A Monte-Carlo Simulator For Light Propagation in Turbid Media using HLS

Abdul-Amir Yassine

Omar Ismail

Yasmin Afsharnejad

Outline

- Motivation
- Algorithm Overview
- Design Overview
- Implementation Challenges
  - Pipelining
  - Float Tricks and Other Optimizations
  - Integration
- Verification and Testing Procedure
- Results and Resource Utilization
- Experience with HLS

Outline

- Motivation
- Algorithm Overview
- Design Overview
- Implementation Challenges
  - Pipelining
  - Float Tricks and Other Optimizations
  - Integration
- Verification and Testing Procedure
- Results and Resource Utilization
- Experience with HLS

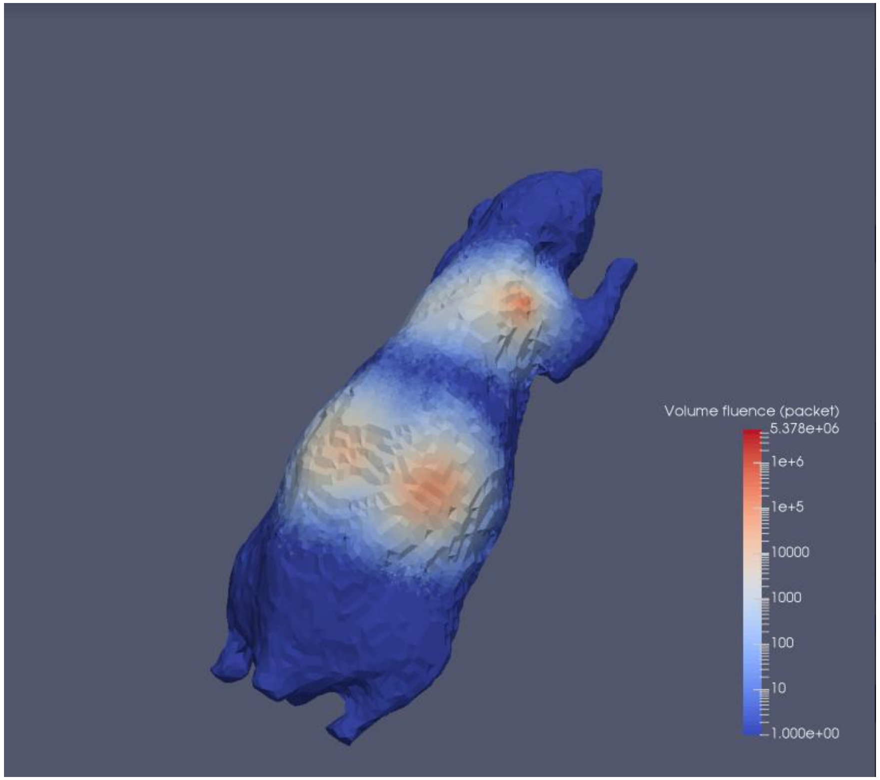
Motivation

- Emerging clinical applications require fast and accurate modelling of light propagation in turbid media.
  - Bioluminescence Imaging (BLI).
  - Photodynamic Therapy (PDT).

- Radiative transfer equation (RTE) is computationally expensive.
- Monte Carlo simulations can model the problem accurately, but they are slow!
- High-level synthesis facilitated the acceleration of such methods in recent years.

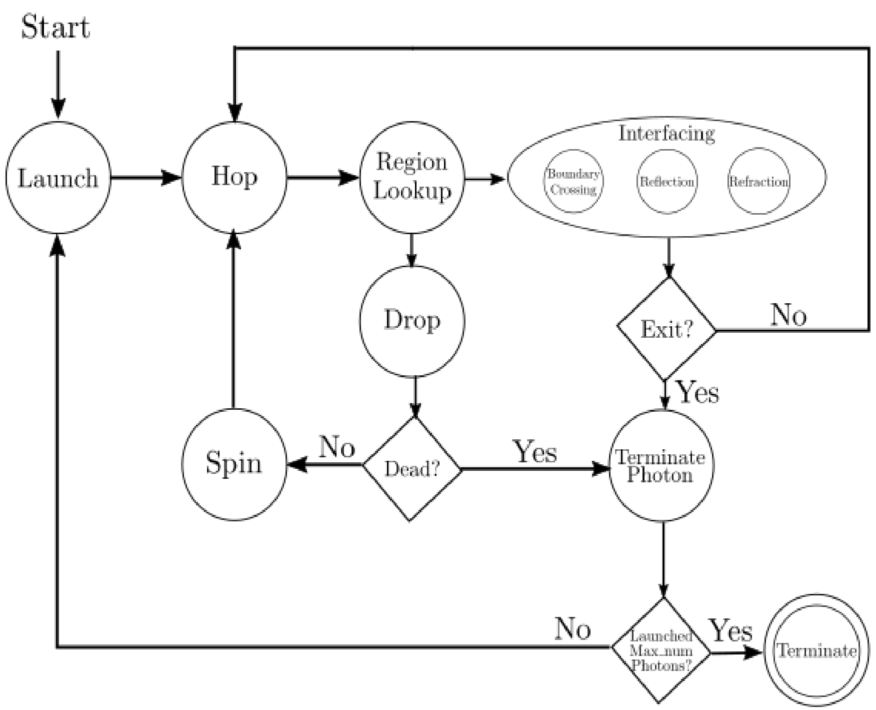
Problem Definition

- Given:
  - Geometry specification (3D tissues with a voxelized representation).
  - Material optical properties.
  - Light source position. (Point Light Sources)
- Output:
  - Light absorption in voxel elements.
  - Trace of light in the tissue.



Algorithm Stages

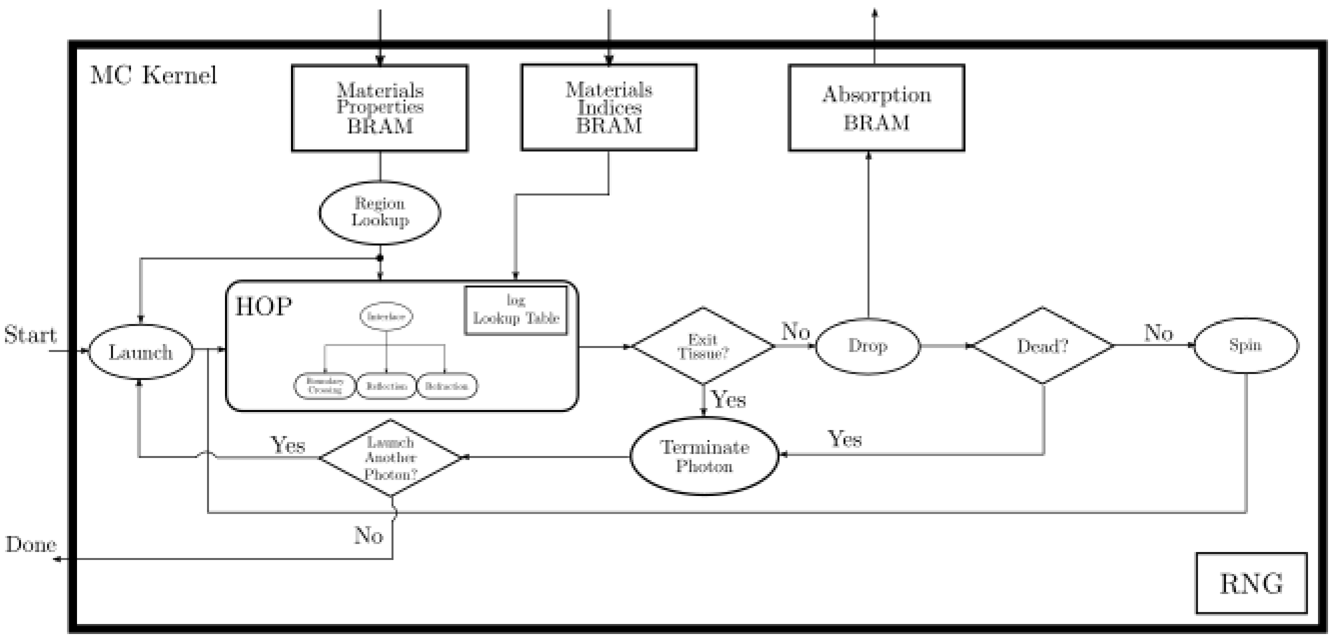
- Launch: Generate a packet of photons with a random direction from the source position.
- Hop: Moves the simulated photon from current position to a new position.
- Drop: Computes the photon's weight to simulate the energy absorbed by the tissue based on the absorption coefficient.
- Roulette: Checks if the photon packet is dead.
- Spin: Deriving the scattering angle for the next hop.



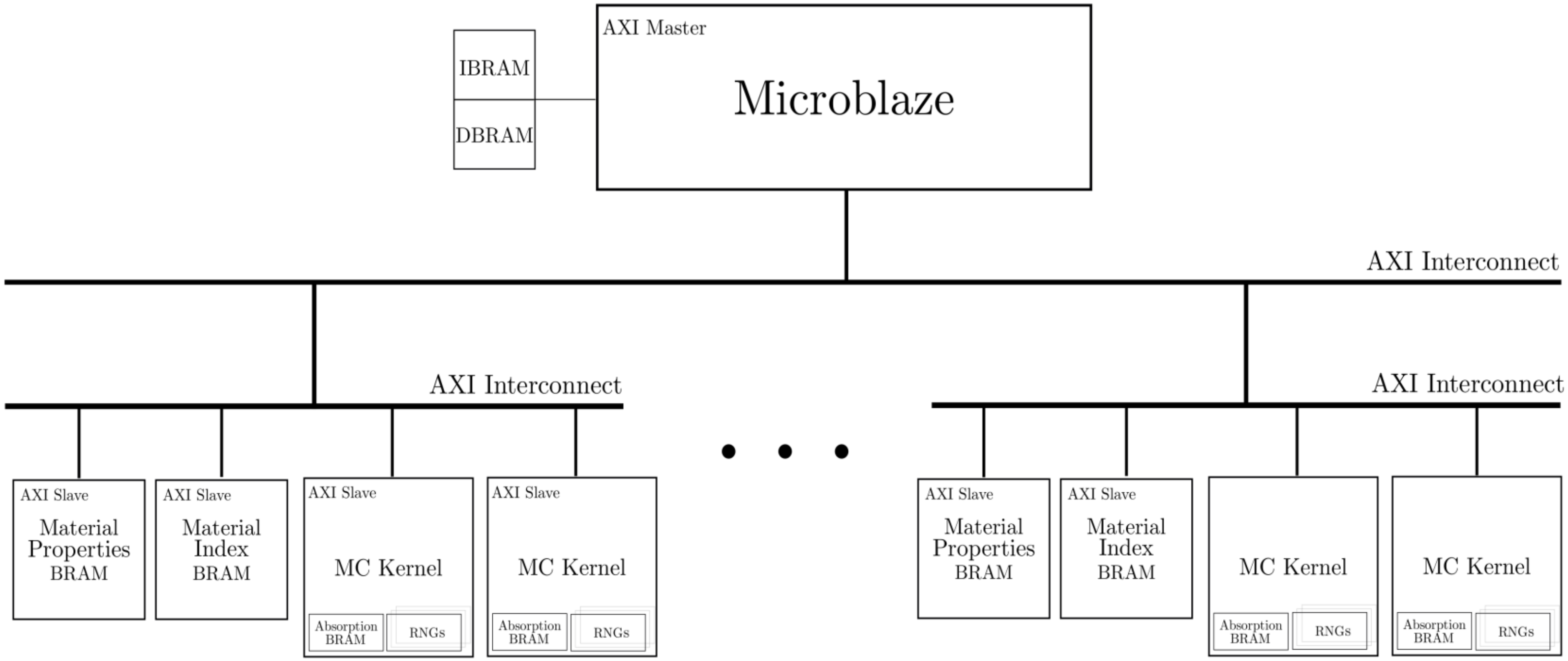
Outline

- Motivation
- Algorithm Overview
- Design Overview
- Implementation Challenges
  - Pipelining
  - Float Tricks and Other Optimizations
  - Integration
- Verification and Testing Procedure
- Results and Resource Utilization
- Experience with HLS

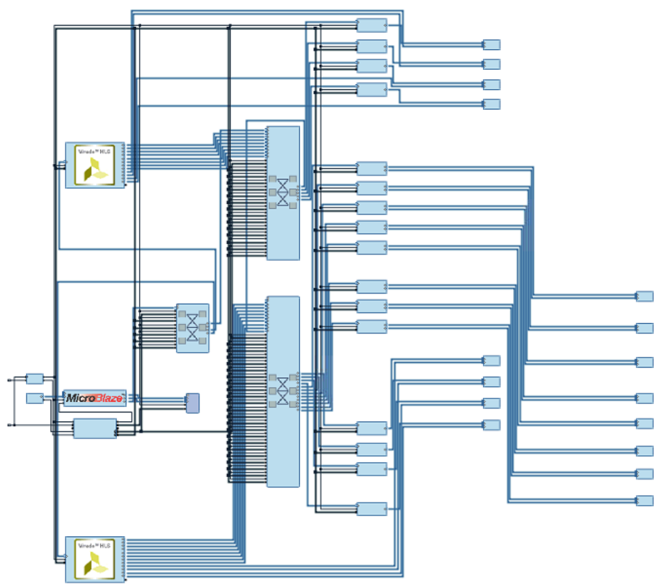
FSM

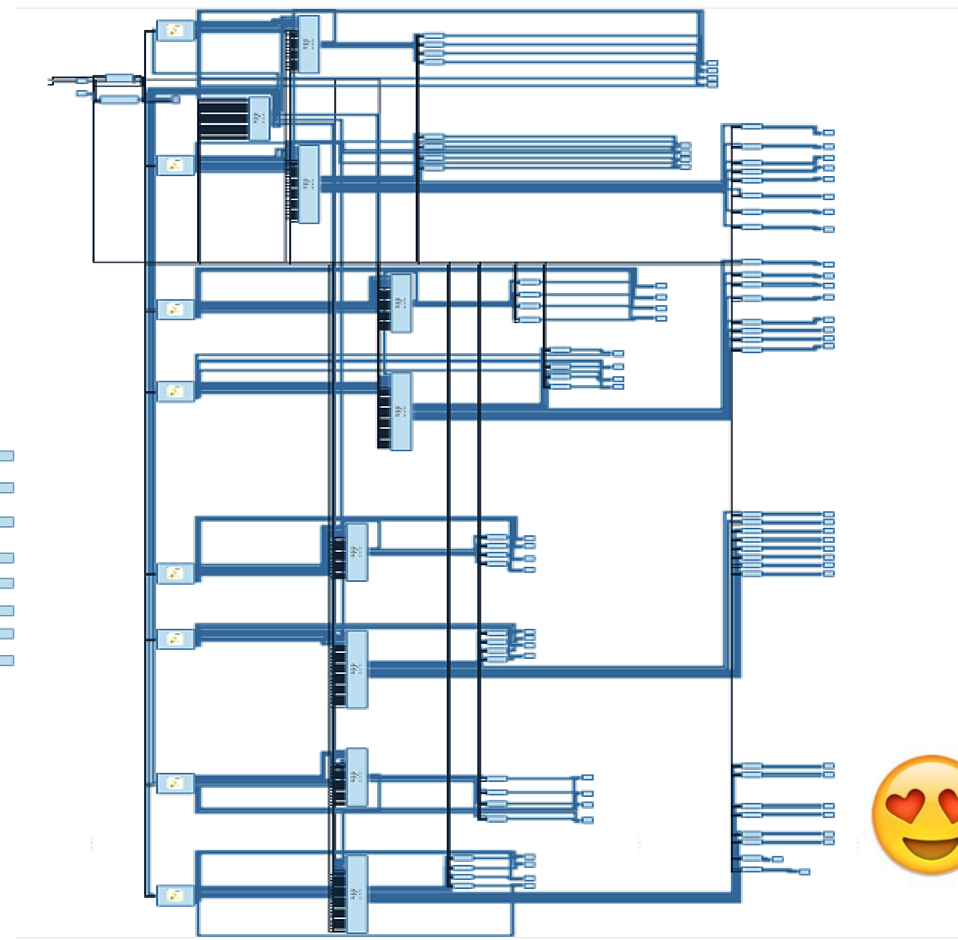

MC Kernel
Materials Properties BRAM
Materials Indices BRAM
Absorption BRAM
Region Lookup
HOP
log Lookup Table
Start
Launch
Exit Tissue?
No
Drop
Dead?
No
Spin
Yes
Terminate Photon
Yes
Yes
Launch Another Photon?
No
Done
RNG

Final Design



Big Picture!





RNG

- TinyMT:
  - A smaller implementation of Mersenne Twister[1].
  - State is 127bits long ➔ Period is $2^{127} - 1$.
  - Only bitwise operations ➔ easily implemented on FPGAs and fast.

```
    void tinymt32_next_state()
    {
#pragma HLS INLINE

        unsigned int x;
        unsigned int y;

        y = status[3];
        x = (status[0] & (unsigned int)MY_TINYMT32_MASK)
        ^ status[1]
        ^ status[2];
        x ^= (x << (unsigned int)MY_TINYMT32_SH0);
        y ^= (y >> (unsigned int)MY_TINYMT32_SH0) ^ x;

        status[0] = status[1];
        status[1] = status[2] ^ (-((int)(y & 1)) & mat1);
        status[2] = x ^ (y << (unsigned int)MY_TINYMT32_SH1)
                ^ (-((int)(y & 1)) & mat2);
        status[3] = y;
    }
```

[1] Matsumoto, M. and Nishimura, T., 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation (TOMACS), 8(1), pp.3-30.

Outline

- Motivation
- Algorithm Overview
- Design Overview
- Implementation Challenges
  - Pipelining
  - Float Tricks and Other Optimizations
  - Integration
- Verification and Testing Procedure
- Results and Resource Utilization
- Experience with HLS

Pipelining Challenges

- Each photon packet runs independently from the other.
- Each photon packet depends on its previous iteration.
- Ideally, datapath can be pipelined . 🙂

- Problems: 😩
  - Boundary crossing.
  - RNG dependency.
  - Absorption Accumulation.

Pipelining Challenges (Cont'd)

- Boundary Crossing:
  - Every time a photon crosses a boundary, it goes through HOP again to jump remaining step.
  - Problem: Pipeline has to stall as following photons cannot enter the Drop stage until the current photon does.
  - Solution: We removed HOP while loop and prevented photon from going to the Drop stage unless a maximum number of crossings, or no crossing occurs.

```
        PHOTON_LOOP1: for ( int i = 0; i < NUM_PACKETS_LAUNCH; i++)
        {
#pragma HLS PIPELINE II=8

            ph = sim1->launch(i);

            bool res = sim1->HOP( i );
            if (res)
            {
                sim1->terminate();
                sim1->photons[i] = ph;
                iter_idx1++;
            }
            if (sim1->get_photon_i_num_hits(i) >= MAX_NUM_HITS_HOP)
            {
                sim1->set_photon_i_remaining_step(i , 0);
                sim1->SPIN(i);
                if (sim1->DROP(i)){
                    sim1->photons[i] = ph;
                    iter_idx1++;
                }
                sim1->set_photon_i_num_of_hits(i, 0);
            }
        }
    }
```

Pipelining Challenges (Cont'd)

- Random Number Generator Dependency:
  - Every stage needs an RNG.
  - Problem: Different photons in different stages of the pipeline access the RNG.
  - Solution: Assign every stage a different RNG with different seed.

- Absorption Accumulation:
  - Photon's partial energy weight should be deposited in tissue.
  - Problem: RAW dependence:
    - Need to read the current value, accumulate then store before other photons can do the same.
    - Takes 6 cycles.
  - Solution: Partial accumulation.
    - Accumulate in different buffers then add at the end.
  - Problem: Large BRAM utilization due to large data sizes.

Float Tricks

```
typedef union {
    float_used f;
    int ui;
} ftoi_union;
```

- Multiplication/Division by 2:

```
// generating random initial direction
ftoi_union rand_z_1, rand_pi_2;
rand_z_1.f = _random_generator.tinymt32_generate_float();
rand_pi_2.f = PI*_random_generator.tinymt32_generate_float();

// Multiplying the random number by 2
rand_z_1.ui += NUM_TO_ADD_UNION;
rand_pi_2.ui += NUM_TO_ADD_UNION;
```

- Subtraction of 1 from a number between 1 and 2.
- Float Comparison.
- Division by constant: store inverse at the beginning and multiply.

Other Optimization

- Log was on critical path.
  - Input values are only between 0 and 1.
  - Approximate by a large lookup table that is distributed uniformly.

- HLS didn't detect similar or independent code in different branches.
  - Move code outside of branches to reduce branch divergence.
  - Example:

```
sim1->set_photon_i_remaining_step(i , 0);
sim1->SPIN(i);
if (sim1->DROP(i)){
    sim1->photons[i] = ph;
    iter_idx1++;
}
sim1->set_photon_i_num_of_hits(i, 0);
```

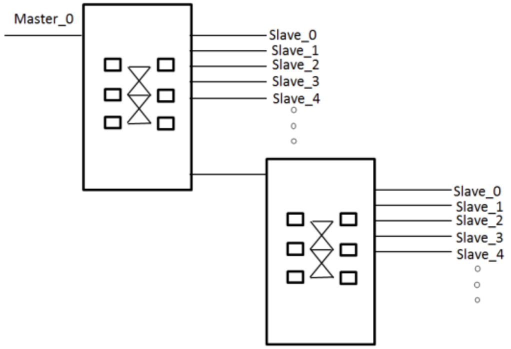
Final Result

- All features are supported.
  - Boundary Crossing.
  - Reflection (internal and Fresnel’s).
  - Refraction.
  - Absorption.

- Optimal II is 6.
- With II of 6, timing met was 10ns.
- By changing target II to 8, we were able to meet 6ns (initial target was 5ns).
- This translates to 167MHz.

Challenges in Integration

- The eight-core design consists of large number of slaves and masters.
- Problems:
  - AXI_Interconnects have limits in number of slaves and masters.
  - Timings were not met in eight core implementation.
- Solution:
  - Cascading the AXI-Interconnect.



  - Adding register slices on interconnect CP to meet the target frequency.

Outline

- Motivation
- Algorithm Overview
- Design Overview
- Implementation Challenges
  - Pipelining
  - Float Tricks and Other Optimizations
  - Integration
- Verification and Testing Procedure
- Results and Resource Utilization
- Experience with HLS

Testing

- Simulation Test-bench in Vivado HLS for getting the absorption results.
  - We wrote scripts to:
    - Generate random data with different material properties.
    - Convert results to VTK format to visualize them.
  - We compared results to existing tools to check output consistency.

- Verification was done through RTL simulation.
- After Integration in Vivado, we verified the results using behavioral Simulation waveforms.

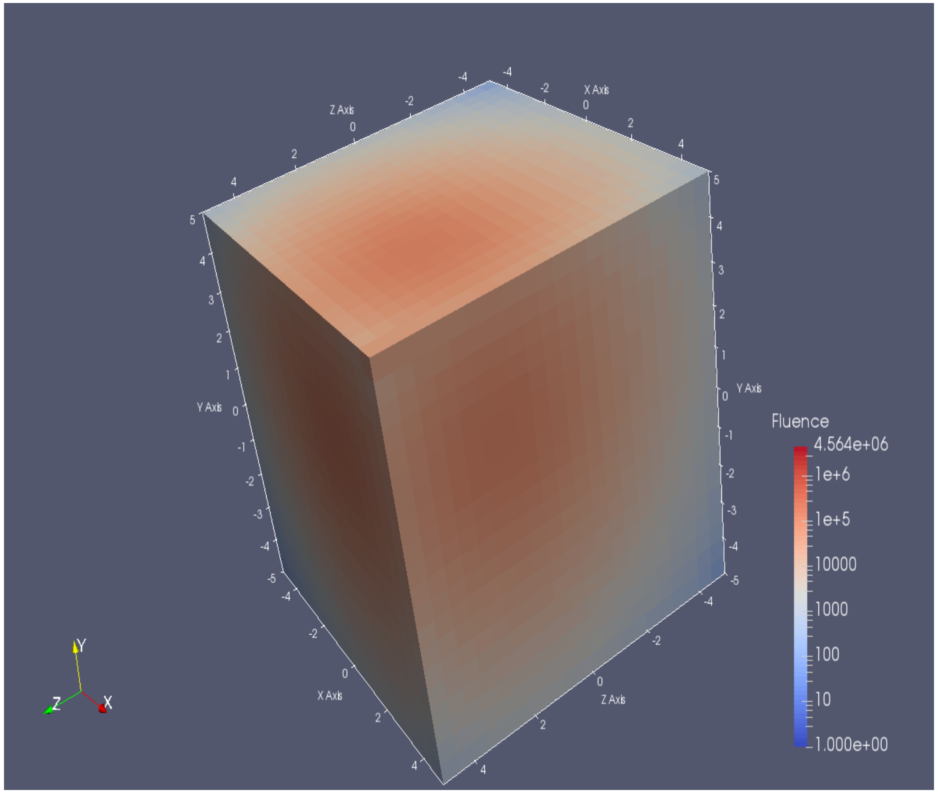
- After moving the design to board, SDK system debugger was used to dump results to files.

Outline

- Motivation
- Algorithm Overview
- Design Overview
- Implementation Challenges
  - Pipelining
  - Float Tricks and Other Optimizations
  - Integration
- Verification and Testing Procedure
- Results and Resource Utilization
- Experience with HLS

Results

- Test case:
  - FPGA: Xilinx® Kintex® Ultrascale™ XCKU115-2 - FLVA1517E (20 nm).
  - One million photons simulated.
  - Tissue size is 8K elements.
  - Eight Kernels.



- CPU time: 14m 30 seconds
- Hardware time: 5 seconds
- Speed-up: 174x

- FullMonte's Time: 16 secs

Resource Utilization

Resource	Available	Utilization	Utilization (%)
LUTs	663K	299K	45
LUTRAMs	293K	20K	7
Registers	1.326M	328K	25
BRAMs	2160	1206	55
DSPs	5520	731	13

Outline

- Motivation
- Algorithm Overview
- Design Overview
- Implementation Challenges
  - Pipelining
  - Float Tricks and Other Optimizations
  - Integration
- Verification and Testing Procedure
- Results and Resource Utilization
- Experience with HLS

Experience with HLS

- Simple, easy to use.
- No need for low level hardware experience.
- It would have taken us at least SIX Months in HDL!

- Issues:
  - HLS hates if-statements!!
  - Couldn’t schedule some independent sections of the code in parallel.
    - Example: handling multiple kernels inside HLS.
  - Minimal support for floating point optimizations.

Thank You!