ECE1378: Final Project Report

Jin Hee Kim, Lawrence Park, Sadegh Yazdanshenas

1. Introduction

Monte Carlo method is a technique for repeatedly performing random sampling from a probability density function to obtain a numerical result. It is used widely in many applications such as financial asset modelling and molecular dynamics simulations. In particular, Monte Carlo method applied to simulation of light propagation on multi-layered tissue has promising applications. For example, Photodynamic Therapy (PDT) [1] is a minimally-intrusive cancer treatment that uses light-sensitive drug to selectively kill cells. In certain treatment, it is crucial to control the fluence (light dose) delivered since the target volume could be near organs. Monte Carlo model of light propagation can provide an accurate method to compute the fluence distribution. This involves simulating millions of photons rather than applying physics laws on the large scale. It is clear that this is a very large and computationally intensive task. Therefore, there is a need to accelerate the simulation as suggested by previous literature [2].

This project focuses on implementing MC simulator (MCML) using vivado HLS. Each photon interacts with the layers independently, which means that there is significant data parallelism. We changed the high-level structure of the code so that it can be pipelined as a hardware implementation. These traits of the application make FPGAs an attractive option for implementation.

MCML algorithm can be divided into 4 steps: launch, hop, drop, and spin as shown in Fig.2. There are several key features included in existing implementations of MCML algorithms which are based on different physical behaviours of light and tissue. A single photon moves into the tissue until it is either fully absorbed or it is reflected or transmitted to another layer as shown in Fig. 1. We implement a simulator based on planar geometry, absorption recording, and reflection and diffusion recording. This implementation also supports glass layers and can model tissues that are as complex as 125 layers. Other features such as non-planar geometries, non-scattering voids, and time-resolved data can also be added to our code but they require major modifications and were not included as our goal.

Our implementation is capable of achieving 103.5x speedup compared with the baseline C code that supports exactly the same features. Both implementations were run on University of Toronto UG machines. Previous GPU implementation for MCML which is known as CUDAMCML was capable of achieving about 80x speedup without supporting glass-layer. Our implementation beats CUDAMCML by achieving a speed-up for 138x when glass layer feature is disabled.

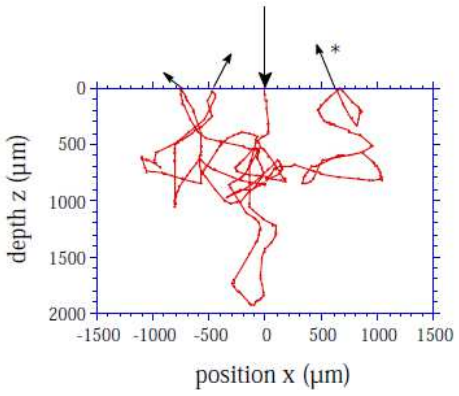

Figure 1. An Example of Photon Movement [3]

The rest of this report is organized as follows. Section 2 goes over the current status of our project and lists our contributions. Section 3 review our initial architecture design. Section 4 specifies the evolution of our architecture throughout the course of this project. Section 5 reviews our methodology for this project. Section 6 lists individual member contributions. Section 7 describes design characteristics. Section 8 describes the problems we faced when implementing MCML. Finally, section 9 concludes this report.

2. Current Status

We managed to finish all promised parts of the project before the deadline. Our contributions are listed below:

- We implemented and verified a working version of MCML that includes all of the basic features on Vivado HLS.
- We changed the high-level algorithm in MCML to allow hardware pipelining.
- We optimized the MCML kernel so that our implementation beats all existing implementations on FPGAs and GPUs in terms of performance.
- We analysed the effect of removing features from MCML on area and performance using HLS.

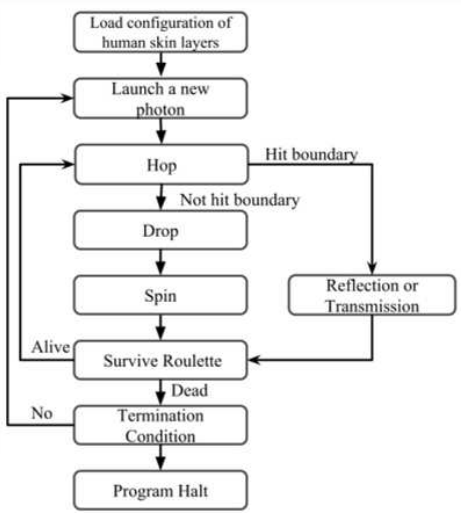

Figure 2. Basic steps in MCML [3]

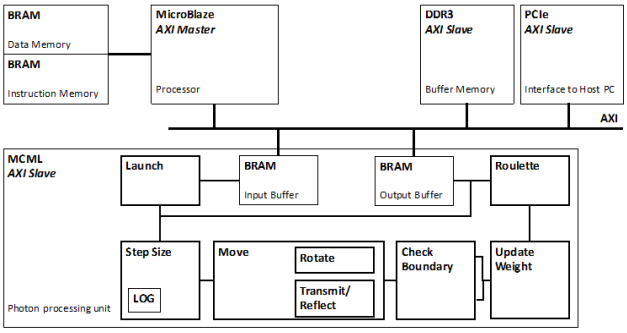

Figure 3. Initial Block Diagram

- We migrated the design on the NetFPGA-SUME board. This board can produce the required output which is then visualized by gnuplot in any target PC.

3. Initial Architecture Design

In this section, we describe our initial specification of our functional and hardware implementation of MCML. The block diagram from our proposal can be seen in Fig. 3.

3.1. Functional Specifications

The functional specifications of the project are provided by the ANSI C implementation of MCML algorithm presented by Wang [4] et al. The MCML algorithm requires information on the number of layers, layer specifications, the size of the tissue to be recorded (detection grid), and number of photons as an input. These are constant values and is used as a reference for the MCML algorithm. For the simulation to start, photon's initial position and direction is also required for each photon. Then the algorithm simulates propagation of photons through multi-layered tissues, and scores the photon absorption in a 2D array. The algorithm also records the photons that are reflected and transmitted in 1D arrays.

As our initial specification, we decided to support all of the features included in MCML. This includes using planar geometry, recording the photon absorption, allowing transmission & reflection of photons, and partial reflection. In addition, we wanted to study the effects of number of layers given that in many previous FPGA studies, this was limited to a fixed number of layers.

The original algorithm initially launched one photon and waited for it to terminate. This meant that there was a while-loop within a for-loop; where the while-loop was repeated many times for one photon until termination and the for-loop indexed the current photon being processed.

3.2. Hardware Specifications

Initial hardware architecture of our MCML implementation consists of a PCIe endpoint, a HLS-generated MCML simulation core, and I/O peripherals. We initially envisioned this project as a high-performance computing application where a host PC provides simulation MCML inputs (i.e. incoming photon position and direction) to FPGA-accelerated MCML simulator, and read back simulation results as soon as they become available. Given the popularity and high I/O bandwidth of PCIe protocol, we consider using the Xilinx PCIe IP to facilitate data transfers between a host PC and the FPGA system.

We initially wanted to create one HLS-generated MCML simulation core that has multiple processing units within the core so it can launch multiple photons at once. This means that each processing unit can be assigned a new photon when the current photon dies. We decided on one core since to implement this with multiple cores, we have to keep track of the number of photons that have been processed. This requires a method for each kernel to communicate with each other. If we divide up the photons into the available cores, we avoid the communication problem; however, we may have some cores that remain idle. In terms of implementing the MCML algorithm, we wanted to use the available HLS implementation of complex math functions such as log, sqrt, and sin/cos.

4. Specification Evolution and Final Architectural Design

In this section, we describe how our design specification has evolved and describe our final architecture. Our final block diagram in Fig. 4 lists all of the

major components of our design. Microblaze provides the layer specifications by writing it to the BRAM. Afterwards, MCML simulation core reads, copies the data into its internal memory structure and carries out the simulation. Finally, it writes the output arrays back into block RAMs for SDK to read out.

4.1. Feature Support

In our initial specification, we wanted to support as many, if not all of the features in the MCML algorithm. Literature review of previous implementations revealed that some implementations opt out of certain features to increase performance. Therefore, we decided to add the option of adding or removing features from the original MCML with ease. This will allow user to only include the necessary pieces of the MCML simulator and gain a performance boost.

4.2. MCML Simulation Core

Our initial HLS-generated MCML simulation core was optimized in various ways. The following sections lists the change in specification and describes the final architecture.

4.2.1. Multiple Processing Unit. The initial focus was on creating parallel processing unit to be able to launch multiple photons. This was due to the fact that each photon is independent of each other and that we had enough resources to be able to spatially parallelize the processing units. To direct HLS tool to create multiple instances of the processing unit, we applied partial loop unrolling to the for-loop (iterating through the number of photons). Since each photon is independent, we hoped that the partial loop unrolling would create multiple copies of the processing unit. However, regardless of code changes we tried, this seemed to be not possible.

4.2.2. Pipelining. Instead of creating multiple instances of the processing unit, we proceeded to pipeline our processing unit. By applying just the pipeline directive, HLS tool did not actually pipeline the design since for each photon, the next iteration depends on its previous. Originally, we used a for-loop to keep track of number of photons being terminated and while-loop to wait for a photon to finish processing before launching a new one. To overcome this issue, we had to make changes to the code. Since we wanted to fill up the pipeline, we needed enough independent photons being processed so that the first photon completes one iteration before the next iteration. Meaning, if the depth of the pipeline is 100, if there are 100 photons in the pipeline, the next iteration, the first photon will have gone through the entire pipeline and now have the new coordinates and direction.

In the final algorithm, we first launch 1000 photons. Then we use for-loop to iterate through the 1000 photons applying the MCML algorithm. In each of these for-loop iterations, we check to see if the photon has terminated; if it has, we launch a new photon and continue with the pipeline. Finally, the simulation is finished when the number of terminated photons exceed the total number of photons. A pseudo-code example of the major algorithmic changes to support pipelining can be seen below:

```
//before:
for (int nPhoton = 0; nPhoton < nTotalPhotons;
    ↪ nPhotons++) {
 while (currentPhoton.status == alive) {
  //hop, drop, spin
 }
}

//after:
while (nPhotons < nTotalPhotons) {
 for (int i = 0; i < piplineDepth; i++) {
  //hop, drop, spin
  if (currentPhoton.status == dead) {
   // launch a new photon
   nPhotons++;
  }
 }
}
```

4.2.3. Memory Access Conflicts. Once we were able to pipeline, we wanted to achieve an initiation interval (II) of 1. This means that we will be able to fully utilize our pipeline and achieve the best performance possible from pipelining. However, due to memory access conflicts, the tool estimated an II of 5. The algorithm requires that at every drop stage, the absorption array (output array) is updated with the deposited weight by the photon. To update the absorption array, it requires us to read from the array, add the floating number to the array, and write to the array before the next photon tries to access the array. This is a read-after-write conflict. If there was an access pattern to the array, we could partition the array in such way that there will not be a read-after-write conflict. In MCML, the photons can travel in any direction randomly, hence no such pattern exist.

After analyzing the access patter, we knew that each access takes 5 cycles (read->add->write). So, we wanted to create 5 absorption arrays so that in each cycle, the photons can deposit in a rotating manner. If we number each absorption array, while absorption array #1 is being used, the next photon in the cycle can use absorption array #2 and while we use absorption array #5, absorption array #1 has become available again. We tried many code transformations to implement this behaviour; however, we found that describing a behaviour per cycle was impossible in HLS.

4.2.4. Random Number Generator. The MCML algorithm requires a uniform random number generator as

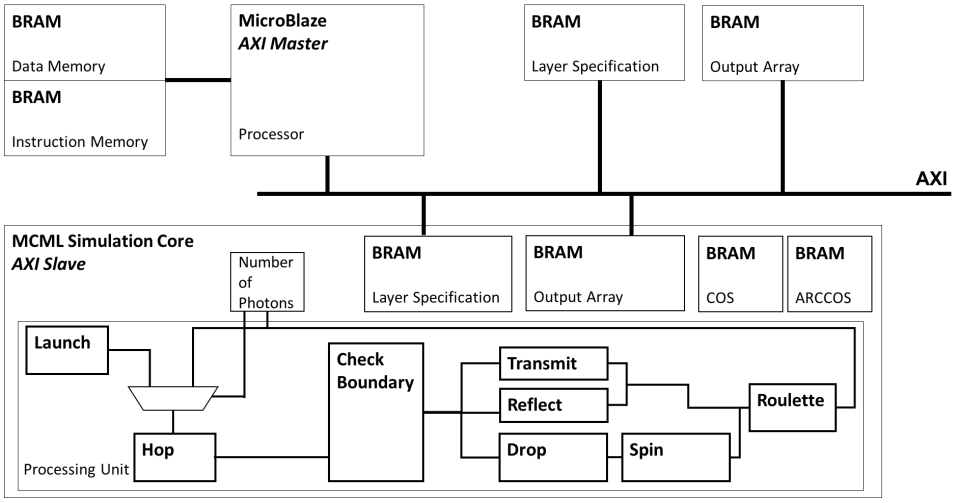

Figure 4. Final Block Diagram

part of the algorithm. Since this is a Monte Carlo simulation, the results of the simulation heavily depend on the quality of the random number generator. Our initial design implemented the random number generator using LFSR. Although LFSRs are simple, we needed a better quality random number generator. First we considered multiply with carry and complementary multiply with carry random number generators. However, these are compute intensive and are not suited for use with an FPGA. After some research, we decided to use the linear congruential generator (LCG). This was a fast generator that only requires minimal memory to retain its previous state.

4.2.5. Other Optimizations. In our initial design, we wanted to use the HLS implementation of the math functions used in the library. However, HLS tool did not support the arccosine function. This resulted in us to use look-up-tables to access the arccosine values. Since this proved to be more efficient than implementing the cosine function, we also converted our cosine function to use a look-up-table.

Another optimization to our initial design is to ask users to store inverse values of certain constant values. Since some values are used to only divide floating-point numbers, we decided to ask the user to simply pass the inverse value. It is possible to compute its inverse in the processor or beginning of the photons launching as well since it needs to be completed only one time. With this method, we avoided many float divisions.

4.3. PCIe and Data Transfer Requirement

Initially, we expected that the the MCML algorithm requires streaming data inputs from the host PC for incoming photons, but a simpler approach is to generate these information on hardware. This is not a compromise in functionality, as our reference implementation of MCML assumes a fixed position for incoming photons. Without the need to supply these inputs, we deemed that PCI Express is not a necessary component in our MCML design, and we opted for a simpler solution based on a JTAG-UART. This decision helped us to meet the project deadline since developing hardware and software support for transferring data over PCI Express require significant design effort.

4.4. System Clock

To achieve the best speed-up, it is often beneficial to run the MCML core at the fastest possible clock frequency, reported by Vivado HLS tool. However, clocking the MCML hardware at an optimal clock rate would require surrounding hardware to be the critical path in the overall design. Initially, we estimated that the MCML design's target frequency to be 200 MHz, but it was later discovered that the maximum frequency of our MCML core is 100 MHz which turned out to be the critical path of our overall design. Therefore, we changed the overall system clocking so that the design is fully synchronous with 100 MHz clock.

4.5. External Block RAMs

In our initial design, we planned to synthesize required MCML RAM resources, including its output arrays (e.g. deposition, reflection, and transmission density) within the HLS-generated MCML core, but we learned that Vivado HLS assumes the array parameters at the top level function to be inferred as a memory interface instead. This necessitates the creation of AXI-mapped BRAM resources on the MicroBlaze design, and AXI master ports on the MCML core. For performance perspective, we believe that it's better to have

infer BRAM resources within the MCML core because expect AXI4 interconnects to add additional delays for memory accesses which often causes performance bottlenecks. An alternative would have been to create our custom AXI wrapper module with HDL, and infer BRAMs within it, but we deemed that this does not justify the extra design effort.

5. Methodology

To the best of our knowledge, Xilinx tutorials offer the most comprehensive and bug-free methodology for Vivado HLS. Therefore, we used the same flow as in Xilinx tutorials.

5.1. Design Environment

We used the NetFPGA-SUME board which is powered by Xilinx's Virtex-7 XC7V690T FPGA. In order to get a reference design for testing output ,we used an existing C implementation of MCML. Two of us were working on HLS kernel optimization and we employed the following strategies so we could both edit project files:

- We used Github among ourselves for revision control.
- We met at least twice a day to make sure we are not editing or optimizing the same part of the project.
- We kept all the directives as pragmas inside the source code to avoid complications caused by additional directive files.

5.2. Partitioning

We decided to keep the software part simple so that we can maximize our exploration and optimization on hardware given the limited amount of time that we had. The software code that resides in Microblaze simply initializes the layer information and number of photons and calls the hardware kernel. The hardware kernel performs all the computations and stores the absorbed photon density, diffusion density and reflection density into the corresponding memory blocks so that software can read them back. We believe that this strategy is simple and efficient since all parts of the MCML kernel cane be pipelined properly.

5.3. Simulation, Verification, and Testing

The main method for verification is reading back block RAM information using the Xilinx SDK toolkit. We further developed two gnuplot scripts that allow us to visualize this information for simpler debugging.

In order to be able to visualize the output to speedup the process of debugging, we wrote two gnuplot scripts

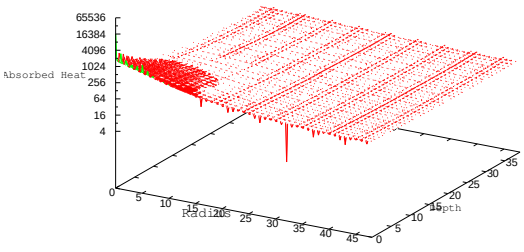

Figure 5. An example visual of photon distribution in a double-layered tissue generated by our gnuplot script

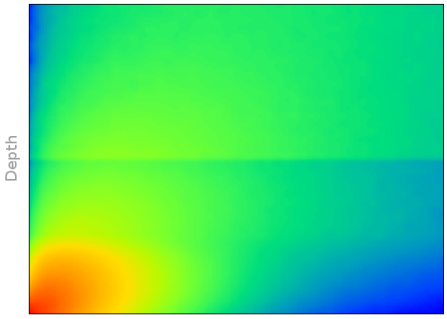

Figure 6. An example visual of heat density in a double-layered tissue generated by our gnuplot script

to show both distributions of absorbed photons in the modelled tissue and their heat density. We also made sure that these numbers don't differ from the CPU version by more than 5% despite having different random number generation systems. A sample visual for photon distribution is shown in Fig. 5 and a sample visual for heat density is shown in Fig. 6.

6. Contributions

Contributions for each member are listen in table 1.

7. Design Characteristics

7.1. Resource Utilization

The following table summarizes the logic utilization of our implemented design on a Virtex-7 XC7V690T FPGA.

As shown in the table, our FPGA implementation of MCML algorithm is LUT-constrained, and the LUT utilization by MCML simulation core (i.e. excluding MicroBlaze processor, AXI4 interconnects, and etc.) is 10.18%. With the remaining logic resources, further speed up of MCML algorithm is possible by replicating the MCML cores on the AXI4 bus. Assuming conservative chip utilization (i.e. 70%), implementing

TABLE 1. MEMBER CONTRIBUTIONS TO THE PROJECT

	Jin Hee	Sadegh	Lawrence
Literature review	✓	✓	✓
Finding previous implementations for reference	✓	✓	✓
Porting a C implementation into Vivado HLS	✓	✗	✗
Debugging and refining the C code to produce valid results	✓	✓	✗
Restructuring the code to make pipelining possible	✗	✓	✗
Writing gnuplot script to visualize output	✗	✓	✗
Optimizing HLS kernel	✓	✓	✗
Integrating the IP with Microblaze	✗	✗	✓
Moving the design to board	✗	✗	✓
Preparing final presentation	✓	✓	✓
Preparing final report	✓	✓	✓

TABLE 2. LOGIC UTILIZATION OF IMPLEMENTED SYSTEM

Resource	Utilization	Available	Utilization %
LUT	60424	433200	13.95
LUTRAM	5059	174200	2.90
FF	72895	866400	8.41
BRAM	120	1470	8.16
DSP	283	3600	7.86
IO	6	850	0.71
BUFG	4	32	12.50
MMCM	1	20	5.00

upto 5 MCML cores should be feasible for a Virtex-7 XC7V690T chip.

7.2. Where the Time Went

A detailed breakdown of how we spent our time on this project is shown in Fig. 7. The total effective time spend on the project was about 3-month person.

8. Problems

At the start of this project, we had a considerable hardware design experience using HLS, but we lacked experience doing the back-end design flow, including system integration, synthesis, and timing closure. Therefore, the significant challenges we experienced are related to completing physical implementation of the design.

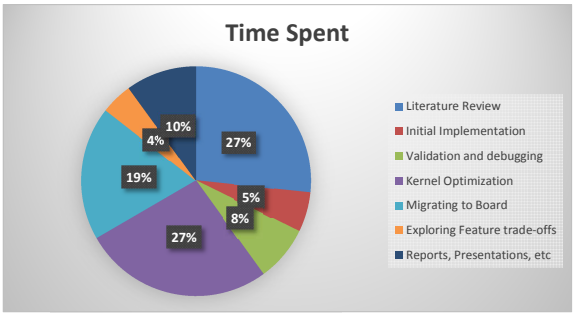

Figure 7. Breakdown of spent time over the course of this project

8.1. Timing Closure

Initially, we plan to clock our design at 100 MHz after HLS reported an estimated critical path delay of 9.53 nano-seconds for the 100 MHz performance constraint. However, we encountered a numerous paths failing setup timing constraints inside the HLS-generated blocks. We realized that the most of these paths contain long data paths (i.e. 30+ LUT levels) that results in excessive net delays which were the main culprit of our issue. It was disconcerting to witness that Vivado HLS can miss the delay estimates by such large margins. This suggests that HLS tool does not have a good net delay model, and we weren't able to rectify issues with HLS directives. Our final solution was to over-constrain the timing constraint (i.e. setting 3 ns for target clock period) , so that the tool is forced to insert FFs on data paths. We believe that this approach is a work-around for the obvious shortcoming of Vivado HLS tool.

8.2. Synthesis Run-time

Another major problem that we had to deal with is a long turn-around time to get synthesis results from Vivado. To achieve timing closure, we experimented various HLS performance constraints and Vivado synthesis strategies. To move forward with our experiments, we had to wait until a previous synthesis run is complete. A problem with this approach is it takes about 2.5 hours to complete a synthesis process on a decent workstation - a PC with Xeon E5-1620 @ 3.50GHz. We were surprised by the slow turn-around time since our logic utilization and performance target are relatively modest for the given FPGA (i.e. Virtex-7).

Due to the elongated physical design process, the embedded software development for our design was delayed a few days. We believe that this problem can

be mitigated by scheduling multiple synthesis runs and more experience with the back-end FPGA design flow.

9. Conclusion, Suggestions, and Comments

9.1. Experience with HLS

The most prominent feature of Vivado HLS is that it allows designers to easily modify functional features of an algorithm, and it keeps the generated hardware well-optimized. This allowed us to do a thorough exploration of features-performance trade-offs. If we had used HDL, this exploration would become the major bottleneck in our project schedule. With HLS, design exploration can be done much faster, and we completed this process in a few days. If we were to do this project using Verilog, we estimate that the same process would have taken at least few months.

The only problem that HLS caused us was that there was no way to force it to understand some parts of the code have no dependency. This was especially true for memory accesses. With RTL, it would have been possible to achieve an initiation interval of 1, we achieved an interval of 5 due to memory dependency.

HLS is definitely the way of the future. In overall, we are very impressed with its design productivity despite minor shortcomings with its tools. The CAD tool still needs work, including more hardware directives. However, we are confident that in near future, HLS will be a mainstream method for hardware design.

9.2. Suggestions and Comments

We feel that this course is one of the best graduate level courses we had in University of Toronto. We learned a lot about HLS design methodology, mastered using HLS tool over two assignments, and carried out a comprehensive design project which resulted in a working prototype of a complex algorithm. The hands-on learning approach taken in this course is the reason why we enjoyed it so much. If we were to take this course again, we would do most of things in the exact same order. However, we would prefer to spend more time to learn about system integration and SDK earlier in the course.

We also learned a lot about systems design in lectures. The course covered a great deal of information on clock domains, synchronizers, microprocessor programming, FPGA architecture, etc which would give a kick-start to any computer engineering student even without proper hardware design background.

The format of the course was excellent. We, however, have several suggestions that think can make the course even better in the future:

- Make the course inter-disciplinary. The department has already done this for other courses, such as mobile application course by Professor Rose. We believe that having students without engineering background, but with interesting project ideas from other departments can yield better project ideas, and improve students' motivations for the final project. These students can have presentations on several topics of their interests instead of doing the first two assignments.
- If there was a target goal for assignment 1 (we suggest a very hard to achieve target), students would be forced to expose themselves to a larger variety of directives and therefore learn a lot more about HLS in an earlier stage. This could also eliminate the need for assignment 2, giving an earlier start to the project.
- We suggest involving students with SDK and integration tools earlier in the semester. Don't leave this part to the project as it is an interesting part of the project and all students can benefit from having an individual exercise in this part.

This course had been a great learning experience. We would like to thank professor Paul Chow, Vincent Mirian, and Charles Lo for all their help throughout the course.

References

[1] 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.

[2] 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.

[3] L. Wang and S. L. Jacques, "Monte carlo modeling of light transport in multi-layered tissues in standard c," *The University of Texas, MD Anderson Cancer Center, Houston*, 1992.

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