N-Body Simulation on an Altera DE2-115 FPGA

Enrique Gomez (exg), Maxwell Guo (mwguo)
https://mwguo15.github.io/n-body-on-fpga/

Milestone Report

Summary

During this week, we have completed the naive n-body simulation in SystemVerilog. It performs floating-point computations through the use of IP cores, allowing it to be as precise as possible. We created a naive C++ n-body simulation and a few python scripts to verify our simulation's output on a set of random bodies. Though we need to perform more testing, our synthesized program can fit at least 256 bodies, which exceeds our goals for this week.

As for synthesizing onto the FPGA, we have successfully done so, but have scrapped the idea of sending results from the FPGA to our devices for verification. There are too many things that can go wrong and we would like to focus more on performance. Additionally, we have met the timing requirements for using the DE2-115's 50MHz clock.

Goals

We have exceeded our expectations for this week. Originally, we were willing to accept imprecise results for performance, but we have achieved decent performance with maximal precision. We have mostly figured out the simulation and synthesis workflow to the point where we are confident moving forward. Specifically, we know exactly how to produce precise results via IP cores, verify our outputs, synthesize our code onto the FPGA, use hardware memory controllers, and track performance.

Thus, we believe we can still hit our planned goals and produce our planned deliverables. In our opinion, we have gotten through the hardest part of the project and can comfortably move forward with the systolic array approach. We planned to implement the systolic array approach over these final two weeks, but we believe it is possible to get a working implementation over the next week. Afterwards, we will have time for further optimizations and data collection.

Here are our goals, which are mostly the same as before:

- **Core Deliverables (Plan to Achieve):** Correct, working FPGA implementation of an optimized n-body simulation. This would involve designing the full systolic array architecture that pipelines all n^2 pairwise computations. This implementation would require N memory accesses per pass over ceil(N/K) passes, where K is the number of

processing elements (PEs) in the systolic array. Each PE will compute force contributions for one body. This minimizes the amount of memory accesses and utilizes the DSP blocks better. We hope to compare performance metrics (resource utilization, energy efficiency, speed) with the naive implementation. We also hope to compare speed with a CPU Barnes-Hut approach and GPU approach. Ideally, we want to have a noticeable speedup over our naive approach. All of these comparisons will be represented with benchmarks, graphs, and a trade-off report.

- **Stretch Goals (Hope to Achieve):** In addition to the core deliverables, we would like to attempt scaling the design to handle 10,000+ bodies by further optimizing the systolic array (deeper pipelines) and/or by compressing the data. We would also like to experiment more with the number of PEs and precision (such as reducing the number of iterations our square root module takes). Lastly, we would like to compare the performance to power (perf/watt) ratios for the FPGA and CPU implementations.

Schedule

April 16 – 19

1. **Systolic Array Architecture Planning (mwguo + egomez)**
    - Finalize the block diagram for the systolic array. Determine how many processing elements (PEs) we can reasonably instantiate within the FPGA's LUT and DSP block budget.
    - Map out the data flow: how bodies are streamed into the array, how partial sums of force are passed along, and how the pipeline handles any arithmetic latency (especially square root).
2. **Position/Velocity/Acceleration Calculator Updates (mwguo)**
    - Extend the naive calculator modules into a form compatible with systolic processing.
    - Insert additional pipeline stages into the floating-point operations (add, mul, div) to allow new operations to issue every clock and mask latencies.
3. **Central FSM for Systolic Array (egomez)**
    - Write a top-level finite state machine (FSM) that manages data loading, coordinate broadcasting, and result accumulation from all PEs.
    - Verify that each PE receives the correct body data and that the aggregated forces can update body positions.

4. **Integration & Debugging (mwguo + egomez)**
    - Combine the updated calculators with the FSM into a fully functional systolic array design.
    - Use a small test set of bodies in simulation to ensure correctness before synthesis.

April 20 – 23

1. **Arithmetic Optimization (mwguo)**
    - Refine the floating-point pipelines (especially the square root unit) to reduce cycle counts.
    - Experiment with fewer Newton-Raphson iterations for sqrt, or consider new initial guesses to minimize error without large performance hits.

2. **Systolic Array Tuning (egomez)**
    - Profile how many bodies per pass we can handle versus resource utilization.
    - Adjust the number of PEs (K) to strike a balance between concurrency and timing closure at 50 MHz.
    - Possibly pipeline the array further if we can't meet timing at a higher concurrency.

3. **Integration Tests & Partial Benchmarks (mwguo + egomez)**
    - Run partial benchmarks on the FPGA using small to moderate N values.
    - Compare against the naive design's performance.
    - Verify that resource usage remains within our device limits.

April 24 – 28

1. **Advanced Optimizations (mwguo + egomez)**
    - Explore deeper pipelining strategies, data compression, or approximate arithmetic (like reduced mantissa) if needed.
    - Investigate whether unrolling outer loops or reorganizing memory layouts can reduce stall cycles.

2. **Data Collection & Graph Generation (mwguo, egomez)**
    - Gather detailed throughput and cycle counts for various N.
    - Compare the improved systolic array design against the CPU baseline and naive FPGA design.
    - Plot performance vs. resource utilization, exploring multiple PE configurations.

3. **Final Report Preparation (mwguo + egomez)**
    - Summarize the pipeline design trade-offs, accuracy vs. performance findings, and resource usage.
    - Compile results into clear charts showing speedup (or slowdown) relative to the CPU, power/watt analysis, etc.

Poster Session

For the poster session, we plan to show performance graphs comparing the CPU and naive + fully optimized FPGA implementations (including any implementations between naive and fully optimized). We would also like to do a performance/watt analysis because our FPGA implementation will likely be slower, but much more power efficient. There will not be a demo because there is not much to demo, besides the verification on our FPGA.

Results

For our naive N-Body simulations, we have been using a CPU C++ implementation to validate our results produced by our Altera DE2-115 Cyclone IV FPGA. Our CPU of choice was the Intel i7-12700–found in the ECE machines–which has a 4.9 GHz max clock frequency and 12 cores (8 performance cores and 4 efficiency cores). Our FPGA has a 50 MHz clock frequency, ~140,000 LUTs (Look Up Tables), and 432 M9Ks (which is BRAM memory–a type of SRAM–which we used to read our bodies' data and write our updated body data).

Our current CPU implementation is a single-threaded implementation, which we plan on optimizing to a Barnes Hut multithreaded OpenMP implementation, for future comparison against our systolic arrays and grid-based-approximation FPGA optimization. Since both implementations run in $O(N^2)$ time, we have limited our input body size to small N. This has also allowed us to perform a significant amount of manual testing to gain confidence in the correctness of both our C++ and SystemVerilog program outputs and our validation Python scripts.

Our scripts generate the data necessary for our bodies (i.e. 32-bit single-precision floating-point 2D position, 2D velocity, and mass), and write it into 160-bit words in a MIF (Memory Initialization File). During synthesis, this data is written into our FPGA's BRAM, and our bodies' data is updated in-place. We then use our FPGA's seven-segment segment displays to display the number of computation clock cycles, the data for the first few bodies in memory, or the sum of all the accelerations at the final time step on all bodies due to gravitational forces. The latter, while not the most rigorous method of testing, should be equal to the same sum computed during simulation or with the C++ implementation. However, we performed very rigorous testing for $N \in [2,\ 20]$ during simulation, ensuring that the $(x,\ y)$ positions/velocities/accelerations and masses matched exactly between the C++ and SystemVerilog implementations.

Ensuring that we had these validation methods was critical to proceed with our optimized hardware designs. Additionally, we have recorded some results to have a benchmark for our more parallel design iterations.

Figure 1: Wall-clock Computation Time for N-Body Simulation on FPGA vs CPU for $N \in \{25, 50, 100, 200\}$

	N = 25	N = 50	N = 100	N = 200
Intel i7-12700 CPU (4.9 GHz Max Clock)	0.000081 sec	0.000214 sec	0.000401 sec	0.000946 sec
DE2-115 Cyclone IV FPGA (50 MHz)	1.377545 sec	5.530045 sec	22.160045 sec	88.720045 sec

As shown above, our FPGA design is thousands of times slower than the CPU version. This is expected. Firstly, our FPGA's maximum clock rate (50 MHz) is roughly 100 times slower than the CPU's (4.9 GHz). Additionally, our hardware's datapath is currently an "acceleration calculator," rather than a fully optimized parallel datapath with systolic arrays. Our hardware also executes arithmetic instructions exactly as specified in the SystemVerilog description, whereas the Intel CPU supports out-of-order execution to overlap computations and hide memory latencies. Finally, the CPU has a lot more advanced arithmetic units. This is especially important to note because we are performing lots of floating point multiplication/division/square root operations. Since we did not have access to a floating-point square root IP core for our FPGA, we had to create our own square root approximation module using the Newton-Raphson approximation method. This approximation first makes an initial guess based on the input value, and then performs 2-3 iterations to find a value closest to the true square root. While we found that this approximation was highly accurate, it takes ~150 clock cycles due to the shifting and floating point additions/multiplications/divisions involved, whereas a modern CPU like the i7 we used would take around 20 clock cycles. This also applies to the rest of the floating-point arithmetic instructions. While a floating-point addition could take between 1-3 clock cycles on a CPU, it takes around 4 cycles on our FPGA using the FP IP cores provided by our Quartus synthesization software.

While the CPU significantly outperforms our design, we are left with lots of interesting design choices. Firstly, our synthesized design only uses less than 1% of the LUTs on our FPGA. This means that we still have a lot of remaining resources to design a highly parallel datapath, and we could synthesize several more floating point units to hide our force calculations' arithmetic latency. Additionally, since our design sacrifices little precision, we could consider how to redesign our square-root approximation module. Reducing our number of iterations for square root convergence could save us a significant amount of computation clock cycles (finding a middle-ground is important, since we don't want to entirely sacrifice correctness of course). The differences in our design's output vs. the CPU's output could amortize over larger timesteps.

Finally, now that we know which components in our design are the bottlenecks, we know what computation to overlap early on in our systolic arrays approach.

Memory accesses are currently not a concern for us. While we could scale our body-input size to very large N and observe how our optimized design's computation time scales, this would likely involve using SDRAM memory. However, this would largely complicate certain design choices and would leave us less time to optimize the systolic arrays approach. Therefore, we will stick to using M9K BRAM, and set a target input size of 10,000-20,000 bodies for our final design to compare against the CPU.

Concerns

A key concern lies in the latency of floating-point arithmetic on the FPGA, particularly the square-root approximation module, which currently requires around 150 clock cycles per operation. Other floating-point operations (multiplication, division, and addition) also take significantly longer on the FPGA than on a modern CPU. To address this, we plan to pipeline these operations more aggressively and experiment with fewer iterations in the Newton-Raphson method for square root. Our goal is to reduce cycle counts while maintaining acceptable numerical accuracy.

We are also attentive to the trade-off between resource utilization and parallelism. While our naive design currently uses less than one percent of the LUTs, a fully expanded systolic array could place heavier demands on FPGA resources. We intend to add processing elements (PEs) incrementally, carefully monitoring resource usage, and only insert additional pipelining where it meaningfully improves performance.

Memory bandwidth and scalability are another concern—particularly for larger N values—because multiple PEs accessing memory in parallel can saturate on-chip resources. We plan to stay within the bounds of the on-chip BRAM for a target of roughly 10,000 to 20,000 bodies. Should bandwidth constraints surface, we will adjust the level of concurrency or investigate simple burst-based SDRAM approaches without overcomplicating our design.

Additionally, while our C++ and Python scripts have already proven effective for validation, we recognize the need for more robust corner-case testing as we move to more advanced designs. We will maintain thorough simulation testbenches and compare outputs (positions, velocities, and partial sums of forces) to our CPU implementation over multiple time steps to ensure correctness before performing final synthesis.

Lastly, we are aware of the tight timeline. Implementing and optimizing the systolic array, followed by extensive data collection and reporting, must be completed in only a few weeks. Our approach is to focus on a minimal but functional systolic design first, then apply further

optimization, deeper pipelining, and approximate arithmetic techniques if time permits. This strategy will help us manage complexity while still delivering meaningful performance and energy-efficiency results.