Towards real time radiotherapy simulation

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

DOI
10.1109/ASAP.2019.000-6

Publication date
2019

Document Version
Accepted author manuscript

Published in
Proceedings - 2019 IEEE 30th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2019

Citation (APA)
Voss, N., Ziegenhein, P., Vermond, L., Hoozemans, J., Mencer, O., Oelfke, U., Luk, W., & Gaydadjiev, G. (2019). Towards real time radiotherapy simulation. In *Proceedings - 2019 IEEE 30th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2019* (Vol. 2019-July, pp. 173-180). Article 8825146 (2019 IEEE 30TH INTERNATIONAL CONFERENCE ON APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS (ASAP 2019)). IEEE. https://doi.org/10.1109/ASAP.2019.000-6

Towards Real Time Radiotherapy Simulation

Nils Voss*†, Peter Ziegenhein‡, Lukas Vermond†§, Joost Hoozemans†,
Oskar Mencer†, Uwe Oelfke‡, Wayne Luk* and Georgi Gaydadjiev*†§
*Department of Computing, Imperial College London, UK
Email: nv916@ic.ac.uk, w.luk@imperial.ac.uk, g.gaydadjiev@imperial.ac.uk
†Maxeler Technologies, London, UK
‡Joint Department of Physics at The Institute of Cancer Research and The Royal Marsden NHS Foundation Trust, UK
§Delft University of Technology, the Netherlands

***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 online 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 and show that three cards are sufficient to meet our real time target of 100 million randomly generated particle histories per second.**

***Index Terms*—Monte Carlo Simulation, FPGA Acceleration, Radiotherapy, Dataflow, Dose Calculation**

I. 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 needs 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 one hand but require relatively long computation times on the other hand.

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 [1]. 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. As a result, using real time imaging techniques, will facilitate accurate radioactive dose delivery. This would minimise dose accumulation in healthy tissue and therefore reduce the risk of new cancer cells growing. Additionally the number of treatments per patient could be reduced, decreasing the overall treatment costs.

To solve the computational challenge of real time dose simulation, different technologies have been proposed which utilise Central Processing Units (CPUs), 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 datacenter FPGA based systems is typically superior. 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 an implementation based on a Xilinx VU9P FPGA.

The remainder of the paper is organised as follows. In section II we will discuss the background of radiotherapy and dataflow computing. Section III will present related work. Afterwards in section IV we will present the architecture

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

used for the FPGA implementation. The performance will be modelled in section V. In section VI we will present and evaluate our implementation. Finally, section VII will conclude the paper and present possible directions for future work.

II. Background

A. 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) [2] 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 technique 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 represents different materials, e.g., bone, tissue or water. In each dimension the cube has a configurable numbers 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.

B. 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 PCIe and DDR but also networking like ethernet. I/O interfaces can be created using a single functional call. Similarly, only a single function call 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 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 proprietary drivers and libraries.

III. Related Work

A. Monte Carlo Dose Simulation

Due to the practical relevance of Monte Carlo dose simulation and the high computational requirements related to it a lot of research has focused on accelerating it. This includes algorithmic improvements as presented in [2]–[7]. There are also multiple studies which use GPUs to accelerate the workload, e.g., [8] and [9]. In these cases, speedups of up to multiple 100x are reported in comparison to CPU code. However, the authors of [10] and [11] 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.

Additionally to the GPU implementations, also CPU based implementations were proposed. Examples for these can be found in [12], [13] and [14]. 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 [14] was further expanded in [15] 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 simulation 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 medical treatment.

To our knowledge there is currently no implementation which manages to meet our real time requirements of finishing the simulation in less than a second.

B. Monte Carlo Simulations on FPGAs

In [16] 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 [17] 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 state of the art GPU, FPGA and CPU designs respectively.

The authors of [18] 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) [19], pricing of Asian options [20], simulation of electron dynamics in semiconductors [21] and simulation of biological cells [22].

IV. 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 additional area. Different materials can be implemented as on-chip memory initialised from DDR. Additionally, interaction equations will use the material coefficients. 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. Per voxel of the patient cube we only need to access a few bytes. This leads to a 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. As a result, 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. 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 externaly generated electron, which repeats these steps until the energy of the electron is depleted. 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 is handed off to a different component, which also handles the transport of electrons between subdomains.

Fig. 1 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.

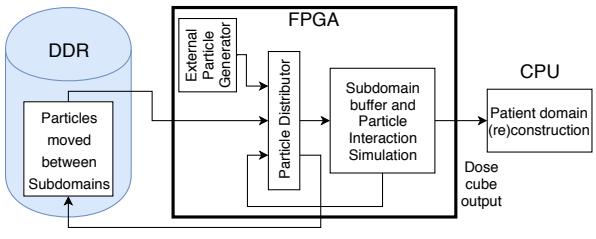

Fig. 1. The simplified architecture of the dose accumulation simulation.

The particle distributor handles the arbitration of the elec-

trons 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 has to be stored for each electron approximates the DDR4 burst size. 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 in 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, the combination of multiple copies of the same subdomain is not required on the FPGA, which removes the requirement to buffer the data on-card. 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. As a result 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 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. 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 V). As such, we can not 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 VI). We use an FPGA accelerator card based on the Xilinx VU9P. The VU9P consists of three individual dies and interconnectivity between these dies is limited. As such it is often a good idea to treat those dies like they would be separate FPGAs. On the platform used here, each die is connected to one DDR4 DIMM 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 compute 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. This will also help with timing closure, since big on-chip memory structures often have problems routing the control signals between multiple 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 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 figure 2. All

arrows, apart from the kernel output sending the dose cube to the host, represent electrons. These connections use FIFOs.

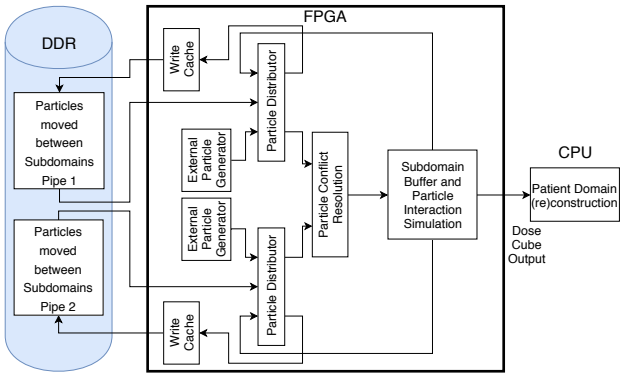

Fig. 2. The architecture of the dose accumulation simulation for a single FPGA die if the kernel processes two electrons on every cycle.

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.

V. 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 evaluates 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 VI.

One of the major challenges in modelling the performance of this application is the extensive use of random 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 number 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 VI while keeping all equations generic here.

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

A. 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 does include 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.

Tab. I 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 multiply these with the number of operations needed and calculate 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 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 and URAMs.

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

In addition to the kernel resource, we also have to consider the state machines and other manager blocks. The state machines predominantly use LUTs, 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 number of subdomains necessary in total. 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

TABLE I
OVERVIEW OF OPERATION COUNT AND PREDICTED AREA USAGE FOR THE SIMULATION OF ONE ELECTRON.

Multiplications	Divisions	Additions	Interpolation	RNG	Sin/Cos/Sqrt	LUT	FF	DSP	BRAM
82	15	45	8	11	8	85,000	120,000	408	54

or 512 bits. Each memory controller requires 3 DSPs, 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.

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

B. 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. Eq. 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.

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

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 compute 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 equation 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 electron calculation heavily depends on the electron generation pattern.

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

C. Memory Bandwidth Requirements

The total amount of data that needs to be transferred to and from DDR memory, S_{DDR}, is calculated in equation 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 equation 7, where DDR_{eff} represents the average memory efficiency.

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

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

D. PCIe Bandwidth Requirements

The PCIe bandwidth requirements are determined by two factors. On one side, the patient cube has to be streamed back to the host and in addition we also send the electrons back, which we can not process within the last batch. Eq. 8 estimates the amount of data that has 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 is simply the product of the cube dimensions, the number of batches that are processed and 4, the size of single precision floating point number in bytes.

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

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.

$$S_{PCIe,Electron} = n_{electron,PCIe} * 64 \tag{9}$$

The required bandwidth can be obtained by calculating the sum of both equations and dividing by the execution time.

VI. EVALUATION

To evaluate our architecture we implemented it using Maxeler MaxCompiler version 2018.3.1 and Vivado 2018.2. We target Maxeler's MAX5C Dataflow Engine (DFE) as our FPGA platform. Its compute device, the Xilinx VU9P FPGA, 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.

We built the implementation with different parallelism degrees 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. Even though we used a server, it is possible to build a workstation with very similar configuration.

A. Area Results

We decided to implement four different configurations. All 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 area usage for these four designs is depicted in tab. II.

The area usage predicted using the equations presented in section V-A are shown in tab. III. The predicted numbers for DSPs and LUTs for designs with a kernel parallelism of one are very close. In the case of a higher kernel parallelism the LUT prediction is slightly higher, which can be explained by a pessimistic estimation of the state machines. Our FIFO and BRAM prediction is usually lower, since we do not include the resources required for the scheduling of the dataflow graph. Additionally, to simplify the prediction task, we only predict BRAM usage and not URAMs. To summarise, the area predictions are sufficiently accurate and allowed us to perform a very fast design space exploration.

B. Performance Results

The results of processing 100 million externally generated electrons are shown in tab. IV. 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 if 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.

We predicted the runtime and the time required to perform memory transfers using the equations in section V. 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. In most cases our predictions are accurate, even though a bit too optimistic. The reason for this is that we omitted initialisation time and the time required to send back subdomains to the CPU when transmission time and compute time can not be fully overlapped. The initialisation time can take up to 100 ms and additionally the time to send back sub domains can take between 60 ms for the smaller patient cube and 500 ms for the bigger one.

In some cases, however, our prediction error is more significant. Usually, the error is bigger if the simulation kernel processes two electrons on every cycle. This can be explained by considering that the chance of two electrons conflicting, therefore reducing the parallelism back to one, is random and has a significant impact on the expected runtime. In our experiments the chance of a conflict occurring was usually around 60%, however the actual timing in hardware might differ significantly from our simulation.

In the cases where the electron beam is sent to the centre of the patient cube the error is even bigger. In these cases the predicted compute time is up to 4 times smaller than the actual time. However, in many of these cases the time for memory transfers needs to be considered. The kernel will stall to access memory, if the access happens in bursts and the DDR memory can not deliver enough data fast enough. This increases the total runtime. Even though it is theoretically possible to overlap compute, memory access and data transfers to the host it turns out that especially in these cases overlap is quite limited. One reason for this is that, the overall run has to be split into significantly more batches to ensure that the off-chip memory address space reserved for each subdomain is sufficient. As a result, up to 64 batches have to be executed. It seems that in many cases not enough electrons belong to one subdomains to overlap streaming back to the host and the compute (see eq. 5). This adds up to 500ms to the runtime.

In run 3 we managed to simulate the required 100 million particles in less than a second even when the post processing on the CPU is included. This requires all three cards. In order to also meet our requirements for bigger voxel cubes we will need to improve the timing characteristics or use more cards.

C. Comparison to Traditional Systems

The comparison to related work for this application is not easy, since the precise test case is often not reproducible. In [14] 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 [8]. A similar test case on our system (Run 12) takes 2.6 seconds including the not fully optimised CPU code. As a result we achieve a speedup of 4.1x compared to the CPU and 8x compared to the GPU implementation.

VII. 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 sub cubes 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 four implementations of the architecture and showed that in most 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 and support of different materials. The integration of our approach into the latest adaptive radiotherapy systems will also be explored.

TABLE II
AREA USAGE RESULTS FOR THE PROPOSED ARCHITECTURE.

Num	Frequency	Design Count	Kernel Parallelism	Cube Size	Subdomain Size	LUT	FF	DSP	BRAM	URAM
1	250	3	1	128	64	337,059 (28.51%)	640,139 (27.07%)	1,233 (18.03%)	1,770 (40.97%)	351 (36.56%)
2	250	3	2	128	64	543,642 (45.98%)	1,069,421 (45.23%)	2,457 (35.92%)	2,916 (67.5%)	384 (40%)
3	250	3	1	256	128	343,918 (29.09%)	669,232 (28.3%)	1,233 (18.03%)	2,562 (59.31%)	672 (70%)
4	250	3	2	256	128	554,637 (46.91%)	1,105,734 (46.76%)	2,457 (35.92%)	3,708 (85,83%)	708 (73.75%)

TABLE III
PREDICTED AREA USAGE RESULTS FOR THE PROPOSED ARCHITECTURE.

Num	LUT	FF	DSP	BRAM
1	338,000	492,000	1,233	2,763
2	623,000	912,000	2,457	3,371
3	338,000	492,000	1,233	5,067
4	623,000	912,000	2,457	5,675

TABLE IV
ACTUAL AND PREDICTED RUNTIME.

Run Num	Build Num	Cards	Offset	FPGA Time [ms]	Total Time [ms]	Predicted Compute Time [ms]	Predicted DDR Time [ms]
1	1	1	yes	2,882	2,977	2,667	110
2	1	2	yes	1,451	1,558	1,333	55
3	1	3	yes	981	1,088	889	37
4	2	1	yes	3,267	3,435	1,901	69
5	2	2	yes	1,173	1,342	950	34
6	2	3	yes	810	988	634	23
7	3	1	yes	3,956	4,612	3,333	378
8	3	2	yes	2,182	2,878	1,667	189
9	3	3	yes	1,589	2,351	1,111	126
10	4	1	yes	3,127	4,241	2,427	351
11	4	2	yes	1,750	2,933	1,214	176
12	4	3	yes	1,387	2,613	810	117
13	1	1	no	7,710	7,987	2,800	1,364
14	1	2	no	4,222	4,596	1,400	682
15	1	3	no	3,475	3,778	933	455
16	2	1	no	4,185	5,084	2,011	1,404
17	2	2	no	3,765	4,134	1,005	702
18	2	3	no	2,642	3,101	670	468

REFERENCES

[1] J. J. W. Lagendijk *et al.*, "MRI/linac integration," *Radiotherapy and Oncology*, vol. 86, no. 1, pp. 25–29, 2019/04/10 2008. [Online]. Available: https://doi.org/10.1016/j.radonc.2007.10.034

[2] J. Sempau, S. J. Wilderman, and A. F. Bielajew, "DPM, a fast, accurate monte carlo code optimized for photon and electron radiotherapy treatment planning dose calculations," *Physics in Medicine and Biology*, vol. 45, no. 8, pp. 2263–2291, jul 2000. [Online]. Available: https://doi.org/10.1088%2F0031-9155%2F45%2F8%2F315

[3] I. Kawrakow and M. Fippel, "Investigation of variance reduction techniques for Monte Carlo photon dose calculation using XVMC," *Physics in Medicine and Biology*, vol. 45, no. 8, pp. 2163–2183, jul 2000. [Online]. Available: https://doi.org/10.1088%2F0031-9155%2F45%2F8%2F308

[4] L. A. Buckley, I. Kawrakow, and D. W. O. Rogers, "CSnrc: correlated sampling Monte Carlo calculations using EGSnrc." *Medical physics*, vol. 31 12, pp. 3425–3435, 2004.

[5] J. Wulff, K. Zink, and I. Kawrakow, "Efficiency improvements for ion chamber calculations in high energy photon beams." *Medical physics*, vol. 35 4, pp. 1328–1336, 2008.

[6] I. Kawrakow, "VMC++, Electron and Photon Monte Carlo Calculations Optimized for Radiation Treatment Planning," in *Advanced Monte Carlo for Radiation Physics, Particle Transport Simulation and Applications*, A. Kling *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 229–236.

[7] C.-M. Ma *et al.*, "A Monte Carlo dose calculation tool for radiotherapy treatment planning," *Physics in Medicine and Biology*, vol. 47, no. 10, pp. 1671–1689, may 2002. [Online]. Available: https://doi.org/10.1088%2F0031-9155%2F47%2F10%2F305

[8] X. Jia *et al.*, "GPU-based fast Monte Carlo simulation for radiotherapy dose calculation," *Physics in Medicine and Biology*, vol. 56, no. 22, pp. 7017–7031, oct 2011. [Online]. Available: https://doi.org/10.1088%2F0031-9155%2F56%2F22%2F002

[9] R. Townson *et al.*, "SU-E-T-476: GPU-Based Monte Carlo Radiotherapy Dose Calculation Using Phase-Space Sources," *Medical Physics*, vol. 39, no. 6 Part 17, pp. 3814–3814, 2012. [Online]. Available: https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.4735565

[10] V. W. Lee *et al.*, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1816021

[11] X. Jia, X. George Xu, and C. G. Orton, "GPU technology is the hope for near real-time Monte Carlo dose calculations," *Medical Physics*, vol. 42, no. 4, pp. 1474–1476, 2015. [Online]. Available: https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.4903901

[12] J. Cassidy *et al.*, "High-performance, robustly verified Monte Carlo simulation with FullMonte," *Journal of Biomedical Optics*, vol. 23, no. 8, pp. 1 – 11, 2018. [Online]. Available: https://doi.org/10.1117/1.JBO.23.8.085001

[13] N. Tyagi, A. Bose, and I. J. Chetty, "Implementation of the DPM Monte Carlo code on a parallel architecture for treatment planning applications," *Medical Physics*, vol. 31, no. 9, pp. 2721–2725, 2004. [Online]. Available: https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.1786691

[14] P. Ziegenhein *et al.*, "Fast CPU-based Monte Carlo simulation for radiotherapy dose calculation," *Physics in Medicine and Biology*, vol. 60, no. 15, pp. 6097–6111, jul 2015. [Online]. Available: https://doi.org/10.1088%2F0031-9155%2F60%2F15%2F6097

[15] ——, "Towards real-time photon Monte Carlo dose calculation in the cloud," *Physics in Medicine and Biology*, vol. 62, no. 11, pp. 4375–4389, may 2017. [Online]. Available: https://doi.org/10.1088%2F1361-6560%2Faa5d4e

[16] V. Fanti *et al.*, "Dose calculation for radiotherapy treatment planning using Monte Carlo methods on FPGA based hardware," in *2009 16th IEEE-NPSS Real Time Conference*, May 2009, pp. 415–419.

[17] G. C. T. Chow *et al.*, "A Mixed Precision Monte Carlo Methodology for Reconfigurable Accelerator Systems," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 57–66. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145705

[18] B. Lindsey, M. Leslie, and W. Luk, "A Domain Specific Language for accelerated Multilevel Monte Carlo simulations," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2016, pp. 99–106.

[19] P. J. Kinsman and N. Nicolici, "NoC-Based FPGA Acceleration for Monte Carlo Simulations with Applications to SPECT Imaging," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 524–535, March 2013.

[20] C. d. Schryver *et al.*, "An Energy Efficient FPGA Accelerator for Monte Carlo Option Pricing with the Heston Model," in *2011 International Conference on Reconfigurable Computing and FPGAs*, Nov 2011, pp. 468–474.

[21] A. Negoi and J. Zimmermann, "Monte carlo hardware simulator for electron dynamics in semiconductors," in *1996 International Semiconductor Conference. 19th Edition. CAS'96 Proceedings*, vol. 2, Oct 1996, pp. 557–560 vol.2.

[22] Y. Yamaguchi *et al.*, "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, Dec 2003, pp. 364–367 Vol. 1.