A Monte-Carlo Simulator for Light Propagation in Turbid Media using High Level Synthesis

Abdul-Amir Yassine, Omar Ismail, and Yasmin Afsharnejad
University of Toronto
Email: {abed.yassine, omar.ismail, yasmin.afsharnejad}@mail.utoronto.ca

***Abstract*—Light has proved to be crucial in several clinical applications in recent years. Therefore, understanding how the light transports inside biological tissues is important to medical researchers. However, the equation that governs such a propagation is computationally expensive, and researchers usually resort to Monte-Carlo (MC) simulations to model the problem accurately. Nevertheless, even MC simulations need acceleration to be beneficial in modern applications. Given the popularity that high level synthesis (HLS) has gained in recent years due to its simplicity, we aim in this project, to accelerate an MC simulator of light propagation in 3D-voxelized tissues on an FPGA. We use Vivado tools to implement the design in HLS on a Kintex Ultrascale Xilinx FPGA. Our results show 174x speedup against the software sequential implementation, and a 3x speedup against the fastest simulator to date.**

***Index Terms*—Monte-Carlo, Light Transport, FPGAs, High Level Synthesis.**

I. Introduction

In recent years, light has been playing a major role in several medical applications, mainly due to its inherent harmless nature and its low cost to produce, guide and measure. Such applications range from imaging techniques as in Diffuse Optical Tomography (DOT), to observing the progression of cancerous tumors in living tissues as in Bioluminescence Imaging (BLI) [1], to even treating tumors through light activated therapy after the administration of photosensitizing drugs as in Photodynamic therapy (PDT) [2]. Therefore, studying how the light transports (scatters, gets absorbed, reflects, refracts, ...etc) in biological tissues is crucial to today's medical field. This transport is governed by a well-known partial differential equation known as the Radiative Transfer Equation (RTE) [3]. However, due to the heterogeneity of biological tissues, and the large number of parameters, this equation is computationally expensive to solve, and cannot be used even with the current advanced computational resources. Therefore, a lot of effort has been put in the past two decades to simulate, model or approximate the light transport in biological tissues.

Looking at the literature, one can see that *Monte-Carlo* (MC) based simulations are the *gold standard* in modeling photon transport in living tissues. MC techniques are generally described as a repetitive random sampling based on some random probability distribution that models the physics behind the problem in order to produce results. In biological tissues, MC methods can be used to simulate several photon samples to get an approximation of the final *fluence* distribution (the amount of energy absorbed per unit volume). However, to get accurate results, millions of photons have to be simulated, and this increases the runtime of the simulation. Therefore, there has been a need for acceleration of the MC methods in recent years [4].

The MC algorithm was first introduced in [5], where the tissue was divided into several 2D layers of different optical properties (absorption and scattering coefficients). The algorithm can be divided into mainly the following four stages:

1) **Launch**: In this step, a photon packet is launched with certain energy (proportional to the source light intensity) and a random direction at the specified source position.
2) **Hop**: In this step, a random step is drawn from an exponential distribution, and the photon is moved to the new position. A region lookup is performed, and if the new position is in a new volume element, a test for reflection (either internal or Fresnel's reflection), refraction and termination is performed (termination happens when the photon exits the tissue). A random step is drawn every time an interface crossing happens.
3) **Drop**: In this stage, a fraction of the photon's energy is dropped in the corresponding volume element based on the scattering and absorption coefficients of that element.
4) **Spin**: In this step, a new random direction of the photon is generated based on the *anisotropy* coefficient of the corresponding volume element.

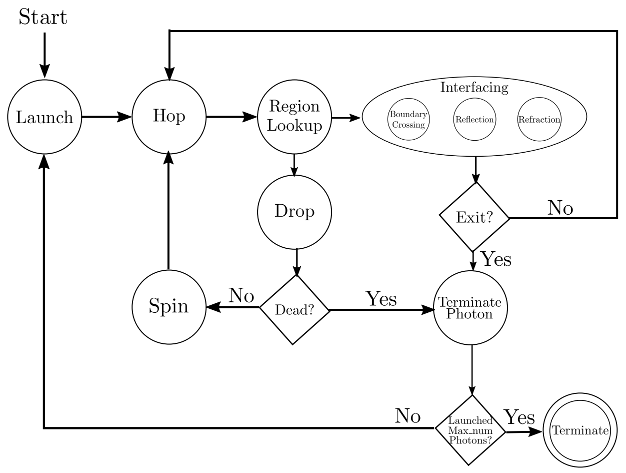

Fig. 1: Overview of the Monte-Carlo algorithm flow

Fig. 1 displays the different stages that the photon pass through, and the overall flow of the monte-carlo algorithm. Since the different stages are independent from each other. Each stage can be mapped exactly to a separate hardware functional block, and data flows between them. Hence, multiple photons can run in parallel in a pipelined manner, where each pipeline stage represent one of the MC stages discussed. This would increase the throughput of the computing platform used. With such a pipelined fashion, the problem hands itself to be implemented on an FPGA, where it can be accelerated at a much less power consumption compared to other computing platforms.

In [4], the authors presented *FullMonte*, the fastest software implementation to date of the MC algorithm, where the tissue was divided into a 3D tetrahedral mesh which gives much higher granularity and more accurate results than the 2D-multi-layered implementation. A similar implementation was done on GPUs in [6]. However, with tetrahedral meshes, the math in the boundary interface stage and the region lookup become harder to take care of. A simpler volume representation, yet with better granularity than the 2D-layered one, is to divide the tissue into equal-sized voxels (3D cubes). To our knowledge, no FPGA implementation of the algorithm has been made with other than a 2D multi-layered volume representation. *The authors of Fullmonte presented an initial FPGA implementation (with tetrahedral meshes) written in* ***Bluespec*** *that showed some promising results, but is yet to be fully optimized [7].* In [8], the authors presented an FPGA implementation of the original MCML algorithm written in **Verilog HDL**. Results showed an 80x speedup against the software implementation. In [9], the authors presented an FPGA implementation of the MCML algorithm written in **OpenCL**. Results showed up to 21x improvement over the sequential MCML code.

In this report, we present an FPGA implementation of the MC light propagation algorithm with a **voxel based** tissue representation (more accurate than MCML) written in a **C++ high level synthesis** language (Vivado HLS). Our results show around 160x speedup against our sequential software implementation with comparable accuracy to the Fullmonte tetrahedral-based implementation [4].

The rest of this report is organized as follows: Section II goes over the current status of our project and what has been implemented so far. Section III presents our initial architectural design, while Section IV reviews the specification evolution throughout the course of the project and the final architectural design. After that, Section V discusses the methodology that we followed in this project. Section VI presents the contributions of each member in the project. Sections VII and VIII review our design characteristics and the technical and non-technical problems that we faced, respectively. Finally, we conclude in Section IX.

II. Current Status

We have implemented all of the promised features for this project, and our contributions can be summarized as follows:

- We wrote a fully tested and working C++ implementation of the Monte-Carlo method for light propagation.
- We wrote several scripts that can extract and visualize the results on Paraview [10] (a 3D visualization software that can read VTK formats [11]).
- We fully optimized the code using Vivado HLS directives, and changed some parts of the code to achieve the desired latency.
- We installed the design on an **ADM-PCIE-8K5** board that has a **Xilinx Kintex Ultrascale KU115-2** FPGA. The design is fully working at a speed of 167MHz. With only one MC kernel, our design can achieve 20x speed-up against the sequential implementation.
- We integrated multiple MC kernels to run in parallel, and achieved around 160x speed-up against the sequential implementation.

In our initial proposal, we promised to integrate a DDR4 module, however, while designing the project, we found out that our design only utilized 12% of the BRAMs on the used FPGA for a tissue of size 8K voxels. This means that we can fit a tissue of size around 64K (which

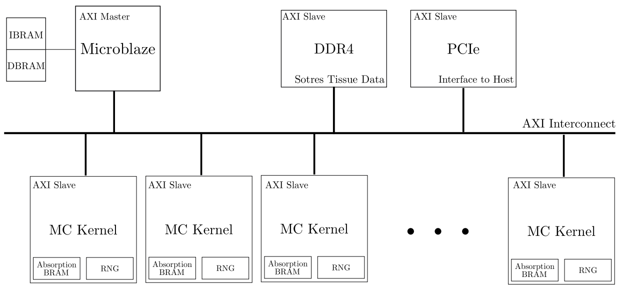

Fig. 2: Initial Block Diagram

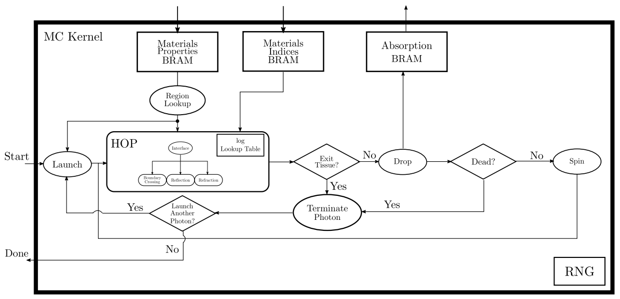

Fig. 3: Overview of the HLS kernel

is good for most biological tissues) without the need of DDR4 memories. If we need to simulate on a larger size, a DDR memory has to be integrated in the design.

Some of the things that can be added to this project are:

- Support of multiple types of light sources. Currently, we only support *point sources*, but one can easily support a line source, which is a superposition of multiple point sources.
- Support of multiple light sources at different positions in the same kernel. Using HLS makes this change much easier. The change to the code should be minimal (one just needs to keep track of all the source positions, and launch photon packets from all of these positions).

III. Initial Architectural Design

This section discuss our initial architectural design shown in Figs. 2 and 3. The discussion is split into two main sub-sections the Initial specifications and the hardware implementation of the initial architecture that was presented in the proposal.

A. Initial specifications

The initial specifications were inspired by the MCML [5] implementation of monte-carlo light simulator. The main functional specifications of the four stages of the algorithm launch, hop, drop and spin were defined according to the prior implementation as described in the introduction.

The model used for the volume representation is a voxel based model (3D cubes). The voxels have a fixed size to control the granularity of the model and to decrease the data needed to represent it. Furthermore, the voxel model was used to decrease the complexity of the math needed to check for the boundary interface and the region lookup.

The inputs to the system that the user should supply are the light source position, the number of volume elements in the tissue, the minimum coordinates of the tissue in the x, y and z dimensions, and the number of elements in each dimension that the tissue span. The algorithm also should take as input a seed to the random number generator. Moreover, information about each volume element (material type and its optical properties) has to be written on a DDR memory so that all kernels read from it.

The simulation process starts by launching multiple photons with a random direction. Then the photons move through the stages saving the absorbed weight of each photon in the corresponding voxel position in a 1D-array. The simulation continuous to propagate the photons through the model and launch another one in place of it when it terminates.

In the initial specification we decided to support the main features of FullMonte [4]. These include using a 3D volume representation, recording the photon absorption allowing transmission & reflection of photons and refraction.

B. Hardware implementation

The initial implementation of the monte-carlo light simulator that was discussed in the proposal consisted of PCI express IP to communicate with the host PC, multiple HLS-based light simulating cores each connected to a set of BRAMs in one kernel, a DDR memory module to store the tissue volume, and a soft processor that would run all the kernels and gather the results to send them to the PC. For our initial implementation, we envisioned the application to be a high performance project taking the inputs from a user on a host PC and sending the final results back to the user's host PC. Given that, we chose to use the PCI express for communication between the host PC and the FPGA.

IV. Specification Evolution and Final Architectural Design

In this section we describe the evolution of our specifications and the final architectural design, which can be seen in Fig. 4. The Microblaze soft processor send the input data to all the kernels, writes multiple copies of the material properties and material indices of each voxel to BRAMs, and every two kernels share one copy of those BRAMs. There is also an absorption BRAM for each kernel, where results are stored. The Microblaze would eventually read those BRAMs to gather all the results in one place.

A. *Kernel*

This section discusses the HLS kernel implementation applied to achieve the results.

1) Multiple Kernels: Initially we wanted to initialize multiple simulation cores within one HLS-generated kernel. By generating a generic code that would use for loops to initialize multiple cores. where each kernel has multiple photons launched and all would simulate a certain amount of photons. Since each simulation core would be independent we hoped that they would be parallelized. However, the synthesis results showed that the cores would be running in sequence. Given that, We opted to initialize multiple kernels within the Vivado IP integrator.

To do so, we started with a dual-core design in which the two cores read the data from the same set of input BRAMs. The cores and the BRAMs and other IPs in Vivado needed to be connected through AXI-Interconnects. Due to the limitations in number of master and slave ports available in AXI-Interconnects and the delay overhead of using a series of AXI-Interconnects, first we tried to find minumum number of BRAM partitions that could be applied to our kernel, while still achieving the optimal II.

Also, we changed the access to memories from direct BRAM access when using only one core to AXI-protocol access so that multiple kernels can share the BRAMs. Although direct BRAM access is giving us higher performance, we needed to change the interface to BRAMs because otherwise we could not take advantage of dual core BRAMs that can be accessed by two kernels in parallel to minimize our BRAM utilization. Hence, as shown in Fig. 4, in our final design, both kernels are mapped to a set of dual-port input BRAMs. The case is different for the absorption result outputs. We have considered distinct set of absorption BRAMs for each kernel. This is due to the fact that access to the same set of BRAMs might cause race conditions between kernels and we might lose some data while writing the results into the absorption BRAMS. Therefore, the results are stored in separate BRAMs for each kernel. The accumulation will be eventually handled by the **MicroBlaze**. The results will then be stored in a final array in the soft-processor's BRAM.

Even with the mentioned consideration, the highest frequency achieved in dual-core was 100MHz, while we were able to achieve a speed of 167MHz with one HLS core. To increase the operation frequency, initially we took advantage of some the features of the AXI-interconnects to maximize their performance in order to have a better chance of meeting the target timing. Afterwards, we started locating the critical paths in Vivado and tried to decrease their latency by inserting register slices in interconnections. Ultimately, these steps led us to achieve an operational frequency of 150MHz.

The next step was to increase the number of cores. Our final implementation consists of **eight** cores. In this design, each two cores still are mapped to the same set of input BRAMs. However, the BRAM sets need to be replicated for each group of two kernels. Hence, eight cores need four replica of the input BRAMs.

2) Pipelining: To achieve a high performance core we proceeded to pipeline the core. By simply applying the pipeline directive on the loop did not help due to the dependencies between iterations. The main dependency was caused because if a photon reached a boundary. In which the HOP stage has to go through interface to check for reflection and refraction. Then the program had to loop back through the HOP stage to continue moving the photon the remaining distance. This was achieved by placing a while loop within the HOP stage. To avoid this dependency, we added a value to the photon data structure that gets incremented every time an interface happens, and does not allow the photon to go through spin and the drop stages unless the HOP stage has completely finished moving the photon (by setting a maximum number of interface steps allowed). This allowed us to get rid of the loop within the HOP stage and pipeline the main loop that simulates the launched photons. One can argue that the *Drop* and *Spin* modules can remain idle a lot of the time with this approach. However, the mean free path of the tissue that determines the step taken by the photon is usually much smaller than the voxel size. This means that the number of boundary crossings a photon passes through is much less than its interactions within the element, and thus, the idle time of the other modules is not significant.

Another dependency that hindered us from fully pipelining was that all the stages were accessing the same random number generator core. To avoid this dependency, we added a separate random number generator to every stage. Given that the *TinyMT* generator [12] was very small, adding multiple modules didn't add much to the area overhead.

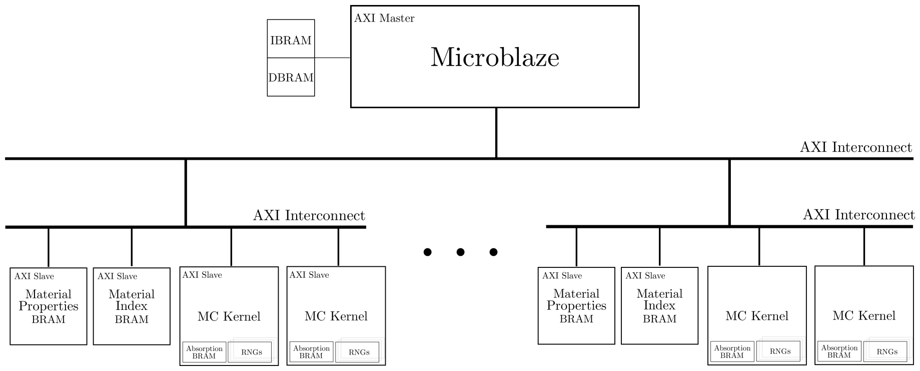

Fig. 4: Final Design Block Diagram

Once we were able to pipeline the loop and remove the dependency of the random generators, we were aiming to achieve an instantiation interval (II) of 1. To reach that II we tried multiple approaches. However, the most successful approach to decrease the II was to decrease the computations within the branching by precomputing the values and assign them within the branch. Given that the II decreased considerably from 194 to 6.

The II of 1 was not achieved because the algorithm requires that at every drop stage the photon has to drop part of its energy weight and deposit it in the corresponding volume element's position in the absorption array. To do that, you first have to load the already accumulated absorption value of that element. Then, we have to add the absorption caused by the photon and store the final value. This load, add and store operation causes a memory access conflict due to the read before store. Ideally, this can be avoided by doing a partial accumulation and buffer the absorption data into multiple buffers that can be accessed based on the photon index. However, the amount of unrolling needed to separate the buffers to access them in parallel was not possible given the data sizes we have and the randomness in the access pattern (multiple photons accessing same volume element due to randomness of the MC algorithm). Therefore, we opted to stay at II of 6.

3) Random number generator: The simulation algorithm requires a uniform random number generator. Due to the nature of the Monte-Carlo algorithm it requires a high quality random number generator as the results highly depend on the random number generator. To produce the quality of random numbers needed by the Monte-Carlo algorithm, we chose to implement the Tiny Mersenne Twister (tinyMT) random number generator in HLS. The tinyMT is a recent small-sized implementation of Mersenne Twister [12]. We choose to use the tinyMT for a random generator due to its small state size of 127 bits. With this state size, tinyMT's period of generated sequences is $2^{127} - 1$, which is good enough for our application.

4) Other optimizations: For our initial implementation, we wanted to use the HLS math operations for all the floating point operations. Despite that, we found out that the $\log$ operation was a bottleneck in our design. To avoid using the $\log$ operation, we decided to create a look-up-table to estimate the $\log$ results. By knowing that we only used the $\log$ operations on values between 0 and 1 in our application, we created a lookup table of size $8K$ for $\log$ values of inputs between 0 and 1, and indexed the table based on the input value.

Another optimization to our initial implementation was to compute and store the inverse values of certain constants. The computation of the inverse values takes place when the kernel receives the data from the external block RAMs. As a result we avoided many floating point division operations, which helped in achieving our optimal II.

B. System clock

In our initial design we were targeting to run our hardware at $200MHz$ clock rate, as was attempted in [7]. However, due to the large amount of mathematical operations needed by the algorithm, the critical path was big, and we were not able to reach this clock rate. With our II of 6, our hardware was only able to run at $100MHz$. However, by setting our target II to 8, HLS was able to reschedule things differently, and we were able to achieve a $6ns$ clock period, which translates to $167MHz$. This means that the overall gain in speed (from 100 to $167MHz$) is larger than the loss in latency

(II of 6 to 8), which lead to a decrease in the overall latency of the core.

C. *Microblaze*

In our initial implementation, we wanted to use the Microblaze to simply issue the start signal to the kernel. Afterwards, we found out that the Microblaze would be most suitable to populate the external block RAMs for the voxels' material indices and material properties. Furthermore, we used to initialize the kernel parameters send the start signal to the kernel, and to collect the results of the absorption from the simulation and convert it to fluence, which is the energy absorbed per square of the unit length.

D. *PCI express*

For our initial specifications, we added PCI express as we expected the kernel is going to need a stream of data to the host PC for initialization and to collect the results. Nonetheless, we found out that we could initialize the data needed by the kernel from the Microblaze and dump the results to the host PC by using the SDK. As the functions of the PCI could be achieved by the Microblaze we deemed the PCI to be unnecessary.

E. *DDR Memory*

Initially, we wanted to use the DDR4 memory to store the data for the model due to its large capacity as we expected that the data would be quite large for any model. However, we figured that due to the fixed-size voxel-based model we implemented, we only needed to store two values for each voxel. These values are the absorption at each voxel and the voxel's material index. Given that, we used block RAMs instead of DDR4 as they were sufficient to store the amount of data we needed for our model. This choice helped us in meeting the deadline of the project.

V. Methodology

The methodology followed by the team was inspired by the design flow in the Xilinx tutorials. This section describes the main design flow that was followed along with the environment, partitioning and verification of the design.

Initially, we implemented a basic design on HLS. To do so, we needed to perform mathematical equations in a way that would be synthesized in Vivado-HLS. We verified the results using RTL co-simulation available in Vivado-HLS to make sure the simulation results are consistent with the RTL results. It should be noted that the results of simulation were converted to VTK [11] files to be visualized and evaluated on Paraview [10]. After making sure the results are correct, we started optimizing that design using some Vivado-HLS directives and some code modifications that would serve the purpose in order to reach the best initial interval (II). Meanwhile, for each optimization, we evaluated the design with RTL co-simulation.

After this, we used the Vivado IP integrator to connect the kernel to the Microblaze and input BRAMs. In this step, we evaluated the design by using the behavioral simulation waveforms in Vivado and comparing the results to the simulation results in Vivado-HLS. Then, we ran *place-and-route* on the design, generated the bitstream and exported the hardware to test it on the FPGA. We evaluated the results by checking the absorption results in the memory window provided in the SDK system debugger.

Once the one kernel implementation was done and fully optimized and working, we implemented a dual-core design of the light simulator by integrating two kernels with the MicroBlaze and the same set of BRAMs. We evaluated the results of triggering the dual-core BRAMs using the results of Vivado behavioral simulations. Then, we increased the number of cores to eight by replicating the dual-core design. Finally, we exported the overall design to the FPGA and verified the result of final accumulation of absorption results. Given that we only used around 50% of the resources (will be shown in a later section), we could have increased the number of cores to 16, if time permitted, which would have doubled our speed-up.

A. *Design Environment*

Our design environment specifications are listed as follows:

- We used Xilinx Kintex Ultrascale as the target platform, with the chip, **XCKU115-2 - FLVA1517E**, as the FPGA.
- We used the Vivado tools to implement our light-simulator. The Vivado-HLS was used to implement our kernel. Vivado IP Integrator was then used to integrate the IP with a soft-processor (MicroBlaze) and BRAMs.
- We managed our data organization using **Bitbucket**, a distributer of `git` version control systems that gives private repositories for free for groups of up to 5 members. We also managed and discussed the project using **Slack**, an easy-to-use interface for project managements that allows users to chat and open different discussion channels for different parts of the project.

- We kept different repository branches for different users, so we could edit the same files. However, we made sure in our meetings and discussions that we worked on different parts of the code (functions, classes, ...etc) so that we could easily merge the branches at the end.

B. *Partitioning*

The kernel design was intrinsically divided into three main pipeline stages. Every stages was coded separately, and then with the simplicity of HLS, it was easy to combine all of the methods and steps together as can be done in a normal software design. We kept all the stages and computations in one core to minimize integration time in the IP integrator. We also believe that combining all the steps in one kernel helped us pipeline the MC algorithm properly.

C. *Simulation, Verification and Testing*

We developed a testbench for simulations in Vivado-HLS. For verifying the results, we converted the absorption results to the VTK [11] format to visualize them in Paraview [10]. We also tested and validated the results of the Vivado design by checking the waveforms of the RTL behavioral simulation. After moving the design to the board, and running it on the FPGA, we evaluated the results by checking the memory addresses of input BRAMs and absorption BRAMs, respectively. We used the SDK system debugger to read back the values of the absorption BRAMs. We also made sure that our results are consistent (same order of magnitude) with those of *FullMonte's* CPU implementation, even though our hardware design suffers from some floating point accuracy loss, and we use different random number generators. Fig. 5 shows a sample output of our results. It shows the output energy distribution after simulation over a cubic tissue of size $8K$ voxels. The optical properties of the different materials were taken from [13].

VI. Contributions

A. *Abdul-Amir*

Here are the contributions I have made throughout the course of this project:

- Since this project is related to my Ph.D. research background, I have a solid background on the literature of this topic.
- I have coded all the data structures and classes that we used in this project.
- I have coded most of the `C++` implementation of the MC algorithm and migrated the code to Vivado HLS.

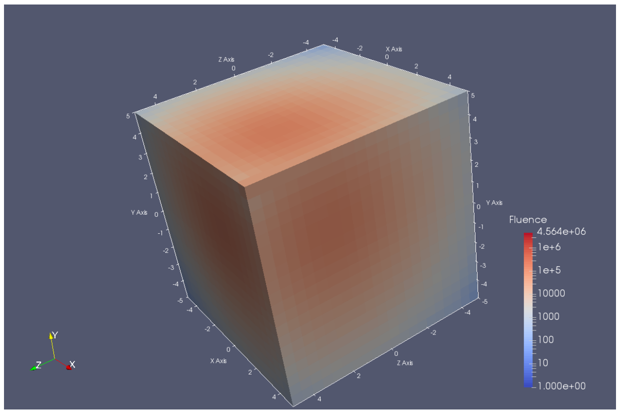

Fig. 5: Sample output of the simulator

- I got a `C++` implementation of the TinyMT random number generator from its official website [12], and migrated it to Vivado HLS.
- I helped Omar debugging and refining the `C++` code to produce valid results.
- I wrote scripts to generate VTK files of the output fluence in order to visualize the data in Paraview [10].
- I did most of the HLS optimizations and the little tricks that decreased the latency of the MC kernel a lot until we got an II of 8.
- I helped integrating the kernel with Microblaze.
- I moved, with the help of Omar, the design to the board.
- I helped Omar debugging the hardware to make sure the results are consistent with the C-simulation.
- I contributed to preparing the final report and presentation.

B. *Omar*

Here are the contributions I have made throughout the course of this project:

- Since I had no background in the project I had to do intensive literature review.
- I have coded the initial implementation of the drop and spin in `C++` of the MC algorithm and migrated the code to Vivado HLS.
- I made multiple scripts that generated multiple data for multiple volumes for debugging purposes.
- I debugged and refined the `C++` code to produce valid results.
- I worked on making sure that the RTL simulation results are consistent with the C-simulation.
- I restructured the code to make the optimization easier.

- I applied modifications to the code to decrease the data saved. That decreased the BRAM utilization.
- I helped in integrating the kernel with Microblaze.
- I helped in moving the design to the board.
- I wrote scripts to convert the dumped results from the FPGA into VTK format with the help of Abed's initial VTK script.
- I debugged the hardware to make sure the results are consistent with the C-simulation.
- I contributed to preparing the final report and presentation.

C. *Yasmin*

Here are the contributions I have made throughout the course of this project:

- Since I am going to work on a project related to the subject of this project, I had some literature review on the topic. However, I did some more detailed studies before I could start the project.
- I revised the codes for the *Spin* stage in our pipelines based on the available methods of implementing the mathematical algorithms of this stage in literature in order to find the one with the best II and least latency.
- I debugged the *Spin* part to evaluate the results of the different ways of its implementation.
- I Integrated the kernel IP with the MicroBlaze and other IPs in Vivado.
- I verified the results in Vivado simulations to make sure they are consistent with the results of HLS simulations.
- I implemented a dual-core design in Vivado IP integrator and verified the results using behavioral simulations in Vivado.
- I worked on timing issues in Vivado integration for the dual-core system to make the timings meet our target frequency that we had with a single-core implementation.
- I extended the design to an eight core implementation.
- I moved the final design to the board and verified the results of our available datasets.
- I contributed in preparing the final report and presentation.

VII. Design Characteristics

A. *Resource Utilization*

The resource utilization after placing and routing of our 8-core final design with a maximum data size of $8K$ volume elements as reported by Vivado is shown in the Table I. From the table, we can see that the limiting factor in our design is the BRAMs with 55% utilization. This means that we can add more cores to the design and speed up the simulator more. We can also increase the data size to simulate bigger tissues. In fact, one core with $8K$-elements tissue was taking only 12% of the BRAMs, which means we could have increased the data size 8 times more while still fitting on the FPGA. If we want to simulate tissues larger than 64K, we need to integrate a DDR block with the design. It is worth mentioning also that our 8-core design can simulate 1 million photons, with $125K$ running on each core. To simulate more than that, we can control the Microblaze processor to re-run some of the cores with different seeds.

Table I: Resource Utilization of an 8-core MC light simulator

Resource	Available	Utilization	Utilization (%)
`LUTs`	663360	299586	45
`LUTRAMs`	293760	20060	7
`Registers`	1326720	328683	25
`BRAMs`	2160	1206	55
`DSPs`	5520	731	13

From Vivado, we also found out that the total on-chip power consumption of our design is $11.2W$, 87% of which $(9.714W)$ is due to dynamic power and the remaining 13% $(1.5W)$ is due to the static power. Among the utilized resources, the BRAMs are responsible for a significant part of the dynamic power.

B. *Results*

We tested our eight-cores hardware on a tissue of size 8000 elements by launching 1 million photons. We also ran the sequential implementation in software on an Intel Xeon E5-1620 CPU running at $3.5GHz$. Our hardware runtime was at **5 seconds**, while our CPU implementation took **14.5 minutes**. This translates to **174x** speedup. We also ran against *Fullmonte*, the fastest software simulation to date [4], and we got a speedup of **3x**.

C. *Where the Time Went*

A breakdown of the main tasks and the time spent on them as a percentage is shown in Table II. The effective total amount of time spent on the project is about a span of 3-months person.

VIII. Problems

A. *Vivado IP integrator*

The Vivado tool could have been more efficient in the progress of our project if some of these problems did not exist.

Table II: Breakdown of tasks as percentage of time

Task	Time (%)
Literature Review	5%
Generating data	4%
Software Implementation and Testing	15%
HLS Optimization	25%
Integration and Validation on HW	25%
Implementing multi-core design	20%
Report and Presentation	6%

- The address editor has many bugs. For example, the auto-addressing sometimes assigns addresses to slaves that are not even valid in the address space seen by the soft processor. Also, it fails when there are a great number of slaves which was the case when we started designing the multi-core and it made the tasks too time-consuming since we needed to manually address each slave.
- At some points, it was convenient if we could copy some parts of the design and replicate it for the other cores. However, the replicating in Vivado did not happen exactly the way the IPs are placed and it made a mess, which forced us sometimes to repeat placing the resources and do the wirings.
- There were some limitations in the number of slaves and masters available in an AXI-Interconnect. In our multi-core design, we needed more slave and master ports than available. Therefore, we looked for IPs that could cascade AXI-interconnects. We found an IP called `axi2axi_connector` which is a utility module for use in the `Xilinx` Embedded Development Kit (EDK) to cascade two AXI Interconnect modules. Although, datasheets are available for this IP, the IP seems not to be available in Vivado which was weird. Therefore, to solve this issue, we tried minimizing the ports needed as much as we could (by decreasing the array partitions) and connecting AXI-Interconnects in a way that one acts as a slave to the other. In this way, we were able to handle the whole connections. However, if the design was much bigger, we might have not been able to solve the problem in this way.
- At some points, especially when trying to meet timing target with multiple cores, we had to re-synthesize and implement every time we changed something in the design, and we had to deal with a very long turn-around time to get the results. Every run took about 3 to 4 hours to finish. This was the case even when running on the `savi` server that we were given access to, which had a 24-core CPU.

B. *Hardware Testing*

One problem that we had to deal with when testing on the hardware was finding a way to initialize the BRAMs with the tissue properties. After several trials, we decided to use the MicroBlaze for initializing the BRAMs. In this case, we had to pay for memory usage by increasing the instruction and data memories of Microblaze, and we had to copy the data to the SDK code for initialization.

IX. Retrospective, Conclusion, Suggestions, Comments

A. *Experience with HLS*

The most noticeable feature of Vivado HLS is the simplicity it provides for changing some functional parts of the design in a very short time, while keeping the other parts optimized. This allowed us to try many design choices and see their effects on the overall latency. Had we coded the design in HDL, it would have taken us months to take such design choices and make sure the overall design is not affected by those choices.

However, one of the things that we had to suffer with HLS is that sometimes we had to multiply a float number by 2 for example, and HLS didn't optimize for that and spent 4 cycles on that operation. To overcome this issue, we used a *union trick*, where we used a `union struct` in C++ consisting of a `float` and an `int`. We set the `float` part to the number we had, and then reading the `int` would give us the bit representation of the number in **IEEE754** single-precision format. By adding 0x00800000 to the int, the float would be multiplied by 2. This would only take one cycle. Similar to this, the random number generator gave numbers between 1 and 2, and to get the number between 0 and 1, we had to subtract 1 every time, which would take 8 cycles. With the *union trick* and some bit manipulations, we were easily able to do this in 1 cycle. In HDL, we would have easily done the bit manipulations as the wires/registers are defined as bit-words.

Another problem with HLS is that some times it could not detect that some sections were independent to schedule them in parallel. For example, we wanted to run multiple kernels from inside HLS by instantiating multiple calls to the *inlined* kernel with different independent inputs, however, the HLS scheduler was always running them sequentially and adding the total latency. Therefore, we had to suffer in the IP integrator to connect a lot of kernels to their inputs and communication interconnects to make sure they ran in parallel.

Overall, HLS is definitely the way to go for designing hardware modules, as it provides easy and simple way to build such complex designs without the need to know the details about the architecture under hand.

B. Suggestions and Comments

Overall, we feel that this course was worth all the time we put in. We learned a new design methodology with HLS that helped us implement such a complex design in a period of five weeks, which would have taken us months to do in a traditional low level HDL.

The format of the course was excellent. The assignments at the beginning of the course helped bringing us up to speed with this design methodology and learning the HLS and Vivado Tools. However, we believe that the relative simplicity of the *Kmeans* algorithm limited our exploration of the tools and what they can perform, which made us spend some significant amount of time during the project researching on the different and large variety of optimizations that can be made with the tools. We suggest giving a more difficult design as an assignment or leave it for choice to make the students read more about what HLS can do, and what limitations it has.

The course has been a great learning experience to us on a technical and non-technical level. We would like to thank Professor Chow for his help and fruitful suggestions that allowed us to achieve what we intended to do with this project. We would also like to thank Naif Tarafdar and Charles Lo for the immense effort they put in answering all of our questions, and ensuring an easy and simple way to access the resources.

References

[1] R. T. Sadikot and T. S. Blackwell, "Bioluminescence imaging," *Proceedings of the American Thoracic Society*, vol. 2, no. 6, pp. 537–540, 2005, pMID: 16352761.

[2] B. C. Wilson and M. S. Patterson, "The physics, biophysics and technology of photodynamic therapy," *Physics in medicine and biology*, vol. 53, no. 9, p. R61, 2008.

[3] M. Mengüç and R. Viskanta, "Radiative transfer in three-dimensional rectangular enclosures containing inhomogeneous, anisotropically scattering media," *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 33, no. 6, pp. 533–549, 1985.

[4] J. Cassidy, L. Lilge, and V. Betz, "Fullmonte: a framework for high-performance monte carlo simulation of light through turbid media with complex geometry," in *SPIE BiOS*. International Society for Optics and Photonics, 2013, pp. 85 920H–85 920H.

[5] L. Wang, S. L. Jacques, and L. Zheng, "Mcmlmonte carlo modeling of light transport in multi-layered tissues," *Computer methods and programs in biomedicine*, vol. 47, no. 2, pp. 131–146, 1995.

[6] N. Ren, J. Liang, X. Qu, J. Li, B. Lu, and J. Tian, "Gpu-based monte carlo simulation for light propagation in complex heterogeneous tissues," *Optics express*, vol. 18, no. 7, pp. 6811–6823, 2010.

[7] J. Cassidy, L. Lilge, and V. Betz, "Fast, power-efficient biophotonic simulations for cancer treatment using fpgas," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE, 2014, pp. 133–140.

[8] J. Luu, K. Redmond, W. Lo, P. Chow, L. Lilge, and J. Rose, "Fpga-based monte carlo computation of light absorption for photodynamic cancer therapy," in *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*. IEEE, 2009, pp. 157–164.

[9] S.-H. Hung, M.-Y. Tsai, B.-Y. Huang, and C.-H. Tu, "A platform-oblivious approach for heterogeneous computing: A case study with monte carlo-based simulation for medical applications," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 42–47.

[10] U. Ayachit, "The paraview guide: a parallel visualization application," 2015.

[11] W. J. Schroeder, B. Lorensen, and K. Martin, *The visualization toolkit: an object-oriented approach to 3D graphics*. Kitware, 2004.

[12] M. Saito and M. M. Matsumoto, "Tiny mersenne twister (tinymt): a small-sized variant of mersenne twister," *Cited on*, p. 44, 2011.

[13] H. Shen and G. Wang, "A tetrahedron-based inhomogeneous monte carlo optical simulator," *Physics in medicine and biology*, vol. 55, no. 4, p. 947, 2010.