OPTIMIZING BALL TO BALL COLLISIONS ON ZEN 3 ARCHITECTURE

Kai Wen Li, Nils Egger, Elias Ruff, Lucas Pfingsten

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Pooltool is a general purpose billiards simulator written in Python. It handles the physics and collisions of the balls as physically accurate as possible. It supports different solvers, one of which is based on the paper "Numerical simulations of the frictional collisions of solid balls on a rough surface" by Mathavan et al. [1]. Although it can already be played at real-time, this paper focuses on taking the Just-In-Time (JIT) compiled python to C, and optimizing it to the best extent possible. Our optimizations lead to a speedup of 4x compared to the JIT python on the AMD Zen 3 architecture. As will be discussed, the method becomes limited by register pressure, as the main loop is heavily dependent on the previous iteration, limiting parallelism. Our best approach is based on scalar optimizations, such as utilizing Fused-Multiply-Add (FMA) instructions, improving branch prediction and approximating the inverse square root using the Newton-Raphson method.

1. INTRODUCTION

Motivation. Pooltool is a billiards analysis tool designed to simulate and study the physics of potential shots. Its ball-to-ball collision algorithm operates in several stages. When two balls make contact, the accumulated energy is dissipated over multiple iterations. During each step, both the linear and angular velocities of the balls are updated. A key tradeoff emerges here. While increasing the number of iterations reduces the step size and improves accuracy, it also increases computational time. Therefore, it is essential to optimize the code to maintain real-time performance without sacrificing precision.

The original implementation was likely developed with speed as the primary focus, but it lacked flexibility for fine-grained control. The collision loop is inherently sequential from a physics standpoint, making it difficult to parallelize. This also results in significant register usage due to intermediate calculations. Additionally, the main loop is filled with nested conditional statements, which are highly susceptible to branch mispredictions. The only values explicitly read from and written to memory are the positions, velocities, and angular velocities, totaling 18 doubles. This suggests that the problem is computationally bound rather than memory-bound, indicating substantial potential for optimization. However, as will be demonstrated, the primary bottleneck lies in register pressure.

Contribution.

We present a scalar optimization of the original problem. It optimizes for branch prediction, instruction mix and inverse square root approximation using the Newton-Raphson method.

2. BACKGROUND ON THE ALGORITHM/APPLICATION

Collision solver. The `collide_balls` routine implements an impulse-based collision solver for two identical, rigid billiard balls (radius R, mass M). Given each ball's position $\mathbf{x}_i$, linear velocity $\mathbf{v}_i$ and angular velocity $\boldsymbol{\omega}_i$ ($i = 1, 2$), it computes post-collision velocities by distributing the normal impulse over N sub-steps to model restitution and sliding friction.

First, we build a contact-frame. Let

$$\Delta = \mathbf{x}_2 - \mathbf{x}_1, \quad d = \|\Delta\|, \quad \mathbf{f} = \frac{\Delta}{d}, \quad \mathbf{u} = (0, 0, 1),$$

and define the right axis

$$\mathbf{r} = \mathbf{f} \times \mathbf{u},$$

so that $\{\mathbf{r}, \mathbf{f}, \mathbf{u}\}$ is orthonormal. We then project velocities:

$$v_{x,i} = \mathbf{v}_i \cdot \mathbf{r}, \quad v_{y,i} = \mathbf{v}_i \cdot \mathbf{f}, \quad \omega_{x,i} = \boldsymbol{\omega}_i \cdot \mathbf{r},$$
$$\omega_{y,i} = \boldsymbol{\omega}_i \cdot \mathbf{f}, \quad \omega_{z,i} = \boldsymbol{\omega}_i \cdot \mathbf{u}.$$

Surface slip at each ball's contact circle is

$$\mathbf{u}_{s,i} = \big(v_{x,i} + R\,\omega_{y,i},\ v_{y,i} - R\,\omega_{x,i}\big),$$

and the ball–ball contact-point slip velocity components are

$$v_c^x = v_{x,1} - v_{x,2} - R(\omega_{z,1} + \omega_{z,2}),$$
$$v_c^z = R(\omega_{x,1} + \omega_{x,2}), \quad v_c = \sqrt{(v_c^x)^2 + (v_c^z)^2}.$$

We distribute the normal impulse per sub-step as

$$\Delta P = \begin{cases} \dfrac{1+e_b}{2}\, M\, \dfrac{|v_{y,2}-v_{y,1}|}{N}, & \Delta P_{\text{in}} = 0, \\ \Delta P_{\text{in}}, & \text{otherwise}, \end{cases}$$

apply normal and tangential impulses based on restitution e_b, ball–ball friction u_b and sliding frictions u_{s1}, u_{s2}, update local velocities and spins via

$$v_{x,1} \mathrel{+}= \frac{\delta P_1 + \delta P_x}{M}, \quad \omega_{x,1} \mathrel{+}= C\,(\delta P_2 + \delta P_y), \ \ldots$$

with

$$C = \frac{5}{2MR},$$

and repeat until the normal relative velocity reverses or the accumulated work $W \geq (1+e_b^2)\, W_{\text{comp}}$. Finally, we project the local $(v_x, v_y, \omega_x, \omega_y, \omega_z)$ back into world-space:

$$\mathbf{v}_i' = v_x\, \mathbf{r} + v_y\, \mathbf{f}, \quad \boldsymbol{\omega}_i' = \omega_x\, \mathbf{r} + \omega_y\, \mathbf{f} + \omega_z\, \mathbf{u}.$$

Cost measure & analysis. For the cost measurement and analysis, we primarily used explicit operation profiling. Using preprocessor directives (#ifdef), we measured the FLOPS of each implementation and counted the number of key operations, such as square roots, additions, multiplications, and divisions. In addition to this, we performed memory usage measurements, though these offered limited insight since, as noted in the introduction, only 18 doubles are loaded from and stored to memory. We also used the Time Stamp Counter (TSC) to record execution time, both in CPU cycles and nanoseconds. These timing measurements were taken for the entire benchmarked implementation as well as for individual major components during profiling.

3. OPTIMIZATION PERFORMED

3.1. Baseline Implementation / Initial C Implementation

The initial baseline implementation, `collide_balls`, is a basic port of the Python code and simulates, as described in the previous part, the collision between two rigid balls. The code can generally be split up into three parts: *pre-loop initialization*, *while loop*, and *post-loop transformation to global coordinates*. This implementation offers no optimizations at all.

In addition to the custom profiler described in the previous section, we also employed perf to guide our optimization efforts. We used it to identify the most expensive lines and performance bottlenecks. While the timing results from perf closely matched those of our own profiler, it provided additional low-level metrics such as CPU cycles, cache misses, branch mispredictions, instructions, instructions per cycle, and frontend stalls (backend stalls were not available on the target architecture). Running perf on the initial baseline implementation revealed the following key insight:

Key Finding: Approximately 95% of the total execution time is spent within the inner loop, mainly due to sqrt and division operations as well as branching instructions.

3.2. Scalar: Square Root Optimizations

As previously mentioned, the baseline implementation performs numerous square root computations. Upon closer analysis, we observed that not all of these were necessary in every iteration of the loop. By moving the corresponding `sqrt` operations inside conditional branches, we were able to reduce the number of square root computations from three to two per iteration.

For the next optimization, we noted that reciprocal square roots were required for all calculations involving `sqrt`. To address this, we utilized the SSE instruction `rsqrt`. However, since our algorithm operates on `double` precision values, this approximation resulted in a loss of accuracy, falling short of our target (around 10^{-6} precision). To resolve this, we combined the `rsqrt` instruction with a single Newton-Raphson refinement step, which successfully restored the required precision.

3.3. Scalar: Branch Prediction

The main loop of the simulation begins with a block of, in the worst case, four nested `if` statements. This is followed by the computation of delta velocities and then a final conditional check via another `if`. Early profiling with `perf` revealed that the algorithm is particularly sensitive to branch mispredictions. Interestingly, we discovered that even seemingly minor reordering of branches (for example, placing conditions that occur in 99.9% of cases first instead of handling them in the `else` clause) could increase branch mispredictions by just 0.05%, yet cause a noticeable worse runtime performance.

To further work on this, we attempted to guide the compiler's branch prediction using the `builtin expect` intrinsic, applying it consistently across all key branches. This offered additional modest improvements by making the branch behavior more predictable at the hardware level.

In the next step, we explored whether we could eliminate branching altogether by replacing conditional logic with bitmasking techniques. Since we are working with `double` precision values, we employed `union` types to reinterpret bit representations for masking purposes. However, this approach proved counterproductive: while it successfully removed branches, it increased the instruction count

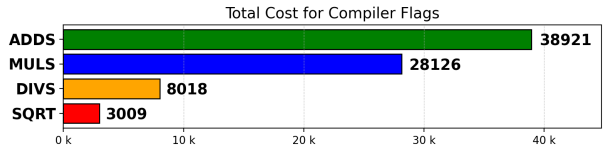

Fig. 1. *Cost Analysis of the basic implementation for $N = 1000$. Remember, $N \neq$ Loop Iterations. There are about 5000 loop iterations.*

by approximately 12% and worse, resulted in longer total runtimes.

An additional insight came from examining the nested `if` structure itself. By inverting the branching logic, we could rearrange computations such that all relevant operations, including the three square roots, are executed unconditionally before any branching. Then, based on the inverted conditions, irrelevant results are masked or zeroed out. Surprisingly, this restructuring did not degrade performance. In fact, in some cases, it even led to marginal improvements, shaving off several hundred CPU cycles.

Focusing on the final `if` condition inside the loop, we recognized a key property: it is guaranteed to evaluate to `true` only once per complete execution. This meant that, in practice, we were redundantly evaluating this branch in roughly half of the total iterations. `perf` profiling confirmed this inefficiency. To optimize this, we implemented a two-stage loop structure. The first loop runs until the final condition is triggered and a second loop resumes without the redundant `if` statement. Since both loops are functionally equivalent except for the absence of the final condition in the latter, this restructuring aimed to reduce branching overhead. The first loop's condition was simplified to a `while(true)` to streamline logic further.

Unfortunately, due to the relatively small number of loop iterations typical for our real-time scenario, the overhead of managing two loops outweighed the benefits, and this approach did not yield a net performance gain. However, it is an interesting experiment to see how it performs in contexts with significantly higher iteration counts, as we will be doing in the next major section.

3.4. Scalar: Precomputation & FMA

Consequently, after applying various combinations of the scalar optimizations discussed previously, such as branch prediction and reducing square root operations, these components were no longer the primary bottlenecks according to `perf` measurements. Instead, the top performance limiter was the huge volume of arithmetic operations, particularly `add` and `mul` instructions. As shown in our cost analysis (Figure 2), these account for approximately 85% of all executed instructions, making them a clear target for further optimization.

To tackle this, we experimented with two primary strategies: aggressive precomputation of redundant arithmetic expressions and the substitution of separate multiply and add operations with fused multiply-add (FMA) instructions, where possible. While FMA yielded improvements, the results of precomputation were unexpectedly disappointing. In fact, precomputing intermediate results often led to a worse runtime, despite a reduction in raw instruction count.

This counterintuitive outcome initially left us puzzled, particularly because none of the usual `perf` statistics, such as cache misses, branch mispredictions, or CPU stalls, pointed to a clear cause. Profiling at the function level also failed to give us any insight, primarily because such profiling disrupts instruction-level parallelism (ILP). In order to query performance counters like cycle counts, all instructions must be completed, thereby artificially serializing execution and masking potential bottlenecks in instruction throughput.

Suspecting a deeper microarchitectural issue, we hypothesized that the performance regression was related to register pressure. The loop structure in our implementation is highly sequential, requiring many values to be held across multiple operations. This can easily saturate the limited number of physical registers, forcing register spills and stalls. Since our target CPU architecture does not expose register pressure metrics through `perf` (same for AMD uProf), we turned to static performance simulation using `LLVM-MCA`. The simulation results strongly supported our hypothesis: while floating-point execution ports remained underutilized, register dependency pressure reached up to over 90%, significantly constraining throughput.

This hypothesis is further validated by comparative benchmarks on architectures with larger physical register files. For example, Apple's M1 chip, with double the amount of registers, ran the same implementation up to four times faster than AMD's Zen 3, which has more limited register capacity. These observations strongly suggest that register pressure, rather than arithmetic latency or memory bandwidth, had become the new limiting factor in our pipeline. More about this can be seen in the experimental results section.

On a more positive note, the introduction of FMA instructions provided measurable gains in efficiency in certain implementations, more specifically in the reciprocal square root implementation (in the original implementation the FMA changes led to worse runtimes due to a different instruction mix). By collapsing separate multiplication and addition operations into single instructions, we reduced dependency chains and freed up valuable execution slots, improving both throughput and latency without negatively impacting accuracy or portability.

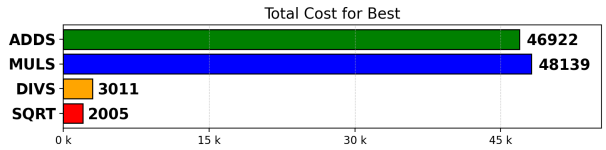

Fig. 2. *Cost Analysis of our best optimization. Compared to Figure 1 there is a reduction in* `divs` *and* `sqrt`*, but a large increase in* `add` *and* `mult` *due to the Newton-Raphson step per reciprocal square root.*

3.5. Scalar: Register Relieve

As identified in our previous analysis, register pressure is the biggest bottleneck. To address this issue, we rewrote the performance-critical loop with explicit register relieve in mind. The optimization strategies employed were as follows:

- **Array-based packing**: By packing related scalars such as local velocities and angular velocities into arrays, we hoped to reduce register aliasing and improved the compiler's ability to reuse registers efficiently. This also has the added effect of increased spatial locality.
- **Trading operations for variables**: Instead of saving more temporary variables, which hold precomputed values, we opted for more cheap operations and less variables inside the loop.
- **Scope-limited computation blocks**: The loop was split into distinct blocks (e.g., `impulse`, `delta`, `velocity`) with minimal overlapping lifetimes. We hoped to help the compiler discard intermediate results earlier and free registers more aggressively.

Unfortunately, while *perf* indicated a slight improvement from these changes, the optimizing compiler, discussed in the following section, had already addressed most of these opportunities. As a result, register pressure remained a significant issue. Although it was somewhat reduced, it continued to represent a major bottleneck, one we were ultimately unable to resolve due to architectural limitations.

3.6. Scalar: Compiler Tuning

GCC versus Clang: While online discussions often suggest that GCC generally produces slightly faster code for compute-bound workloads, we decided to evaluate this claim again in our case. Through experimentation, we observed that GCC consistently outperformed Clang in terms of runtime, both with default settings and with aggressive optimization flags.

We speculate that some of this performance difference may come from different strategies in register allocation. GCC's global register allocator tends to be more aggressive in retaining values in registers within tight loops, which helps alleviate pressure in compute-intensive regions. In contrast, Clang's default register allocator appears more prone to register spilling under high pressure, leading to additional memory traffic and degraded performance in our case.

Through testing multiple different flags, we found that the ones below produced the best results.

GCC compiler flags: `-mfma -fno-math-errno -Ofast -funroll-loops -ftree-vectorize -mavx2`

3.7. Single Instruction, Multiple Data (SIMD)

In the first SIMD implementation, we kept the while loop scalar and focused on vectorizing the surrounding operations. Using AVX intrinsics, we stored most variables in AVX registers and applied vectorized operations where applicable. However, this resulted in no performance improvement, likely because the compiler with our flags had already performed similar vectorization. This was confirmed by inspecting the generated assembly code.

In the second, fully vectorized SIMD version, we targeted the loop itself. We again stored all relevant data in AVX registers and used SIMD instructions, including blends, permutes, and bitmasking, to emulate the `if` condition within the loop. Unfortunately, this led to worse performance. The compiler had already applied similar optimizations, and our use of complex instructions like permute and blend introduced additional computational overhead, resulting in an overall slowdown.

In the final version the whole SIMD implementation was rewritten in SSA style and we perform dot, cross and impulse updates manually. Furthermore, we use much more extensive broadcasting and in the branching step, use more explicit conditional blending with multiple mask operations. While this approach was an improvement to previous SIMD approaches, we were unable to beat the best scalar version as we once again had register pressure. A significant reason for this was the need to extract scalar values from SIMD vectors inside the loop. Because some of these values were simultaneously relevant for both SIMD computations and scalar logic, we spent considerable time rearranging vectors and moving data between SIMD and scalar registers, ultimately negating much of the performance gained inside the loop from vectorized computation. This also exacerbated register pressure, further limiting the benefits of SIMD in our case.

3.8. Single Program, Multiple Data (SPMD)

As a last effort to find further optimizations gains, we implemented SPMD on SIMD, where each SIMD lane represents one independent computation, allowing four computations to be processed simultaneously. Our algorithm is a great candidate for this use case because of its sequential nature. Hence, processing multiple collisions at once is easier than making one collision more parallel. For this, the function is modified to receive the data of 8 balls simultaneously. The only tricky part is to consider that not all collision require the same amount of iterations in the while loop, hence a mask for finished operations needs to be added.

Due to the elimination of branching and therefore its included speedups, the application of our other most effective performance improvements, such as FMA and reciprocal square roots only yielded minimal improvements. As confirmed with the assembly code, the compiler was able to more effectively apply FMA himself in this version, with only negligible changes resulting from manual FMAs. And the reciprocal square root gains were mostly offset by the Newton-Raphson step.

As will be seen in our results, the method is twice as fast as handling four collisions sequentially. Although the use case of four simultaneous collisions in a billiards physics simulation, at exactly the same time step, is debatable.

4. EXPERIMENTAL RESULTS

In the following section we will show our experimental results and visualize how we went from the original implementation to our best version as seen in 3.

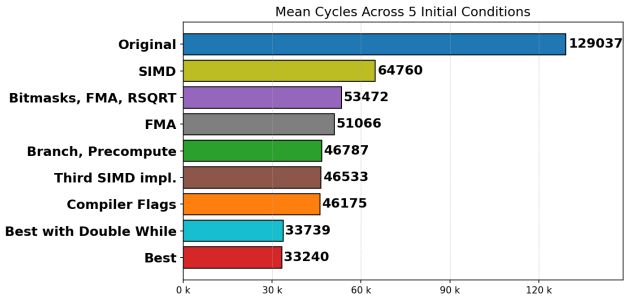

Fig. 3. *A selection of our methods compared, ordered by average cycles count across 5 different initial conditions.*

Experimental setup: Our testing and benchmarking was primarily conducted on two systems within our group of four. Both systems were running Ubuntu 24.04.2 LTS and featured AMD Ryzen processors on the AM3 platform. Although we initially included Windows in our testing, results on it proved inconsistent and were ultimately excluded from our analysis.

The compilers used across both systems were GCC version 13.3, configured with the optimization flags described earlier. The two machines used for testing were equipped with a Ryzen 5 5600X and a Ryzen 7 7735U, respectively. Unless otherwise noted, all measurements and plots presented in this report were taken from the latter system (Ryzen 7 7735U).

All benchmarks were run on a small, consistent set of test cases, averaging results over 10,000 iterations, with an additional 1,000 warm-up iterations to eliminate cold-start artifacts. We also experimented with cache flushing, but found it had no measurable impact on performance. To minimize bias, the execution order of tests was randomly shuffled in between runs.

Results: Looking at the flops/cycle in 4, we can clearly see that our best implementation does significantly better than most of the rest in terms of runtime. This is due to an optimized instruction-mix, combined with all of the positive optimization methods applied. The double while loop has a slightly better flops/cycle value here as expected, but does not outperform our best approach due to added instructions and loop overhead, even for larger N as seen in Figure 12.

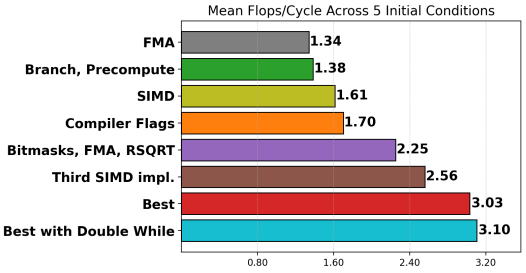

Fig. 4. *A selection of our methods compared, ordered by average flops per cycle across 5 different initial conditions.*

Branch Misses: In figure 5 we can see the effect of our branch optimizations, in the two best versions where we apply the above mentioned methods. We were able reduce a noticeable amount of branch misses. It's also visible here that some of the SIMD vector operations confused the compiler, resulting in worse branch predictions and thereby resulting in higher branch miss count.

Cache Misses: Looking at the relative cache misses in 6, it can be clearly seen that the relative cache misses of our implementation actually got worse in time. However, this is a side effect from the fact that the total amount of cache misses actually were severely reduced as seen in 7 through our optimizations.

Furthermore, through testing and optimizing for temporal as well as spatial locality whenever possible, we noticed that the difference, between a cache optimized implementation and a "standard" implementation, was negligible and often resulted in worse ILP. We therefore opted for a slightly

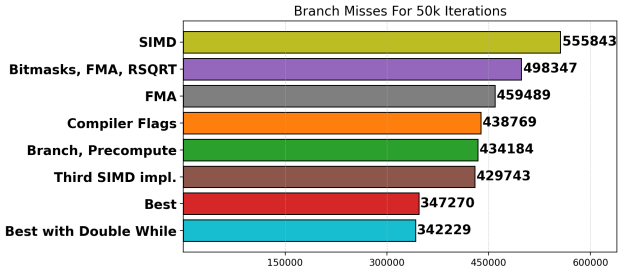

Fig. 5. *A selection of our methods compared, ordered by average total branch missess over 50'000 iterations.*

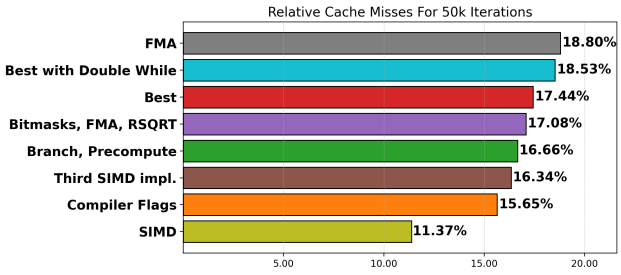

Fig. 6. *A selection of our methods compared, ordered by average relative cache missess over 50'000 iterations.*

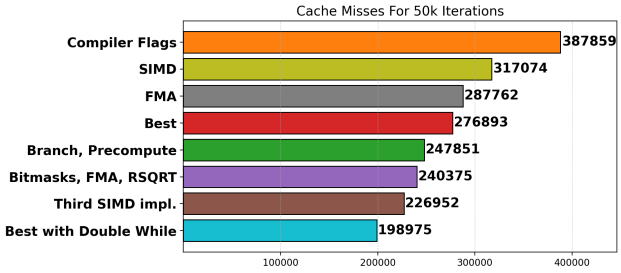

Fig. 7. *A selection of our methods compared, ordered by average total cache missess over 50'000 iterations.*

worse relative cache miss in favor of better runtimes.

Register Pressure: Throughout the optimization process, register pressure consistently emerged as a significant challenge. As shown in Figure 8, backend pressure steadily increased with the addition of each optimization technique. In our most optimized version, where nearly all methods were applied, backend pressure peaked at approximately 95%. This highlighted the constant tradeoff we had to face: achieving lower runtimes at the cost of increased backend pressure.

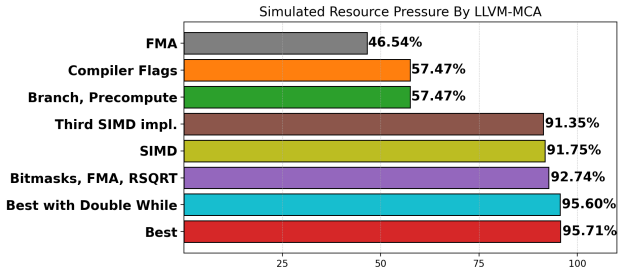

Fig. 8. *LLVM-MCA simulated resource pressure.*

To further validate our observations, we tested the implementations on an Apple M1 processor. As seen in Figure 9, average runtimes decreased across all implementations and also instructions per cycle generally improved, which could potentially be contributed to the additional (twice as many!) registers this chip has over our AMD processors. Interestingly, most optimizations that reduced runtime on the AMD systems actually led to performance degradation on the M1. This likely reflects the fundamentally different bottlenecks and architectural characteristics between the AMD platforms and Apple's M1.

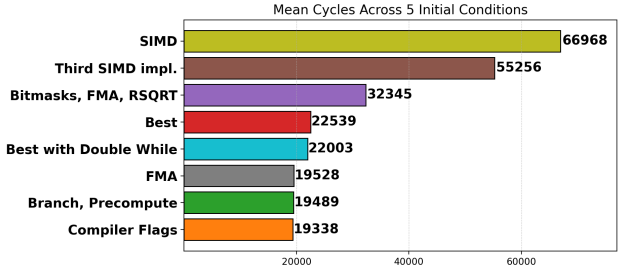

Fig. 9. *Apple M1: average cycles across 5 different initial conditions.*

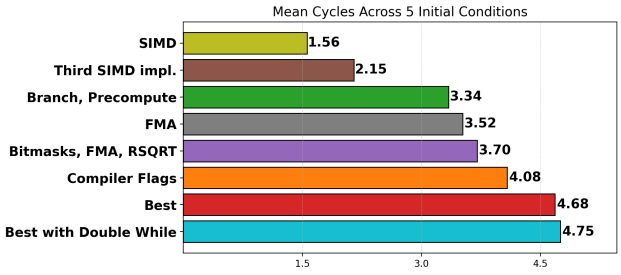

Fig. 10. *Apple M1: average FLOPS per cycle across 5 different initial conditions.*

SPMD: In Figure 11 one can see the beneficial impact of handling four collisions at once. The method is twice as fast as running our best optimization four times sequentially.

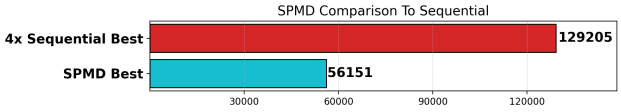

Fig. 11. *Our best scalar optimization, executed four times sequentially, compared to SPMD.*

Smaller Steps, Larger N: Lastly, we wanted to see if any of our implementations perform specifically better with increasingly larger *N*. In Figure 12 we can see that the growth of runtime for all the implementations is consistent for any *N*, meaning, since the loop iterations are highly sequential, there is no method distinguishing itself for the use of larger `N` any more than our best optimization.

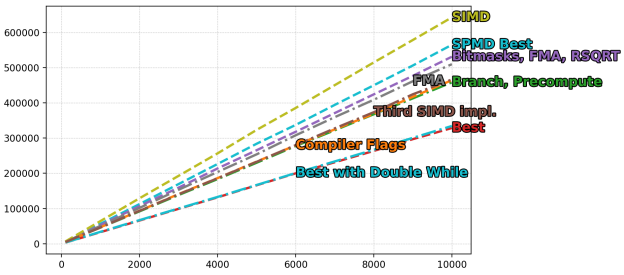

Fig. 12. *A selection of our methods compared by growth in cycles given different N. All methods display a linear growth in runtime.*

5. CONCLUSION

In conclusion, we began with a Just-In-Time compiled Python implementation of the ball-to-ball collision function and incrementally optimized it through a cycle of analysis and refinement. Starting with basic improvements like eliminating unnecessary square roots, we progressed through enhancements involving branch prediction, precomputation, register pressure relief, and finally SIMD vectorization.

Despite these efforts, we encountered architectural problems, most notably, register count limitations on our two test systems, which, along with the inherently limited parallelism of the collide function, became the primary bottlenecks. While we achieved an overall speedup of 4x compared to the scalar baseline, our SIMD implementation did not surpass this gain.

Each optimization stage was thoroughly analyzed, and our findings consistently pointed to register pressure as the dominant limiting factor. Our experimental results support this conclusion and highlight the diminishing returns of further optimization under these constraints.

We believe however, that under different processors, further performance gains are possible.

6. CONTRIBUTION OF TEAM MEMBERS

- **Kai Wen Li and Nils Egger:** *Optimizations:* Initial C implementation, All scalar optimizations discussed in the report, meaning sqrt, branch pred, register relieve, compiler choice and flags, double while, bitmasking, FMA and precomputation. Initial two SIMD implementations, namely a SIMD implementation with the scalar loop as well as another implementation with a fully vectorized loop. Totaling 25/27 variations of `collide balls`.

  Analysis: Profiling and benchmark codes for analysis and plots.

- **Elias Ruff:** Focused on SIMD, rewriting the SIMD implementations to be fully vectorized, including the loop part, with Lucas.

- **Lucas Pfingsten:** Focused on SIMD, enhancing our implementations by vectorizing the core computational loop to improve instruction-level parallelism, with Elias.

7. REFERENCES

[1] S. Mathavan, M. R. Jackson, and R. M. Parking, "Numerical simulations of the frictional collisions of solid balls on a rough surface," *Sports Engineering*, 2014.