraction provides an easy way to implement massively parallel computational hardware structures without requiring deep understanding of the underlying hardware concepts.

Additionally, MaxCompiler uses a *Manager* to describe connections between kernels and all external interfaces. These I/O interfaces include Peripheral Component Interconnect Express (PCIe) and double data rate synchronous dynamic random-access memory (DDR), but also networking like ethernet. I/O interfaces can be created using a single function call. Similarly, only a single operator is necessary to connect these interfaces with each other or user logic. Another block that can be included in the manager is a *State Machine*. A state machine can be used to program custom flow control based on simple push and pull interfaces. As a result state machines are harder to program, but allow the implementation of more complicated and latency critical components, e.g., complex data arbitration tasks.

MaxCompiler targets different FPGA accelerator cards, including in-house developed so called Dataflow Engines (DFEs), Xilinx Alveo cards and the Amazon EC2 F1 instances. The main assumption is that a CPU based host is available and connected to the card. Additionally, the *SLiC* runtime interface can be used to integrate the FPGA design into a normal CPU application utilising Maxeler's proprietary drivers and libraries.

3 Related Work

3.1 Monte Carlo Dose Simulation

Due to the practical relevance of Monte Carlo dose simulation and the high computational requirements related to it a considerable amount of research has focused on accelerating it. This includes algorithmic improvements as presented in [2, 9, 10, 15, 18, 24]. There are also multiple studies which use GPUs to accelerate the workload, e.g., [7] and [20]. In these cases, speedups of up to multiple 100x are reported in comparison to CPU code. However, the authors of [13] and [8] report that this performance advantage is actually a lot smaller, if realistic test cases are considered and the comparison is performed against optimised CPU code. In those cases, the speedup of GPU over CPU implementations is closer to 2.5x.

In addition to the GPU implementations, also CPU based implementations were proposed. Examples for these can be found in [3], [21] and [26]. The latter manages to finish the dose simulation in less than a minute and outperforms well-known GPU implementations.

To facilitate adaptive radiotherapy and the required real time dose simulation, the work in [26] was further expanded in [27] by adding support for cloud computing. The authors propose to use the scalability of cloud based systems to create a bigger cluster of cloud instances to perform the simulation. They manage to reduce the runtime of Monte Carlo dose simulations to values between 1.1 and 10.9 seconds depending on the specific use case. Additionally, they make use of encryption to facilitate privacy for the medical data transferred into the cloud. However, cloud based solutions have the disadvantage of requiring a very good and stable internet connection in the hospital to be useable for reliable medical treatment.

This work presents an extension to [23]. Most notably we extend the implementation to also use a Xilinx Alveo U250 card and provide a significantly more detailed evaluation.

3.2 Monte Carlo Simulations on FPGAs

In [6] the authors propose an FPGA implementation for Monte Carlo based dose simulation. They simulate photons and electrons, where the initial photons are generated by an external source and sent to the FPGA. Afterwards, the dose is calculated and accumulated in the patient cube. However, the patient cube voxels are only saved in on-chip memory, limiting the resolution of the patient cube to 64 voxels in each dimension. A speedup of up to two orders of magnitude compared to a CPU implementation is claimed.

In [5] a methodology to develop FPGA based mixed precision Monte Carlo designs is presented. The authors propose an analytical model to determine the optimal precision and resource allocation for a given Monte Carlo simulation. They combine an FPGA and a CPU to achieve the desired accuracy while using reduced precision. As a result they report speedups of up to 4.6x, 7.1x and

163x compared to recent GPU, FPGA and CPU designs respectively.

The authors of [14] present a domain specific language for the development of Monte Carlo simulations which targets FPGAs and GPUs. They report a 3.7x speedup compared to CPUs for the generated FPGA designs. The advantage of this work is that the user only needs to describe the Monte Carlo simulation using a high level framework based on LaTeX equations to obtain the FPGA design.

A significant amount of other related work exists, in which different Monte Carlo simulations are accelerated using FPGAs. This includes image reconstruction for Single-Photon Emission Computed Tomography (SPECT) [11], pricing of Asian options [17], simulation of electron dynamics in semiconductors [16] and simulation of biological cells [25].

Additionally, there exists work to accelerate particle simulation in hardware. The most prominent examples of these are the ANTON Application-Specific Integrated Circuits (ASICs) [1, 19]. In contrast to our workload they focus on the accurate simulation of atoms to understand proteins, while we focus on particles in a high energy context and the radioactive dosage accumulation caused by them.

The major challenge of accelerating the dose simulation compared to this related work is the difference in data structures and most notably the random memory access into the patient cube. As a result a novel architecture is needed.

4 Architecture

For simplicity the full capability of DPM is not completely implemented. For example, we focus only on the simulation of electrons, do not consider Bremsstrahlung and only use water as material within the patient cube. However, these simplifications have no impact on the feasibility of the architecture and adding them will add only minimal overhead. For the full feature set the following changes are required. Bremsstrahlung is an additional form of particle interaction and as a result only needs more area. Different materials can be implemented as added on-chip memory initialised from DDR. Additionally, interaction equations will need to select different material coefficients dependent on the current voxel. All in all the simulation kernel area will slightly increase and a bit more on-chip memory as well as DDR bandwidth will be required.

One of the major challenges involved in implementing the dose accumulation simulation is the memory access into the patient cube. This is due to the random paths an electron takes through the patient cube. As a result, the position of the memory access into the patient cube to accumulate the dose is also random. We only need to access a few bytes per voxel of the patient cube. If placed in DDR memory, this would lead to a practical bandwidth of less than 10 % of the theoretical achievable bandwidth due to the random access pattern to DDR memory. For this reason the patient cube has to be buffered on-chip.

The on-chip memory capacity of even the largest contemporary FPGAs is not sufficient to store patient cubes of the required resolution for all envisioned use cases. For example the overall memory capacity of a Xilinx VU9P FPGA is just under 48 MB while a patient cube of size 256 in each dimension will require at least 64 MB of memory. As a result, the possible patient cube sizes would be severely limited, since other components also require on-chip memory. To circumvent this issue, we decided to decompose the patient cube into multiple subdomains, where each subdomain fits into on-chip memory. Since we only consider water as material, the on-chip patient cube buffer only needs to store the dose. If other materials are also used, we would also need to store the material type which can usually be encoded in two bits. Due to on-chip buffering of the patient cube, we can perform fully random memory access without impacting performance.

The buffer containing the patient cube is implemented in a kernel. Additionally, this kernel contains the arithmetic to perform the actual simulation of the electrons and the calculation of the emitted dose. As described above, the simulation of the electron decides which interaction occurs. Based on this, the emitted dose is calculated and the values of the electron can be updated. The updated electron moves into a new direction and has updated energy and fuel values. The energy and fuel values determine which interaction occurs and when an electron gets absorbed. In the CPU implementation, a while loop is executed for each externally generated electron, which repeats these steps until the energy of the electron is depleted. This is shown in Fig. 1, which depicts the loopflow graph of the original application. In a loopflow graph the boxes represent loops and arrows represent the flow of data, where the thickness corresponds to the amount of data that flow between the components.

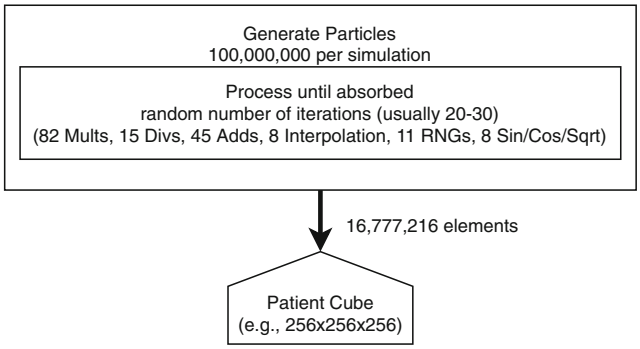

Figure 1 Loopflow Graph of the CPU implementation.

Within the box, the number iterations and the operations executed are indicated.

However, in our case, the kernel accepts new electrons, evaluates the interaction and outputs the updated electrons on every cycle. Since the kernel is deeply pipelined, a loop implementation is not feasible. To circumvent this, the processing order of electrons differs between CPU and FPGA. This is a valid transformation, since all electron interactions are fully independent. As a result, the data arbitration and loop logic are handed off to a different component, which also handles the transport of electrons between subdomains.

Fig. 2 shows the simplified architecture of the application. An *External Particle Generator* kernel generates new electrons and sends them to the *Particle Distributor* state machine. The particle distributor has three inputs, one from the *External Particle Generator*, one from DDR and another from the kernel containing the subdomain buffer and interaction simulation. Additionally, it has outputs to DDR and to the particle simulation kernel. This kernel sends the patient cube back to the host via PCIe once the dose is calculated and forwards the updated electrons to the particle distributor.

The particle distributor handles the arbitration of the electrons and controls the simulation kernel. It decides when the simulation kernel is going to switch to the next subdomain. Additionally, it makes sure that only electrons which are in the current subdomain are sent to the kernel. If they belong to a different subdomain, they are buffered in DDR and read as soon as the kernel switches to the correct subdomain.

The amount of data that have to be stored for each electron approximates the native word width of the DDR4 memory controller. To simplify memory layout, we decided to pad the electron data structure to 512 bits. Additionally, for each subdomain we reserve the same memory capacity. However, this means that only a very limited number of electrons can be buffered in the DDR memory. Since we need to generate around 100 million electrons for statistically significant results and each of those electrons can create multiple additional electrons, this has to be considered.

If the complete simulation is run, it is not possible to buffer all electrons for a specific subdomain in the allocated off-chip memory block. As a result, we split the overall simulation into multiple batches. Within each batch we run through all subdomains, which means that each subdomain of the patient cube is processed multiple times. However, we decided that we could further simplify the architecture by sending the current part of the patient cube back to the CPU once processing of the current batch is finished. As a result, it is not required to accumulate the dose of multiple batches on the FPGA, which removes the requirement to buffer the results of the simulation on-card once processing for the current subdomain is finished. As such, we decided to double buffer the patient cube. Therefore, when processing of a subdomain finishes the buffers can be switched and the now inactive half can be streamed out and set to zero in preparation for the next subdomain.

As a result of splitting the patient cube into subdomains, a problem occurs if an electron updated by an interaction moves into an already processed subdomain. Since we use multiple batches, it is possible to buffer these electrons in DDR for the next batch. However, on the last batch this is not possible. To address this problem, we added another output to the particle distributor which sends those electrons back to the CPU when the last batch is currently processed. Since the amount of electrons sent back is orders of magnitude smaller than the total, it is possible to simulate those electrons on the CPU. We also start the simulation of the electrons sent back as soon as they arrive to overlap the compute time on the CPU and on the FPGA.

In the proposed architecture, DDR memory is used only to buffer electrons. Potentially, the amount of electrons which have to be buffered in DDR is very large. As such, we need to consider the access patterns to optimise the achievable bandwidth. By using long continuous memory access we can get closest to the theoretical peak memory bandwidth.

Reading electrons from DDR is inherently linear, since we can simply read all electrons buffered for a specific subdomain sequentially. However, the access pattern on the write side is not linear. Since the direction of electrons after

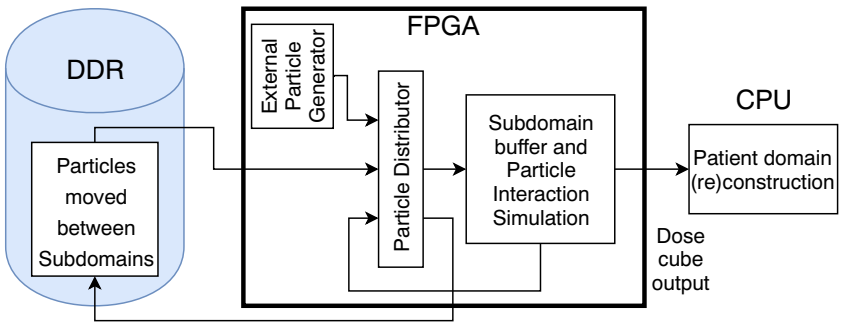

Figure 2 The simplified architecture of the dose accumulation simulation.

interaction is based on random number generators, it is very likely that each electron is written to different parts of the memory. To alleviate this problem, we added an additional state machine, which has small on-chip buffers for each subdomain. We accumulate multiple electrons in these on-chip buffers and only when they are full we write the complete buffer to DDR. Additionally, they can be flushed by the particle distributor to make sure that all electrons for the current subdomains are written to memory, so that they can be read again for processing. We decided to make these buffers hold sixteen electrons, which limits the required on-chip memory capacity but already manages to achieve up to 90% of the peak bandwidth. By packing all individual buffers into a single on-chip memory we can also increase the on-chip memory utilisation. Each individual buffer has a unique address range in the bigger on-chip memory and since only one particle can be received per cycle there is no possibility for write port conflicts. By ensuring that read and write patterns are linear, we are able to significantly improve off-chip memory bandwidth.

The area required for the simulation of a single electron is small compared to the area available on modern FPGAs (see Section 5). As such, we cannot only rely on the pipeline parallelism but also need to exploit algorithm level parallelism to use all available chip resources. We exploit the inherent parallelism of the Monte Carlo simulation on two levels.

The first additional level of parallelism creates multiple instances of the entire design. The motivation for this can be found in the platform we target (see Section 6). We use FPGA accelerator cards based on the Xilinx Ultrascale+ architecture. The big devices used in this work consist of multiple individual dies and interconnectivity between these dies is limited. As such it is often a good idea to treat those dies like separate FPGAs. On the platforms used here, each die is connected to one DDR4 DIMM (dual in-line memory module) and as a result implementing one design on each die is easy. The individual designs only share the PCIe controller and are otherwise completely independent.

The second additional level of parallelism allows us to process multiple electrons in parallel within the same simulation kernel. Parallelising the computation in the kernel itself is accomplished by simply duplicating the dataflow graph. However, the patient cube buffer has to be shared to save on-chip memory resources. As a result, we need to consider potential memory access conflicts. To decrease the likelihood of such events, we implement each xy plane of the cube as a separate memory instance. As a result we have z individual memories holding the data of one xy planes. This will also help with timing closure, since big on-chip memory structures often have problems routing control signals between memory columns.

Another state machine is introduced which checks the electrons coming from the particle distributor for memory access conflicts. Only if the z position of the electrons is different, meaning different physical memories are used, or they access the same memory position, all electrons are sent to the simulation kernel. Otherwise, only a conflict free subset is forwarded. To avoid starving one input, a round robin scheme is used to prioritise all inputs fairly. Since it is non-trivial to parallelise the particle distributor, we decided to instead create one instance of the particle distributor for each electron processed in parallel. This also means that the off-chip memory space has to be equally split between each particle distributor. The overhead introduced by this is negligible, but the implementation complexity is significantly reduced.

The final architecture for a single die where the kernel processes two electrons per cycle is shown in Fig. 3. All arrows, apart from the kernel output sending the dose cube to the host, represent electrons. These connections use FIFOs.

Figure 4 shows the loopflow graph of the application after all the proposed changes for a patient cube size of 256 in all dimensions, split into subdomains of size 128 in x dimension and 64 in all others. Additionally the execution is done in two batches. In comparison to Fig. 2, one can see, that the algorithm structure is more complicated, but it is better suited to hardware acceleration.

To summarise, the main technical challenges are to support big voxel cubes and the processing of multiple electrons within the same kernel using the same on-chip dose memory. The first challenge is addressed by splitting the voxel cube into multiple subdomains and adding particle distribution logic to deal with electrons transitioning between subdomains. Additionally, we add logic to improve memory efficiency and maximise our usable memory bandwidth. The second challenge is addressed by adding a unit for resolving potential memory conflicts at the input of the kernel.

5 Performance Model

The performance model consists of a set of simple equations capturing the most important system characteristics. It is used for rapid design space exploration without running place and route. It also guides the architectural design and is used to evaluate the final implementation. The architecture described above was developed using an iterative process of performance modelling and refinement. We, however, present only the final results. The performance model will be used to verify if our implementation meets our expectations in Section 6.

One of the major challenges in modelling the performance of this application is the extensive use of random

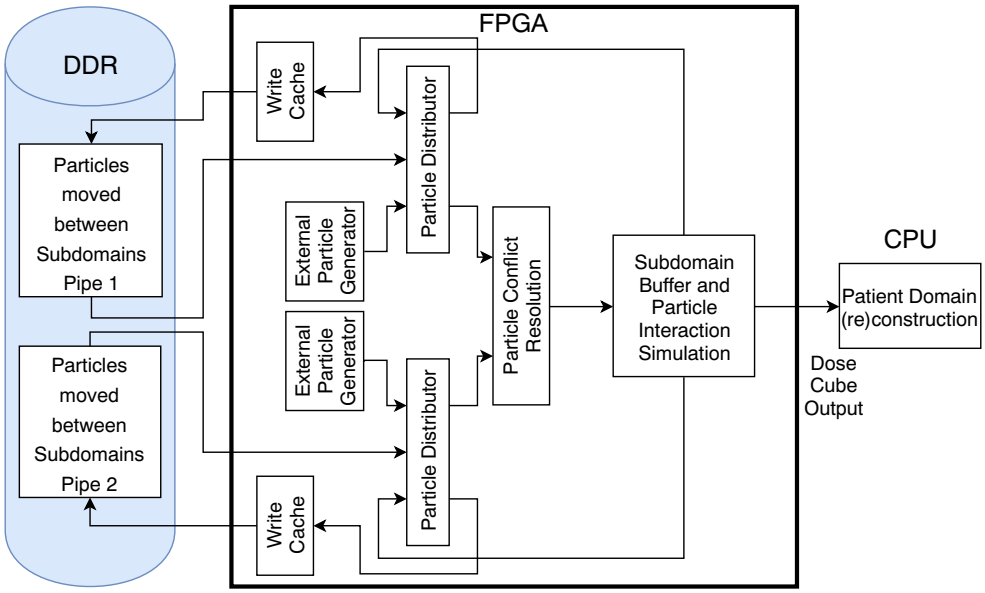

Figure 3 The architecture of the dose accumulation simulation for a single FPGA die if the kernel processes two electrons on every cycle.

number generators. For example, after how many iterations an electron is absorbed and the amount of electrons stored in DDR are both variable. As such, we need to work with estimations based on measurements using the CPU code.

We will denote the amount of electrons updated by interactions before they are absorbed as n_{inter}. The percentage of electrons which move between subdomains, and therefore require DDR buffering, is noted as p_{sub}. Finally, the percentage of cases in which there is a memory access conflict in the patient cube buffer of the simulation kernel is represented by p_{mem}. These factors are also highly dependent on the way in which the external electrons are generated and as a result we will discuss the factors in more detail in Section 6 while keeping all equations generic here.

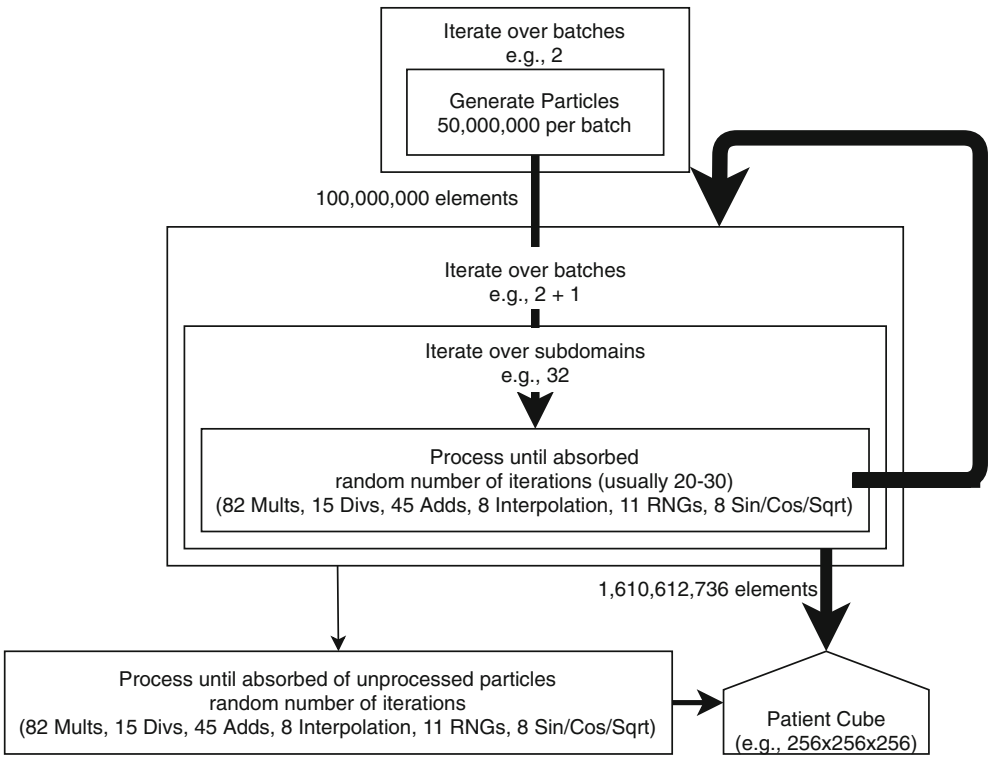

Figure 4 Loopflow Graph of the FPGA implementation.

In this section, we will provide equations for area usage, the achievable electron processing speed, memory bandwidth and finally PCIe bandwidth requirements.

5.1 Area Usage

To forecast the area usage of the implementation we need to count the operations in the CPU code. The simulation of the electrons includes multiple trigonometric functions and square roots. For some of those MaxCompiler offers API functions and we implement the remaining ones as a linear interpolation between values in a ROM lookup table.

Table 1 shows the operation count and the predicted area usage for one simulation kernel which processes one electron per cycle. We determined the area usage for each operation using micro benchmarks and then simply multiplying these with the number of operations needed and calculating the sum over all operations. The area usage will scale linearly with the number of processed electrons per cycle. It should be noted that additional memories and FFs (flipflops) are needed for scheduling of the dataflow graph.

The simulation kernel also contains the memory to buffer the patient cube. The size of this memory depends on the dimensions of the subdomain, x_{sub}, y_{sub} and, z_{sub} in voxels. Additionally we have to consider the depth and width of the physical memories, which we call mem_d and mem_w respectively. Eq. 1 calculates the number of physical on-chip memories required for a single xy plane. The parameter $accWidth$ represents the number of bits required for the datatype used for the dose accumulation. In total z_{sub} of these memories are needed. However, they might use different memory resources, since MaxCompiler will automatically use either BRAMs (Block Random Access Memories) and URAMs (Ultra Random Access Memories).

$$\#mem_{cube} = \left\lceil \frac{accWidth}{mem_w} \right\rceil * \left\lceil \frac{x_{sub} * y_{sub}}{mem_d} \right\rceil \quad (1)$$

In addition to the kernel resource, we also have to consider the state machines and other manager blocks. The state machines predominantly use LUTs (lookup tables), FFs and on-chip memory. We can safely estimate the number of LUTs and FFs required per state machine to be less than 5,000 and 10,000 respectively. The particle distributor does not need any additional memory resources, while the write cache to improve memory efficiency mainly consists of a single buffer. The size of this buffer can be estimated using Eq. 2 with $elec_w$ representing the width of the electron data structure in bits without padding, 417 bits in our case. The depth is determined by the total number of necessary subdomains. d represents the depth of the memory per subdomain, which in our case is 16.

$$\#mem_{cache} = \left\lceil \frac{elec_w}{mem_w} \right\rceil * \left\lceil \frac{\frac{x_{cube}}{x_{sub}} * \frac{y_{cube}}{y_{sub}} * \frac{z_{cube}}{z_{sub}} * d}{mem_d} \right\rceil \quad (2)$$

Lastly, we need to consider the remaining manager blocks. The memory requirements for each FIFO can be estimated using Eq. 3. Usually the depth of a FIFO is 512 and since most FIFOs buffer electrons the width is usually either 417 or 512 bits. Each memory controller requires 3 DSPs (digital signal processing slices), roughly 20,000 LUTs and 30,000 FFs and around 50 BRAMs. Per design instance we will require one memory controller. Finally, the resource requirements for the PCIe controller can be estimated as 8,000 LUTs, 12,000 FFs and 35 BRAMs. The PCIe controller is shared between all instances of the design.

$$\#mem_{FIFO} = \left\lceil \frac{FIFO_w}{mem_w} \right\rceil * \left\lceil \frac{FIFO_d}{mem_d} \right\rceil \quad (3)$$

5.2 Electron Processing Speed

To calculate the electron processing speed, we need to estimate how many electrons can be processed by the kernel at a given frequency. Equation 4 shows how to calculate this. n_{elec} represents the number of electrons processed per second, while n_{design} and n_{pipes} represent the parallelism in terms of number of instances of the design and electrons processed in parallel respectively. Finally f represents the assumed frequency the implementation will be running at.

$$n_{elec} = n_{design} * (p_{mem} + n_{pipes} * (1 - p_{mem})) * f \quad (4)$$

Additionally, we have to consider that for each subdomain the on-chip buffer has to be written back to the host. Normally, this can be overlapped with the compute latency using double buffering. However, if only a very small number of electrons belong to a given subdomain the time required for the computation might not be sufficient to flush the previous buffer. As a result, we need to wait for the previous buffer to be fully written back before we can switch to the next subdomain. The number of cycles required for that per subdomain can be calculated as shown in Eq. 5. In this case, $readout_{width}$ represents the number of voxel values read from the patient cube buffer per cycle. The overlap between the flushing of the patient cube buffer and the

Table 1 Overview of operation count and predicted area usage for the simulation of one electron.

Mults	Divs	Adds	Interpolation	RNG	Sin/Cos/Sqrt	LUT	FF	DSP	BRAM
82	15	45	8	11	8	85,000	120,000	408	54

electron calculation heavily depends on the electron generation pattern.

$$cycles_{flush} = \frac{x_{sub} * y_{sub} * z_{sub}}{readout_{width}} \quad (5)$$

5.3 Memory Bandwidth Requirements

The total amount of data that need to be transferred to and from DDR memory, S_{DDR}, is calculated in Eq. 6. Each electron requires 64 bytes and needs to be written and read only once. Additionally, the data volume depends on the number of electrons created by the external particle generator $n_{elec,total}$. The required bandwidth can then be calculated as a function of the execution time t_{total} as shown in Eq. 7, where DDR_{eff} is the average memory efficiency.

$$S_{DDR} = 2 * 64 * n_{elec,total} * n_{inter} * p_{sub} \quad (6)$$

$$BW_{DDR} = \frac{S_{DDR}}{t_{total}} * \frac{1}{DDR_{eff}} \quad (7)$$

5.4 PCIe Bandwidth Requirements

The PCIe bandwidth requirements are determined by two factors. The patient cube has to be streamed back to the host and in addition we also send the electrons back, which we cannot process within the last batch. Equation 8 estimates the amount of data that have to be transmitted for the patient cube. We assume that all values sent back from the FPGA are converted to single precision floating point, to ease usage on the CPU side of the system. As such, the total amount of data are simply the product of the cube dimensions, the number of batches that are processed and four, the size of single precision floating point number in bytes.

$$S_{PCIe,PatientCube} = x_{cube} * y_{cube} * z_{cube} * batches * 4 \quad (8)$$

Additionally, the amount of data transferred for the electrons that have to be sent back to the CPU is calculated in Eq. 9 based on the number of electrons sent back $n_{electron,PCIe}$. This factor again depends on the external particle generation.

$$S_{PCIe,Electron} = n_{electron,PCIe} * 64 \quad (9)$$

The required bandwidth can be obtained by adding both equations and dividing by the execution time.

5.5 Lessons Learned From the Performance Model

As stated above, the performance model and architecture were iteratively developed alongside each other, where problems identified by the performance model lead to changes to the architecture. We only present the final iteration of this pairing in the paper to avoid confusion, but it has to be stressed that the detailed performance model was crucial to derive the proposed architecture. In this section we plan to highlight a few examples of the impact the performance model had on the architecture development and how it was used for design space exploration.

Using equations similar to those presented in Section 5.1 we determined that it would be infeasible to store the complete voxel cube on-chip and it would need to be stored in DDR memory. We then modelled the required memory bandwidth to access the memory regions along the trajectories of the particles. Due to the random nature of these trajectories it was quickly discovered that an architecture which fully simulates one particle at a time would be heavily limited by off-chip memory bandwidth. To mitigate this issue we decided to instead focus on subregions of the memory cube which are simulated one at a time. This decision leads to most other design decisions in the proposed architecture.

Modelling the memory bandwidth requirements more accurately, we noticed that the process of writing particles which cross subdomains back into the memory might lead to high bandwidth requirements as well. This is caused by the random access pattern. To circumvent this issue we added the write caches.

We also used the performance model to determine what we could send back over PCIe to the host computer at what time. The model showed that the overhead of accumulating the dosage cube on the FPGA would be relatively big and instead it is possible to send each individually processed subdomain to the host where the partial results can be combined. Additionally the model suggested that in most cases it would be faster to send particles which can not be processed in the last batch to the CPU rather then making another iteration.

The hardware builds generated in Section 6 are a direct result of the design space exploration using the performance model. By evaluating the equations for area usage we could quickly determine the possible sizes for the subdomains stored on the chip. Additionally, we calculated how much parallelism was feasible while not requiring too much area or too much IO bandwidth.

6 Evaluation

To evaluate our architecture we implemented it using Maxeler MaxCompiler version 2019.1 and Vivado 2018.3. We target Maxeler's MAX5C Dataflow Engine (DFE) and the Xilinx Alveo U250 as our FPGA platforms. The compute device of the MAX5C DFE is the Xilinx VU9P

FPGA, which consists of 1,182,240 LUTs, 2,364,480 FFs, 6,840 DSPs, 4,320 BRAMs and 960 URAMs. Additionally the card has three 16GB DDR4 DIMMs which provide a peak theoretical bandwidth of 15 GB/s each. The Xilinx Alveo U250 card is based on the VU13P FPGA and offers 1,728,000 LUTs, 12,288 DSPs, 5,376 BRAMs and 1,280 URAMs. The card has four DIMMs of DDR4 memory.

We evaluated our MAX5C based implementation with different degrees of parallelism and cube sizes to run the resulting bitstreams on up to three cards in parallel. For this we used a 2U server powered by two six-core Intel Xeon E5-2643 v4 CPUs running at 3.4 GHz. The U250 design was tested on only one card, which was hosted in a server powered by two 18-core Intel Xeon Gold 6154 CPUs running at 3.0 GHz. Even though we used servers, building a workstation with similar configurations is possible.

6.1 Area Results

We decided to implement eight different configurations, four for the MAX5C and another four for the Alveo U250. In the case of the MAX5C all configurations use three design instances to make optimal use of the three dies of the VU9P. For builds 1 and 3, the simulation kernel processes only one electron per cycle, while builds 2 and 4 process two. In the case of builds 1 and 2, we set the patient cube size to 128 voxels in each dimension and the subdomain size is 64 voxels accordingly. For builds 3 and 4, the resolution is increased and the cube size is set to 256 in each dimension. The subdomain in these cases consists of 128 voxels in x dimension and 64 in y and z.

The four builds targeting the Alveo U250 all consist of four design instances to make optimal use of the four SLRs (super logic regions) and DIMMs available. Builds 5 and 7 process one electron per cycle while 6 and 8 process two. Again, the patient cube size is set to 128 voxels in all dimensions and the subdomain size to 64 voxels for builds 5 and 6. Builds 7 and 8 have a patient cube size of 256 in each dimension and the subdomain size is set to 128 in x dimension and 64 in y and z. An overview of all design points is provided in Table 2 and the area usage for these designs is depicted in Table 3.

The area usage predicted using the equations presented in Section 5.1 as well as the error compared to the actual usage are shown in Table 4. The prediction for DSPs is highly accurate and has no error. The LUT prediction is also very close to the actual usage with errors ranging from an underestimation of 3% to an overestimation of 14.9% compared to the actual usage. In the case of a higher kernel parallelism the LUT prediction is slightly higher than the actual usage, which can be explained by a pessimistic estimation of the area required by the state machines.

The prediction of memory resources is more complicated. The reason for this is that MaxCompiler automatically allocates URAMs and BRAMs, while trying to use roughly the same percentage of available URAMs and BRAMs. This is done to help achieve timing closure on chips consisting of multiple SLRs [22]. In order to provide predictions for the URAM as well as BRAM usage we decided to use the following approach: First, we predict the memory usage using only BRAMs. Then we assume that the same memories can be implemented using URAMs with an efficiency of 50%, since this leads to a balanced mapping between URAMs and BRAMs as targeted by the memory mapping algorithm used by MaxCompiler. This means that eight BRAM18s can be replaced by one URAM. Using this assumption, we can calculate how many URAMs would be needed to implement all on-chip memories. Based on this translation between URAM and BRAM memory capacity we can also calculate the percentage of the overall memory which is available as URAMs and BRAMs respectively. The final resource usage is predicted by multiplying these percentages with the worst case mapping assumptions calculated above.

Applying this method to the design leads to a URAM prediction which is overall very close to the actual usage with an error ranging from -2.9% to 11.6%. The error for the BRAM prediction is significantly larger and in the worst case we underestimate the BRAM usage by up to 38%. The reason for this is that we do not consider the scheduling

Table 2 Design points evaluated for the proposed architecture.

Num	Card	Frequency	Design Count	Kernel parallelism	Cube size	Subdomain size
1	MAX5C	250 MHz	3	1	128	64
2	MAX5C	250 MHz	3	2	128	64
3	MAX5C	250 MHz	3	1	256	128
4	MAX5C	250 MHz	3	2	256	128
5	U250	250 MHz	4	1	128	64
6	U250	250 MHz	4	2	128	64
7	U250	250 MHz	4	1	256	128
8	U250	250 MHz	4	2	256	128

Table 3 Hardware resource usage for the different design points.

Num	LUT	FF	DSP	BRAM	URAM
1	339,030 (28.68%)	641,249 (27.12%)	1,233 (18.03%)	1,209 (27.99%)	414 (43.13%)
2	547,980 (46.35%)	1,071,404 (45.31%)	2,457 (35.92%)	2,535 (58.68%)	468 (48.75%)
3	346,363 (29.30%)	669,903 (28.33%)	1,233 (18.03%)	2,469 (57.15%)	708 (73.75%)
4	558,875 (47.27%)	1,108,486 (46.88%)	2,457 (35.92%)	3,471 (80,35%)	804 (83.75%)
5	431,572 (24.98%)	830,041 (24.02%)	1,644 (13.38%)	1,482 (27,57%)	548 (42.81%)
6	704,282 (40.76%)	1,395,707 (40.39%)	3,276 (26.66%)	3,198 (59,49%)	596 (46.56%)
7	441,146 (25.53%)	868,371 (25.13%)	1,644 (13.38%)	3,066 (57,03%)	948 (74.06%)
8	719,487 (41.64%)	1,445,042 (41.81%)	3,276 (26.66%)	4,350 (80,92%)	1052 (82.19%)

of the dataflow graph, which requires FIFOs that are often implemented using BRAMs. As a result, especially in cases where the degree of parallelism is higher and the dataflow graph is larger our error is bigger as well. This means that in general we are underestimating the memory footprint of the application, which is especially noticeable in the case of larger designs. However, the URAMs are mostly used by MaxCompiler to implement the voxel cube buffers, which can be implemented with a URAM efficiency of 75%, which somewhat mitigates this underestimation. As such we tend to overestimate memory usage for smaller design points while underestimating it for larger design points. To summarise, the area predictions are sufficiently accurate and allowed us to perform a very fast design space exploration.

6.2 Performance Results

The results of processing 100 million externally generated electrons are shown in Table 5. The particles are generated as a single beam, where all electrons enter the voxel cube at the same point with an energy of 6MeV. The offset column indicates whether this voxel is within a subdomain or at the centre of the cube. No offset means that the beam is pointed at the centre of the patient cube. In this case, the cube is entered at the intersection of four subdomains, significantly increasing the number of electrons needing DDR buffering.

We show FPGA and total runtime separately. The total runtime includes the time required to finish simulation for all electrons sent back to the CPU as well as the time required to combine all partial results into one single patient cube. In the case of the Alveo U250, no particles are sent back to the CPU due to limitations of the PCIe controller. As such only the combination of partial results has to be carried out. To measure the runtime we make five independent measurements and report the mean over these five runs. Additionally, the min and max values as well as the standard deviation are provided for the FPGA runtime.

We predicted the runtime and the time required to perform memory transfers using the equations in Section 5. For each combination of offset and build we simulate a smaller run on the CPU to derive the factors determined by random number generators like the number of iterations for each initial electron and the rate of electrons which require buffering in DDR.

We manage to reach our target of sub one second runtime for the complete simulation including CPU execution for run 6. This run uses three MAX5C cards simulating a patient cube of 128 voxels in all directions and processing two pixels in parallel in each of the three design instances. It stands to reason that we could reach the same target using two Alveo U250 cards using build 6. However, we did not have access to the hardware required to verify this.

Since the execution time is dependent on the random numbers generated, one can expect a certain amount of deviation between the predicted runtime and the actual runtime. In most cases the standard deviation between executions is very low and stays below 10 ms and in some cases even below 1 ms. However, e.g., in the case of run

Table 4 Predicted hardware resource usage results for the proposed design points and prediction error.

Num	LUT	FF	DSP	BRAM	URAM
1	338,000 (-3%)	492,000 (-30.4%)	1,233 (0%)	1,888 (36%)	420 (1.4%)
2	623,000 (12%)	912,000 (-17.4%)	2,457 (0%)	2,043 (-24.1%)	455 (-2.9%)
3	338,000 (-2.5%)	492,000 (-36.2%)	1,233 (0%)	3,547 (30.4%)	789 (10.3%)
4	623,000 (10.3%)	912,000 (-21.5%)	2,457 (0%)	3,702 (6,2%)	824 (2.4%)
5	448,000 (3.7%)	652,000 (-27.3%)	1,644 (0%)	2,391 (38%)	570 (3.9%)
6	828,000 (14.9%)	1,212,000 (-15.2%)	3,276 (0%)	2,574 (-24.2%)	614 (2.9%)
7	448,000 (1.5%)	652,000 (-33.2%)	1,644 (0%)	4,506 (32%)	1,073 (11.6%)
8	828,000 (13.1%)	1,212,000 (-19.2%)	3,276 (0%)	4,689 (7.3%)	1,117 (5.8%)

Table 5 Actual and predicted runtime.

Run num	Build num	Cards	Offset	Mean FPGA time [ms]	Stdev FPGA time [ms]	Mean total time [ms]	Predicted compute time [ms]	Predicted DDR time [ms]
1	1	1	yes	2,869.8	1,9	2,968.6	2,667	110
2	1	2	yes	1,452.6	0.9	1,546.6	1,333	55
3	1	3	yes	980.4	1.67	1,088	889	37
4	2	1	yes	2,294.2	2.9	2,443	1,901	69
5	2	2	yes	1,167.8	3.1	1,325.2	950	34
6	2	3	yes	813.2	2.2	967.8	634	23
7	3	1	yes	3,956.2	4.7	4,572.2	3,333	378
8	3	2	yes	2,151.2	20.2	2,832.4	1,667	189
9	3	3	yes	1,652.8	55.8	2,358	1,111	126
10	4	1	yes	3,128.2	12	4,264.6	2,427	351
11	4	2	yes	1,702.8	20	2,814.4	1,214	176
12	4	3	yes	1,460.4	57.6	2,610.6	810	117
13	1	1	no	5,431.2	103.7	7075	2,800	1,364
14	1	2	no	3,539.8	60.2	4,585.8	1,400	682
15	1	3	no	3,134.8	252.9	3,575.6	933	455
16	2	1	no	6,099.2	1119.9	6,060.8	2,011	1,404
17	2	2	no	3,136.2	378.7	3,522.8	1,005	702
18	2	3	no	2,655	128.7	3,079.8	670	468
19	5	1	yes	2,171.4	2.5	2182.4	2,000	83
20	6	1	yes	1,735.8	9	1742.4	1,325	52
22	7	1	yes	3,113.2	34.6	3,179.4	3,333	283
23	8	1	yes	2,989.2	107.8	3,216.4	1,821	264
25	5	1	no	1,986.4	16	2,010.2	2,100	1,023
26	6	1	no	1,735.8	9	1,742.4	1,508	1,053

16, the standard deviation increases significantly and in this case reaches more than 1,000 ms.

In general, there are three common patterns for runs with increased standard deviation of execution times. The first is that multiple electrons are processed in parallel. This can, for example, be observed by comparing runs 1 and 4 or 19 and 20. In these cases the deviation increases slightly for builds which have the same parameters apart from this additional degree of parallelism. The reason for this is that a random chance of conflicts for patient cube memory accesses is introduced. The resolution of these conflicts results in changes to the degree of parallelism possible and as a result impacts the execution time. This deviation can be expected even though the impact is very small.

The second factor is the size of the patient cube. If the patient cube is large as in the cases of builds 3, 4, 7 and 8 the discrepancy between runs is also larger. This has two reasons: Firstly, each voxel represents a smaller portion of the actual patient. As a result particles can travel across more voxels leading to an increase in subdomain changes. Secondly, more data have to be sent back to the CPU, leading to stalls if the PCIe bus is currently congested. This congestion is caused by multiple designs flushing their patient cube at the same time. Overall the introduced deviation is still limited.

The third factor is the position of entry of the particle beam into the patient cube. If the beam enters in the centre of the cube, which is also the boundary of four subdomains, the standard deviation of execution times increases significantly. By comparing the deviation that occurs in runs 13 to 18 with runs 25 and 26, one can see that only for the first two cases the deviation is high while for the other it is not. The only major difference between both designs, apart from the additional design copy, is that in the case of the U250 based design particles are not sent back to the CPU if they cannot be processed on the card itself. Indeed, closer examination of the run shows that a significant portion of the particles is sent back to the CPU. Since this data transfer is not based on Direct Memory Access (DMA) but on active polling, it seems like the CPU execution and the order in which threads get access to the data mostly cause this high deviation.

We also need to evaluate the accuracy of our predictions. For runs 1 to 6 the prediction is accurate even though it is a bit too optimistic. In general we predict the system to be roughly 100 to 200 ms faster than in reality. The reason for

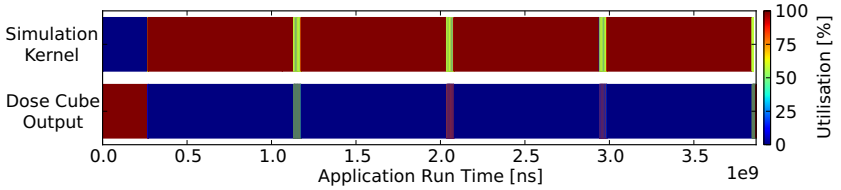

Figure 5 Visualisation of the kernel and PCIe activity for run number 7.

this prediction error is twofold: Firstly, we ignore the system initialisation time, which can be in the order of hundreds of milliseconds for Maxeler systems. Secondly, there is a high likelihood that multiple designs are sending their patient cube data back to the CPU at the same time leading to an additional small delay.

Figure 5 shows the activity of the compute kernel as well as of the output, which is used to send the patient cube back to the CPU for run 7. For these runs (7-12) the difference between predicted and actual runtime starts to grow, but is still limited to an error of 50%. There are two interesting observations that can be made from it. First it takes roughly 200 ms until the kernel becomes active. This verifies that the initialisation overhead is partly causing the discrepancy between actual and predicted runtime. The reason for the higher than expected initialisation time can be traced back to the seed initialisation of the random number generators. It is possible to remove this overhead by sharing the seed between multiple initialisations and not resetting the random number generator in between runs to not generate the same numbers on every run.

The second observation is that the kernel execution sometimes has to pause for patient cube data to be sent over PCIe back to the CPU. This is caused by subdomains for which no or only very small numbers of calculations have to be performed. In our performance model we assume that it is possible to overlap data transfer and calculation, which is only possible if for each subdomain a sufficiently high number of calculations is performed. This adds an additional overhead in the order of a few hundred milliseconds.

To mitigate this effect we could, for example, skip subdomains which have nothing to compute or increase the PCIe bandwidth by switching to PCIe Gen. 3, for which driver support is in development. Additionally, it is expected that the addition of Bremsstrahlung will change the distribution of particles across subdomains removing this problem altogether.

For runs 13 to 18 the difference between the actual and predicted runtime is the biggest. Again by comparing to runs 25 and 26 we can conclude that this is caused by the transmission of particles back to the CPU. While it might be possible to partly mitigated this problem by higher PCIe bandwidth, it seems more promising to perform more batches without the generation of new particles on the FPGA in these special cases. Additionally, we think that it is possible to further optimise the CPU code and to avoid this worst case in the actual deployment of the system altogether by shifting the position of the subdomains accordingly.

For the designs using the U250 the same patterns can be seen as for the builds using the MAX5C card as discussed in detail above. To summarise, the model seems to be highly accurate for the most straightforward cases as represented by runs 1 to 6 as well as 19 and 20. If the cube size is increased or the particle beam enters the patient cube at the boundary of subdomains effects that were not included in the model have significant impact on the runtime leading to bigger prediction errors. We think it is possible to mitigate most of these effects in the future.

6.3 Comparison to Traditional Systems

A fair comparison to related work for this application is not easy to accomplish, since the precise test case is often not reproducible or accessible due to patient data protections. In [26] the authors report execution times of 10.8 seconds for a patient cube of size 256x256x234 on a two socket Intel Xeon system. Additionally they report a speedup of 1.95x compared to the GPU implementation presented in [7], which is based on a single GPU card. A similar test case on our system (Run 12) takes 2.6 seconds including the not fully optimised CPU code. Since our test case is slightly larger the comparison should be slightly biased towards the CPU and GPU based technologies. As a result we achieve a speedup of 4.1x compared to the CPU and 8x compared to the GPU implementation for a slightly larger patient cube.

The FPGA as well as the CPU and GPU systems can be realised as a single workstation system. In this use case the space efficiency of the solution is the most important metric, since the simulation system should be placed close to the treatment machine to reduce latencies and minimise potential network issues. As such the simulation system has to fit into the existing rooms available for the treatment which are often in dedicated underground buildings due to radiation shielding requirements. The price of the simulation system is negligible compared to the costs of the complete installation.

7 Conclusion and Future Work

In this paper, we presented an FPGA based implementation for real time Monte Carlo dose simulation for adaptive

radiotherapy. We proposed an architecture which decomposes the voxel cube representing the patient into multiple subdomains to reduce on-chip memory space requirements. The performance and area usage for this architecture were modelled using simple equations to predict the hardware implementation characteristics. Finally, we presented eight implementations of the architecture and showed that in the most typical cases the performance model provides an accurate indication of the measured runtime. We manage to fulfil our realtime goals of simulating 100 million electrons in less than a second using three FPGA cards for a voxel cube with a size of 128 in all dimensions.

Future work will include the implementation of Bremsstrahlung, additional particle types, support of different materials as well as an increase of the PCIe bandwidth. The integration of our approach into the latest adaptive radiotherapy systems will also be explored.

Acknowledgments The support of the United Kingdom EPSRC (grant number EP/L016796/1, EP/N031768/1, EP/P010040/1, EP/S030069/1 and EP/L00058X/1), Maxeler, Intel and Xilinx is gratefully acknowledged.

References

1. Ant (2014). The ANTON 2 chip a second-generation ASIC for molecular dynamics. In *2014 IEEE Hot Chips 26 Symposium (HCS)* (pp. 1–18), https://doi.org/10.1109/HOTCHIPS.2014.7478807.
2. Buckley, L.A., Kawrakow, I., Rogers, D.W.O. (2004). CSNrc: correlated sampling Monte Carlo calculations using EGSnrc. *Medical Physics*, *31*(12), 3425–3435.
3. Cassidy, J., Nouri, A., Betz, V., Lilge, L. (2018). High-performance, robustly verified Monte Carlo simulation with FullMonte. *Journal of Biomedical Optics*, *23*(8), 1–11. https://doi.org/10.1117/1.JBO.23.8.085001.
4. Chetty, I.J., Charland, P.M., Tyagi, N., McShan, D.L., Fraass, B.A., Bielajew, A.F. (2003). Photon beam relative dose validation of the DPM Monte Carlo code in lung-equivalent media. *Medical Physics*, *30*(4), 563–73.
5. Chow, G.C.T., Tse, A.H.T., Jin, Q., Luk, W., Leong, P.H., Thomas, D.B. (2012). A Mixed Precision Monte Carlo Methodology for Reconfigurable Accelerator Systems. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '12* (pp. 57–66). New York: ACM. https://doi.org/10.1145/2145694.2145705.
6. Fanti, V., Marzeddu, R., Pili, C., Randaccio, P., Siddhanta, S., Spiga, J., Szostak, A. (2009). Dose calculation for radiotherapy treatment planning using Monte Carlo methods on FPGA based hardware. In *2009 16th IEEE-NPSS Real Time Conference* (pp. 415–419). https://doi.org/10.1109/RTC.2009.5321468.
7. Jia, X., Gu, X., Graves, Y.J., Folkerts, M., Jiang, S.B. (2011). GPU-Based fast Monte Carlo simulation for radiotherapy dose calculation. *Physics in Medicine and Biology*, *56*(22), 7017–7031. https://doi.org/10.1088/0031-9155/56/22/002.
8. Jia, X., George Xu, X., Orton, C.G. (2015). GPU Technology is the hope for near real-time Monte Carlo dose calculations. *Medical Physics*, *42*(4), 1474–1476. https://doi.org/10.1118/1.4903901.
9. Kawrakow, I., Kling, A., Barão, F.J.C., Nakagawa, M., Távora, L., Vaz, P. (Eds.) (2001). *VMC++, Electron and photon monte carlo calculations optimized for radiation treatment planning*. Berlin: Springer.
10. Kawrakow, I., & Fippel, M. (2000). Investigation of variance reduction techniques for Monte Carlo photon dose calculation using XVMC. *Physics in Medicine and Biology*, *45*(8), 2163–2183. https://doi.org/10.1088/0031-9155/45/8/308.
11. Kinsman, P.J., & Nicolici, N. (2013). Noc-based FPGA Acceleration for Monte Carlo Simulations with Applications to SPECT Imaging. *IEEE Transactions on Computers*, *62*(3), 524–535. https://doi.org/10.1109/TC.2011.250.
12. Lagendijk, J.J.W., Raaymakers, B.W., Raaijmakers, A.J.E., Overweg, J., Brown, K.J., Kerkhof, E.M., van der Put, R.W., Hårdemark, B., van Vulpen, M., van der Heide, U.A. (2008). MRI/linac integration. *Radiotherapy and Oncology*, *86*(1), 25–29. https://doi.org/10.1016/j.radonc.2007.10.034.
13. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P. (2010). Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput Archit News*, *38*(3), 451–460. https://doi.org/10.1145/1816038.1816021.
14. Lindsey, B., Leslie, M., Luk, W. (2016). A Domain Specific Language for accelerated Multilevel Monte Carlo simulations. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (pp. 99–106). https://doi.org/10.1109/ASAP.2016.7760778.
15. Ma, C.M., Li, J.S., Pawlicki, T., Jiang, S.B., Deng, J., Lee, M.C., Koumrian, T., Luxton, M., Brain, S. (2002). A Monte Carlo dose calculation tool for radiotherapy treatment planning. *Physics in Medicine and Biology*, *47*(10), 1671–1689. https://doi.org/10.1088/0031-9155/47/10/305.
16. Negoi, A., & Zimmermann, J. (1996). Monte carlo hardware simulator for electron dynamics in semiconductors. In *1996 International semiconductor conference. 19th edition. CAS'96 proceedings*, (Vol. 2 pp. 557–560). https://doi.org/10.1109/SMICND.1996.557443.
17. Schryver, C.d., Shcherbakov, I., Kienle, F., Wehn, N., Marxen, H., Kostiuk, A., Korn, R. (2011). An Energy Efficient FPGA Accelerator for Monte Carlo Option Pricing with the Heston Model. https://doi.org/10.1109/ReConFig.2011.11.
18. Sempau, J., Wilderman, S.J., Bielajew, A.F. (2000). DPM A fast, accurate monte carlo code optimized for photon and electron radiotherapy treatment planning dose calculations. *Physics in Medicine and Biology*, *45*(8), 2263–2291. https://doi.org/10.1088/0031-9155/45/8/315.
19. Shaw, D.E., Deneroff, M.M., Dror, R.O., Kuskin, J.S., Larson, R.H., Salmon, J.K., Young, C., Batson, B., Bowers, K.J., Chao, J.C., et al. (2008). Anton, a special-purpose machine for molecular dynamics simulation. *Communications of the ACM*, *51*(7), 91–97. https://doi.org/10.1145/1364782.1364802.

20. Townson, R., Jia, X., Zavgorodni, S., Jiang, S. (2012). SU-E-t-476: GPU-based Monte Carlo Radiotherapy Dose Calculation Using Phase-Space Sources. *Medical Physics*, *39*(6 Part 17), 3814–3814. https://doi.org/10.1118/1.4735565.
21. Tyagi, N., Bose, A., Chetty, I.J. (2004). Implementation of the DPM Monte Carlo code on a parallel architecture for treatment planning applications. *Medical Physics*, *31*(9), 2721–2725. https://doi.org/10.1118/1.1786691.
22. Voss, N., Quintana, P., Mencer, O., Luk, W., Gaydadjiev, G. (2019). Memory mapping for multi-die fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 78–86). https://doi.org/10.1109/FCCM.2019.00021.
23. Voss, N., Ziegenhein, P., Vermond, L., Hoozemans, J., Mencer, O., Oelfke, U., Luk, W., Gaydadjiev, G. (2019). Towards real time radiotherapy simulation. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, (Vol. 2160-052X pp. 173–180). https://doi.org/10.1109/ASAP.2019.000-6.
24. Wulff, J., Zink, K., Kawrakow, I. (2008). Efficiency improvements for ion chamber calculations in high energy photon beams. *Medical Physics*, *35*(4), 1328–1336.
25. Yamaguchi, Y., Azuma, R., Konagaya, A., Yamamoto, T. (2003). An approach for the high speed Monte Carlo simulation with FPGA - toward a whole cell simulation. In *2003 46th Midwest Symposium on Circuits and Systems*, (Vol. 1 pp. 364–367). https://doi.org/10.1109/MWSCAS.2003.1562294.
26. Ziegenhein, P., Pirner, S., Kamerling, C.P., Oelfke, U. (2015). Fast CPU-based Monte Carlo simulation for radiotherapy dose calculation. *Physics in Medicine and Biology*, *60*(15), 6097–6111. https://doi.org/10.1088/0031-9155/60/15/6097.
27. Ziegenhein, P., Kozin, I.N., Kamerling, C.P., Oelfke, U. (2017). Towards real-time photon Monte Carlo dose calculation in the cloud. *Physics in Medicine and Biology*, *62*(11), 4375–4389. https://doi.org/10.1088/1361-6560/aa5d4e.