ACCELERATING MATHAVAN'S BILLIARD BALL COLLISION ALGORITHM

Joris Belder, Gül Oymak, Wilton Arthur Poth

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Sports and games, including cue sports such as pool, have been around for centuries. With the advent of computers, many of these activities were gradually digitalized into a video game form. An example of this is *Pooltool*, which aims to accurately depict reality using a previously developed physics simulation algorithm by Mathaven et al., for simulating the outcome of colliding billiard balls. While the original implementation is written entirely in Python, this work optimizes this implementation starting with an equivalent version written in the C programming language. By iteratively applying several structural and algorithmic transformations, a speedup of $1.93\times$ was achieved compared to the initial C implementation, albeit with a minor trade-off in result accuracy.

1. INTRODUCTION

Cue sports have become increasingly popular over the years, now attracting millions of viewers worldwide [1]. While there are many variations of the sport, they all involve precisely striking a cue ball with the aim of accurately directing other billiard balls through controlled collisions between the balls. The physics underlying these collisions have long been an area of study, with publications dating back as early as 1835 [2].

With a deeper understanding of the physics, it has become possible to algorithmically simulate billiard ball collisions accurately. These algorithms are employed in a wide range of applications, including training system for professional billiard players [3], or the development of cue sports playing robots [4]. They also aid in the training of artificial intelligence systems designed to formulate game-playing strategies for cue sports [5], and ball-tracking systems for TV broadcasting, such as *Hawk Eye* [6]. Additionally, the algorithms are used to create realistic digital cue sport games, such as the open-source billiards simulator *Pooltool* [7].

The algorithm used by *Pooltool* to simulate billiard ball collisions is Mathaven's algorithm [8]. It computes the physically accurate outcome of two billiard balls colliding based on their velocity, spin, friction, and coefficient of restitution of the balls. Mathaven's algorithm achieves significantly more accurate results than previous works, by incorporating the effects of both spin and friction in the computation. Furthermore, the algorithm is generalisable as it can be applied to various cue sports, where balls may differ in size and weight.

To the best of our knowledge, *Pooltool* is the only open-source codebase that includes an implementation of Mathaven's algorithm. The implementation is a relatively straightforward adaptation of the algorithm described by Mathaven, written in Python, with minor optimisations applied to avoid the use trigonometric functions. However, the current implementation leaves room for improvement in terms of runtime. A faster implementation would benefit several use cases, for instance, reducing the training time of artificial intelligence systems, and improving the responsiveness of interactive applications such as *Pooltool*. Motivated by these limitations, this work aims to improve upon the existing *Pooltool* implementation.

In this paper we present an improved implementation of Mathaven's algorithm, achieving a $1.93\times$ speedup over the base implementation on the *Zen 2* microarchitecture, while maintaining output accuracy within 1% relative error of the original output. We begin by providing an overview of the original implementation in Section 2. The applied optimisations and validation methodology are detailed in Section 3. Subsequently, we present our experimental results and analysis in Section 4. Finally, we conclude with a summary of our findings in Section 5.

2. BACKGROUND

In what follows, we illustrate the implementation of Mathaven's algorithm as used in the *Pooltool* simulator, to provide context for the optimisation work described in Section 3.

2.1. The Base Algorithm

Mathaven's algorithm simulates the interaction between two colliding billiard balls, the cue ball and object ball [8]. The input to the algorithm includes the location, velocity, and angular velocity of the balls, as well as their physical pa-

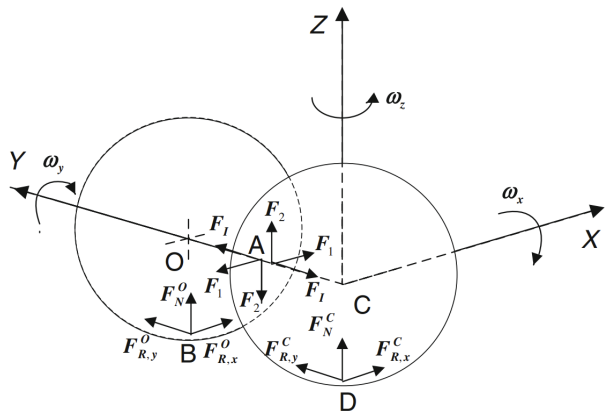

Fig. 1. Illustration of the local reference frame used showing the X, Y, and Z axes, cue ball (C), object ball (O), the 3 contact points A, B, and D as well as the forces acting at the contact points [8].

rameters: mass, radius, ball-table and ball-ball friction coefficients, coefficient of restitution, and a parameter N.

The algorithm begins by transforming the global position, velocity, and spin vectors of the two balls into the local reference frame shown in Figure 1 using basic linear algebra. The main computation is then performed in a large `while`-loop that continues until all the work in the collision is completed. The parameter N determines the number of iterations by controlling the size of the impulse, ΔP, applied from the cue ball to the object ball in each step. ΔP is chosen such that the loop runs for approximately N iterations.

The work is performed in a compression phase where the balls compress at the point of contact and a restitution phase where the balls spring back. During each iteration, a set of impulses representing the forces acting at the contact points are computed. The presence of the impulses between the balls and the table depends on whether the balls make contact with the table, which in turn depends on the presence of slip at the contact point between the balls. This interdependence is handled using a nested branch structure inside the loop.

Based on the calculated impulses, the velocities and angular velocities of the balls can be updated. The updated velocities are then used to update the frictions at the contact points and other variables. At the end of each iteration, the total work performed is incremented, and the algorithm checks whether the compression phase has completed. When the restitution phase begins, the total amount of work required for the remainder of the interaction is computed based on the energy dissipated during the compression phase.

Finally, after the `while`-loop completes, the final velocities and angular velocities are converted back into the global reference frame and returned. A pseudocode representation of the base implementation is shown in Algorithm 1 for reference.

Algorithm 1 Schematic of base implementation

```
Projecting global vectors into local reference frame
ΔP = ...                                   ▷ Dependent on N
while W < W_f and in Compression Phase do
    ΔP_i's initialised to zero
    if Movement at contact point then
        ΔP_1 = ...
        if Z-movement at contact point then
            ΔP_2 = ...
            ...                            ▷ More branches
        end if
    end if
    Update velocities and frictions based on ΔP_i's
    Increment W                            ▷ Dependent on ΔP
    if Start of Restitution Phase then
        W_f = ...          ▷ Sets final work, depends on W
    end if
end while
Project local vectors to global reference frame
return velocity and spin vectors
```

2.2. Profiling

To measure various performance metrics, Core Performance Monitoring Counters (PMCs) built into the hardware are used [9, sec. 2.1.15]. These PMCs provide access to measurements such as clock cycles (cyc), the number of floating-point operations (FLOPs), and cache accesses, among many others. On the *Zen 2* microarchitecture, operations counted as FLOPs are additions, multiplications, divisions, square roots, and Fused Multiply-Adds (FMAs). Another PMC of interest is the number of token stall cycles, which are cycles where the CPU is ready to dispatch instructions but it cannot due to scheduling constraints, such as resource contention or instruction dependencies. In this paper, the main focus is laid on reduction of the runtime of the algorithm. This refers to the number of cycles the computation takes until completion, while performance, another commonly used metric, refers to the number of FLOPs per cycle.

3. OPTIMISATIONS

Starting with the *Pooltool* implementation mapped from Python to C as the base implementation, optimisations were applied iteratively. The optimisations are primarily algorithmic, with the aim of reducing the number of operations performed. Memory hierarchy and data layout optimisations

were not considered, as all inputs, outputs, and intermediate states fit into L1 cache entirely. In some optimisations, functions were swapped for faster approximation functions, resulting in a faster runtime, but less accurate output.

Hereafter, we motivate each optimisation, discuss the changes made to the code, as well as describe the methodology used to validate the correctness of the optimisations.

3.1. Scalar Optimisations

We first present the scalar optimisations, which do not rely on any explicit packed vector operations. The optimisations are presented in the order they were applied, with the names of the implementations in bold.

Reciprocal Square Root (rsqrt). The base implementation performs multiple square root computations within the main loop, with each result subsequently used as a divisor. Both divisions and square root calculations are computationally expensive operations with latencies of 13 and 20 cycles, respectively, on our test system using *Zen 2* [10].

To reduce the cost, we explored several options. A fast reciprocal square root approximation function is Greg Walsh's famous "Fast Inverse Square Root" algorithm [11], of which many adaptations exist. Alternatively, modern CPUs also provide built-in instructions for computing the reciprocal square root. We settled for the latter, as our analysis found the reciprocal square root instruction to be significantly faster, requiring only 5 cycles [10], compared to both the original method and all "Fast Inverse Square Root" variants evaluated.

However, the optimisation introduces a trade-off in terms of numerical accuracy. On systems without AVX-512, which includes ours, the reciprocal square root instruction is available only for single-precision `float` types, thus requiring type conversion between `float` and `double` for our purpose [10]. The instruction is accurate up to roughly 12 bits ($\approx$ 4 decimal digits) compared to 53 bits ($\approx$ 16 decimal digital) using the original method [12].

Our testing showed that, using the instruction, the output remained within a relative error of 0.1% with respect to the baseline. Further discussion of the error tolerance and justification is provided in Section 3.3.

Reciprocal Mass (recmass). When updating the velocities of the balls in the `while`-loop, the mass of the balls, M, appears as a divisor. This results in four division operations per iteration, which are expensive as mentioned above.

To reduce the runtime, the reciprocal mass, M^{-1}, can be pre-calculated once outside the loop. The reciprocal mass is then used as a multiplicand in place of a division, thereby reducing the instruction latency to 3 cycles per multiplication [10].

Common Subexpression Elimination (cse). Common subexpression elimination reduces the number of operations by replacing subexpressions that are computed multiple times with a single stored result. Many computations are symmetric for both balls and therefore share common subexpressions.

We applied CSE at four locations, including within the main loop. One such example concerns the computation of the impulses ΔP_1, and ΔP_2:

```
deltaP_1 = -u_b * deltaP * u_ijC_x * u_ijC_xz_mag;
deltaP_2 = -u_b * deltaP * u_ijC_z * u_ijC_xz_mag;
```

The expressions differ in only one variable, thus enabling the following optimisation:

```
double t0 = -u_b * deltaP * u_ijC_xz_mag;
deltaP_1 = u_ijC_x * t0;
deltaP_2 = u_ijC_z * t0;
```

Reorder. In the base implementation, many values within the main loop are computed on every iteration, regardless of whether they are used in the taken execution path. Some computations also involve multiplications by impulses that may be zero depending on the execution path, meaning the velocities and other values stay the same despite performing these computations. These circumstances lead to unnecessary arithmetic operations.

To address this, the main loop was restructured by moving computations into the specific branches where their results are actually required. Expressions involving impulse terms were split up and moved such that they are only evaluated when the corresponding impulse is non-zero. For example, the *recmass* implementation includes the following velocity update at the end of the main loop:

```
v_ix += (deltaP_1 + deltaP_ix) * recM;
```

After reordering, the computation is split and relocated to the appropriate branches:

```
deltaP_1 = ...     // deltaP_1 is non-zero
v_ix += deltaP_1 * recM;
...
deltaP_ix = ...    // deltaP_ix is non-zero
v_ix += deltaP_ix * recM;
```

Finally, these transformations also allow us to eliminate the initialisation of impulses to zero at the start of each iteration.

Separate While Loops (sepwhile). As described in Section 2.1, the algorithm consists of a compression and restitution phase. In the base implementation, in every loop iteration, the current phase is determined in the loop condition and whether the restitution phase has started is determined at the end of the main loop, as shown in Algorithm 1.

To avoid these redundant condition evaluations, two separate `while`-loops are used instead. The first denotes the compression phase and ends at the start of the restitution phase. The second denotes the restitution phase and is executed until all work has been performed, i.e. $W \geq W_f$. The final work, W_f, is computed between the two `while`-loops and the `if`-statements at the end of the loop bodies

Algorithm 2 Schematic of sepwhile

```
Projecting global vectors into local reference frame
ΔP = ...                                  ▷ Dependent on N
while In Compression Phase do
    Update velocities, frictions, and work
end while
W_f = ...                  ▷ Sets final work, dependent on W
while W ≤ W_f do
    Update velocities, frictions, and work
end while
Project local vectors to global reference frame
return (angular) velocity vectors
```

are removed. A schematic of this implementation is shown in Algorithm 2.

Final Optimisations (double). The following three final optimisations are relatively minor and are thus combined into one implementation. First, the condition

```
if (u_iR_xy_mag < INFINITY) {
```

was found to always evaluate to true during the restitution phase, thus it was removed. Second, the work variable W is only ever compared to itself, which means the scaling factor `halfDeltaP` can be removed from the following statement:

```
W += halfDeltaP * fabs(v_ijy_old + v_ijy);
```

Finally, due to the separated `while`-loops, assumptions can be made about the sign of the variable `v_ijy`, which represents the relative velocity of the balls toward each other. This allows for the removal of `fabs()` in the following statements:

```
W -= (v_ijy_old + v_ijy); // first loop
...
W += (v_ijy_old + v_ijy); // second loop
```

Float. In addition to the algorithmic enhancements described above, we explored a variant where single-precision floating-point values (`floats`) are used instead of double-precision variables (`doubles`). This reduction in data size may allow the compiler to keep more values in registers when vectorisation is enabled, therefore reducing the data movement between cache and registers. Additionally, the type conversions between `double` and `float` surrounding the reciprocal square root instruction are no longer required.

However, `floats` only provide approximately seven decimal digits of precision, corresponding to a relative error of $\approx 10^{-7}$. 26 out of 204 test cases violate a 0.1% error threshold, increasing further as N increases, although relaxing the threshold to 1% reduces failures to just one test case. See Section 3.3 for further details on our validation methodology. Since using `floats` inherently leads to a drop in accuracy, this variant can be considered optional and is therefore treated separately in our later analysis.

3.2. Vector Optimisations

Vectorisation aims to improve performance by applying the same operation to multiple data elements simultaneously. In this section, we present our vectorised implementations, which use Intel intrinsics [13] to explicitly utilise packed SSE and AVX instructions.

Vectorised Double (double_vec). In an attempt to achieve a runtime improvement through the use of vector instructions, a vectorised version based on the double-precision scalar-optimised code was implemented first. This was done in three stages.

First, the entire function implementation was vectorised as well as possible using the appropriate intrinsics, while preserving the overall code structure, including conditional branches. In the second stage, all `if`-statements were replaced by `cmp_pd` statements, effectively removing all branches from the main loop body. Finally, the branch-free implementation was further refined by combining additional operations to improve performance.

Unfortunately, the nature of the algorithm limits the achievable vectorisation. Many scalar operations remain, due to the presence of standalone variables that are not involved in the same type of computations as others, as well as a high amount of data dependencies between intermediate values.

Vectorised Float (float_vec). In parallel, we developed a vectorised implementation based on the *float* implementation. In the first optimisation pass, vector intrinsics were used in the dot product, matrix vector multiplication, and cross product helper routines, while the main loop remained largely scalar. In the second pass, velocities and impulse components were packed into vectors and the entire loop body was vectorised. Here, similar challenges were faced as in the *Vectorised Double* implementation. In particular, no more than four values could be effectively packed into one vector, which is the reason only 128-bit vectors were used in the implementation.

3.3. Validation

Throughout the optimisation process, it was essential to continuously verify the validity of the implementations. This section outlines the methodology used to ensure the correctness of all optimisations.

The validation process involves comparing the output of a specific implementation against reference results for a given set of input parameters. For our purposes, the original *Pooltool* Python implementation serves as the ground truth. To construct a comprehensive test set, we collected 199 test cases from collisions within the *Pooltool* simulator. Additionally, five test cases, representing measured real-world

collisions, presented by Mathaven et al. [8] were used, resulting in 204 test cases total.

Outputs returned by our implementations were considered correct if they fell within 1% relative error of the output of the original implementation. Mathaven et al. [8] finds that the original algorithm exhibits relative errors of up to 12% when compared to real-world measurements. Therefore, we argue that optimisations resulting in relative errors of up to 1% do not meaningfully degrade the accuracy of the implementation with respect to reality.

4. EXPERIMENTAL RESULTS

This section presents the benchmarking setup and the results of testing the optimised implementations, along with a detailed analysis of the observed performance characteristics.

4.1. Experimental Setup

All performance tests were conducted on an AMD EPYC 7302 Processor, which is based on the *Zen 2* microarchitecture. The processor operates at a base frequency of 3.0 GHz, with a maximum dynamic frequency of up to 3.3 GHz [14]. *Zen 2* has an L1 cache size of 64 KiB, equally divided between instruction and data caches. As L1 cache is larger than the total size of the working set of the algorithm under study, the lower-level caches are irrelevant here. All tests run on a single core, since the algorithm is single-threaded.

Throughout development, the Clang compiler was primarily used, with the following compiler flags: `-O3`, `-march=native`, `-ffast-math`, and `-fno-finite-math-only`. The last compilation flag is required due to the presence of operations involving non-finite values. Unless stated otherwise, all results presented use this configuration. Notably, all speedup measures are relative to the initial implementation with this default configuration. Additional compiler flags were evaluated and are discussed in Section 4.4.

For benchmarking, the same 204 test cases as for validation (Section 3.3) are used. Each test case is executed 100 times, and the average of these runs is taken to eliminate minor variations in timing and overhead from the measurement code. The final result is then computed as the mean across all test inputs. The input parameter N, which roughly determines the number of iterations of the `while`-loop, is set to 1000, matching the value used by *Pooltool*.

Metrics are measured using PMCs as described in Section 2.2. To ensure accurate and consistent results, the benchmarking framework uses real-time scheduling and CPU affinity to a CPU core that was isolated using the `isolcpus` and `nohz_full` Linux kernel boot parameters. Additionally, Simultaneous Multithreading (SMT) was effectively disabled for the selected core to further reduce interference.

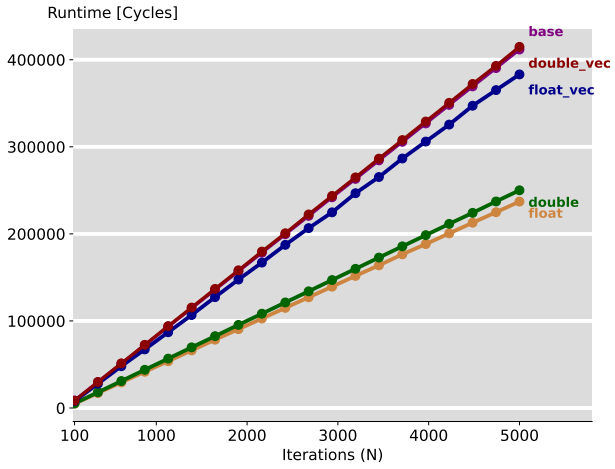

Fig. 2. The runtime of five selected implementations over a range of values for N.

4.2. Optimisation Results

Using the default compiler configuration described in Section 4.1, we present the results and observations from benchmarking several implementations. Generally, more focus is put on the runtime compared to the performance, as the runtime is more relevant to the purpose of the optimisations.

Figure 2 shows the runtime of selected implementations over a range of values for the input parameter N. The first observation is that all lines are perfectly linear. This is expected, as the majority of computations are performed in the `while`-loop, hence the runtime scales linearly with N. Additionally, each loop iteration performs the same operations on a small fixed working set. As a result, cache or memory aspects have no influence as N is varied. Overall, the scalar optimisations *double* and *float* achieve the highest speedups of approximately $1.65\times$ and $1.87\times$, respectively.

All implementations are discussed hereafter, in the same order as in Section 3, starting with the scalar optimisations, followed by the vectorised implementations. Figure 3 shows the cumulative speedup of all scalar implementations relative to the base implementation, in the order in which the optimisations were applied.

Reciprocal Square Root (rsqrt). The *rsqrt* implementation already achieves a significant speedup of $1.26\times$ as shown in Figure 3. Even with the overhead of converting between `float` and `double`, the runtime significantly improves over the base implementation. This is expected, as on *Zen 2*, the two conversions plus the reciprocal square root instruction have a total latency of 11 cycles, compared to 33 cycles for a double-precision square root and division operation [10]. The PMCs also show a drastic reduction in square root and division operations, as well as an increase in FMA operations, indicating the compiler was better able

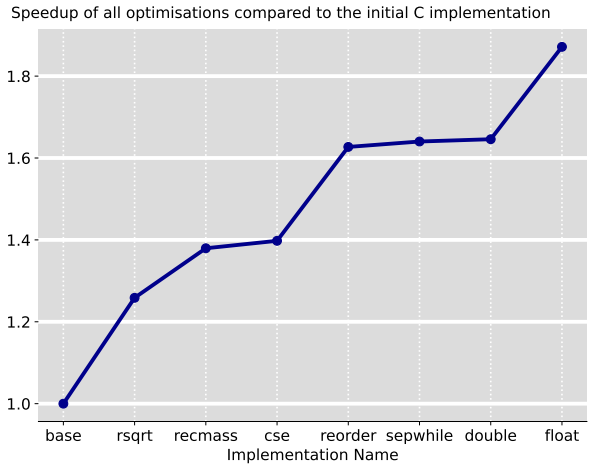

Fig. 3. Cumulative speedup over the base implementation for all scalar optimisations.

to apply other low-level optimisations.

Reciprocal Mass (recmass). The speedup improved further to $1.38\times$ for the *recmass* implementation as shown in Figure 3. PMCs interestingly showed that the number of division operations did not actually decrease further, meaning the compiler already performed this optimisation itself on the assembly level. However, the compiler again was able to better apply other low-level optimisations, resulting in a further increase in FMA instructions.

Common Subexpression Elimination (cse). The *cse* implementation only resulted in a minor additional speedup. This is expected as compilers are generally good at performing common subexpression elimination themselves.

Reorder. The reordering of computations led to a significant improvement, resulting in a total speedup of $1.63\times$, with respect to the base implementation as shown in Figure 3. PMCs showed a significant decrease in the number of FLOPs, which was the goal of the optimisation.

Separate While Loops (sepwhile). The *sepwhile* implementation achieved another minor speedup, as a result of a decrease in the number of instructions executed, measured by the PMCs. This aligns with the expected outcome of the optimisation, which eliminates some condition checks.

Final Optimisations (double). With the incorporation of the final minor optimisations, a cumulative speedup of $1.65\times$ was achieved. The final improvement is a result of a small decrease in the number of FLOPs as per the PMCs.

Float. Using `float` types instead of `double` types results in a cumulative speedup of $1.87\times$ as shown in Figure 3. This speedup is due to the removal of the conversions between single- and double-precision floats before and after the reciprocal square root intrinsic, which each add a latency of 3 cycles on *Zen 2*.

Vectorised Double (double_vec). As is evident from Figure 2, explicitly vectorising the optimised implementation using packed vector intrinsics leads to a huge increase in runtime. This result is due to the algorithm containing various execution branches and several variables that do not share the same type of computations as any other values, making the application of packed instructions difficult. Additionally, the algorithm has long and narrow dependency chains, with few independent computations that could be executed in parallel at any given time. Finally, data dependencies between computations often involve many shuffle and permute operations, adding additional overhead. All these complications lead to frequent scenarios where performing several independent scalar computations in parallel using instruction-level parallelism (ILP) is faster than data shuffling followed by a single packed floating-point operation.

Another goal of vectorisation is reducing memory and cache traffic, as more data can be fit into a single packed register. While measurements indicate that the number of load-store operations are reduced by a factor of almost five compared to the scalar variant, this does not outweigh the previously mentioned complications. As confirmed by PMCs, the working set of the algorithm is fully stored in L1 cache, and even though L1 cache accesses do add a few additional cycles, memory accesses do not happen frequently enough to have a major impact on runtime.

Vectorised Float (float_vec). Similarly to the *Vectorised Double* implementation, the single-precision variant could not improve the runtime over the scalar variant as shown in Figure 2. Here, the challenges faced are similar to those of the *Vectorised Double* implementation. Using single-precision floating-point values increases the complexity of vectorisation, as 8 values can fit in 256-bit registers, meaning eight operations need to be executed simultaneously for optimal performance. For these reasons, the implementation was also not able to improve the runtime.

4.3. Performance and Bound Analysis

Based on the runtime results of the implementations, the performance is computed and analysed below. In addition, bounds on the runtime and performance are established as a means of comparison.

Figure 4 shows the performance, in FLOPs per cycle, for five implementations as N increases. The performance largely stays constant, and changes only slightly for low values of N where the operations outside the loop have higher influence with decreasing number of iterations. Note that comparing the implementations based on performance is difficult as they differ in the number of FLOPs performed.

A lower bound on the runtime was established, based on the dependencies between FLOPs, which include additions, multiplications, divisions, square roots, and precision conversion, assuming the longest dependency chain.

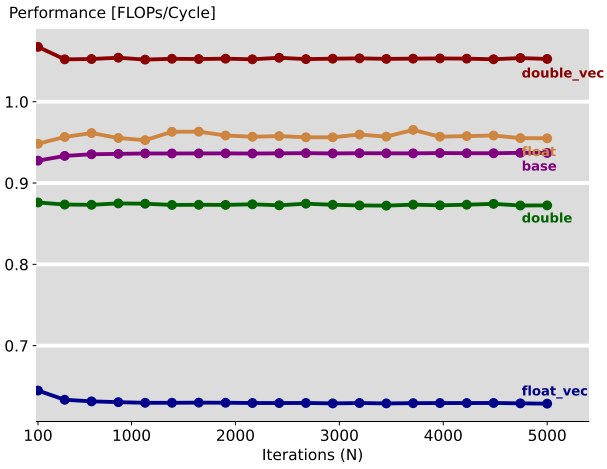

Fig. 4. The performance, in FLOPs per cycle, of selected implementations over a range of N values. Since all optimisations have vastly different operation counts, this figure does not serve as a comparison between implementations.

The lower bound for the runtime of the *base* implementation is roughly $77{,}000$ cyc for $N = 1000$ iterations, while for the fully optimized *double* implementation, this lower bound is $44{,}000$ cyc. With total FLOP counts of $77{,}796$ and $43{,}829$, this leads to a theoretical peak performance of approximately 1.01 and 0.99 FLOPs per cycle, respectively, for this algorithm. According to the measured performance, *base* and *double* implementations achieve 90.2% and 87.7% of this performance bound, respectively. However, this performance is still less than 16% of the theoretical peak performance of the *Zen 2* CPU, which is 6 FLOPs per cycle.

Moreover, these runtime lower bounds show that the implementations are close to optimal. The *double* implementation has a runtime of $50{,}200$ cyc for $N = 1000$, which is close to the lower bound of $44{,}000$ cyc, while *base* has a runtime of $85{,}300$ cyc and a lower bound of $77{,}000$ cyc. Actual runtime will always inevitably be higher than these bounds due to data movement and limited computation throughput, hence these results show that large further improvements can only come from algorithmic changes.

4.4. Other Compiler Configurations

As part of the evaluation, other compiler flags were tested to analyse the effects of enabling or disabling certain compiler optimisations. In addition to the default configuration described in Section 4.1, the following other compiler flag configurations were tested:

- *novec*: `-O3 -fno-tree-vectorize -fno-slp-vectorize`
- *o3*: `-O3`
- *archnative*: `-O3 -march=native`

We found that changing optimisation flags significantly changes the runtime differences between implementations. The *recmass* implementation with the *novec* compiler configuration, for instance, has a lower runtime compared to the final *double* implementation with the default configuration. This shows that enabling compiler vectorisation leads to a worse runtime in this case. According to PMC measurements, the difference in runtime mainly stems from a higher number of token stall cycles for the latter. Specifically, both cases have a high amount of cycles where there are not enough resources available to successfully schedule a floating-point computation, although for different reasons [9, sec. 2.1.15.4.4].

Likewise, we found that with *novec*, the *cse* implementation increases in runtime compared to *recmass*. Here, the previously mentioned types of token stall cycles are actually lower for *cse*. The culprit seems to be a stall type not recorded in the official documentation for our processor [9], however it is documented for another *Zen 2* processor [15, sec. 2.1.15], where it indicates cycles in which instructions were already executed but cannot yet be retired, due to conflicts and dependencies with previous instructions that have not finished.

There are many similar cases where token stall cycles are the main cause for runtime differences between compiler configurations as well as implementations. Overall, the fastest runtime was achieved with the *float* implementation using the *archnative* configuration on Clang, achieving a speedup of $1.93\times$. With another compiler, GCC, higher runtimes were observed in most cases. All these observations suggest that compilers, despite their complexity, are not able to consistently apply optimal optimisations.

5. CONCLUSIONS

A significant speedup of $1.93\times$ was achieved for Mathaven's billiard ball collision algorithm on the *Zen 2* microarchitecture through a series of structural and algorithmic optimisations. The improvements were achieved through incremental scalar optimisations applied to a direct C translation of the *Pooltool* Python implementation. While some optimisations introduced a slight reduction in numerical accuracy, all remained within a relative error margin of 1%. Attempts at vectorisation using vector intrinsics were unsuccessful, largely due to the branching structure and many data dependencies within the algorithm.

The results highlight the effectiveness of scalar optimisations in performance-critical, physics-based simulations. These results demonstrate promising directions for further research, particularly in exploring alternative parallelisation strategies or algorithmic modifications to better address the complications faced during optimisation of the algorithm.

6. CONTRIBUTIONS OF TEAM MEMBERS

Authors are always listed in alphabetical order according to their surname.

Joris. Did the *Reciprocal Square Root*, *Reciprocal Mass*, *CSE*, *Separate While Loops*, initial version of *Float*, and refinement of *Reorder* optimisations. Wrote initial version of code for creating plots and did performance counter analysis on all optimisations.

Gül. Did the *Float* and *Vectorised Float* optimisations. Collected and processed test input dataset. Wrote code for creating plots.

Wilton. Did the *Reorder*, *Final Minor Optimisations*, and *Vectorised Double* optimisations. Developed benchmarking / performance analysis and testing framework. Also did performance analysis of optimisations, and wrote a bit of code for plots.

7. REFERENCES

[1] Pro Billiard Series, "Pbs kicks off season with record-breaking viewership," Mar. 2024, Last accessed on the 16th of June 2025.

[2] G. Coriolis (1792-1843), "Théorie mathématique des effets du jeu de billard," *Carilian-Goeury*, 1835.

[3] R Atladottir, J Gay, Kasper Løvborg Jensen, RB Jensen, I Lontis, LB Larsen, and S Larsen, "Multi modal interaction in an automatic pool trainer," in *Proceedings of the Second Nordic Conference on Multimodal Communication*. 2005, p. 111, Göteborg University.

[4] Fei Long, J. Berland, M.-C. Tessier, D. Naulls, A. Roth, G. Roth, and M. Greenspan, "Robotic pool: an experiment in automatic potting," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, Sendai, Japan, 2004, vol. 3, p. 2520–2525, IEEE.

[5] Jean-Pierre Dussault and Jean-François Landry, "Optimization of a Billiard Player – Tactical Play," in *Computers and Games*, H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers, Eds., Berlin, Heidelberg, 2007, p. 256–270, Springer.

[6] Baljinder S. Bal and Gaurav Dureja, "Hawk Eye: A Logical Innovative Technology Use in Sports for Effective Decision Making," in *Sport Sci Rev*, 2012.

[7] Evan Kiefl, "Pooltool: A python package for realistic billiards simulation," *Journal of Open Source Software*, vol. 9, no. 101, pp. 7301, 2024.

[8] S. Mathavan, M. R. Jackson, and R. M. Parkin, "Numerical simulations of the frictional collisions of solid balls on a rough surface," *Sports Engineering*, vol. 17, no. 4, pp. 227–237, Dec. 2014.

[9] Advanced Micro Devices Inc, "Processor Programming Reference (PPR) for AMD Family 17h Model 31h, Revision B0 Processors," AMD Documentation Hub, Advanced Micro Devices, Inc, Sept. 2020.

[10] Agner Fog, "Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs," Technical report, Technical University of Denmark, Nov. 2022, Last accessed on the 16th of June 2025.

[11] Rys for Consumer Graphics, "Origin of Quake3's Fast InvSqrt() - Part Two," Dec. 2006, https://www.beyond3d.com/content/articles/15/, Last accessed on the 16th of June 2025.

[12] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, M–U," Technical Reference Manual Volume 2B, Intel® Corporation, 2020, Section 4-594.

[13] Intel Corporation, "Intrinsics - Intel® C++ Compiler Classic Developer Guide and Reference," 2021, https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/intrinsics.html, Accessed 2025-06-18.

[14] Advanced Micro Devices Inc, "AMD EPYC™ 7002 Series Processors," 2018, https://www.amd.com/en/products/processors/server/epyc/7002-series.html, Accessed 2025-06-18.

[15] Advanced Micro Devices Inc, "Processor Programming Reference (PPR) for AMD Family 17h Model 71h, Revision B0 Processors," AMD Documentation Hub, Advanced Micro Devices, Inc, Sept. 2019.