PERFORMANCE OPTIMIZATIONS FOR A SOCIAL FORCE MODEL IMPLEMENTATION

Quy Anh Nguyen, Linus Baumberger, Smail Alijagic, Ahmed Kallala

Department of Computer Science
ETH Zürich, Switzerland

ABSTRACT

We present an optimized single-core implementation of the Helbing-Molnár social force model for x86 machines. Starting from a baseline C++ implementation, we iteratively applied performance optimizations including inlining, data layout reorganization, vectorization, instruction-level parallelism, and approximation techniques. We evaluated our optimizations on an x86 machine supporting AVX-512, achieving a maximum speedup of 18 over the baseline. This paper documents our optimization process and performance results.

1. INTRODUCTION

Motivation. The social force model, introduced by Helbing and Molnár [1], simulates pedestrian dynamics by taking into account various influencing factors. Its accuracy in predicting crowd behavior makes the model a useful urban design tool, particularly when high-traffic spaces such as shopping malls or transit hubs are planned. The algorithm is computationally expensive. Its asymptotic runtime complexity is $\Theta(n_{iter} \cdot {n_p}^2)$. It scales quadratically with the number of pedestrians, making realistic simulations of crowded areas computationally expensive.

Contributions. In this project, we implemented the social force model and optimized its performance. Our contributions are as follows:

- **Baseline Implementation:** We implemented the social force model as presented by Helbing and Molnár [1]. Particularly, our implementation simulates pedestrian dynamics in the same original setting: a horizontal rectangular corridor bounded by two parallel walls along the x-axis at the top and bottom, with open boundaries on the left and right sides. The corridor is 50 meters in length and 10 meters in width, with pedestrians moving in either horizontal direction.
- **Optimizations:** We systematically optimized the computations by removing optimization blockers, improving the data layout, vectorizing the code, exploiting instruction-level parallelism (ILP), and performing strength reduction of mathematical operations.
- **Benchmarks**: We evaluated our optimizations on a x86 machine that supports AVX-512 and presented our results below. At 5000 pedestrians and 200 simulation rounds, we achieved a maximum speedup of 18 compared to our baseline implementation.

2. BACKGROUND ON THE ALGORITHM

The Social Force Model simulates pedestrian dynamics by modeling human interactions as physical forces acting on each individual [1]. Similar to the original paper, we implemented a stripped-down version of the model that captures the essential dynamics while excluding attractive forces and random fluctuations from consideration. Each pedestrian α experiences an acceleration force towards their desired destination, a repulsive force from every other pedestrian β, and a repulsive force from each obstacle. In our implementation, the walls at the top and of the corridor are the only two obstacles.

Acceleration Term. By default, each pedestrian has a preferred speed v_α^0 at which they want to move towards their desired destination. The acceleration term captures the force that the pedestrian needs to perform in order to match their current velocity with their desired velocity in case the two differ. It is calculated as

$$F_\alpha^0(\vec{v}_\alpha, v_\alpha^0 \vec{e}_\alpha) := \frac{1}{\tau_\alpha} \left(v_\alpha^0 \vec{e}_\alpha - \vec{v}_\alpha \right) \tag{1}$$

with $\vec{v}_\alpha$ being the velocity of pedestrian α, τ_α being the relaxation time (how long it takes for the pedestrian to again reach their desired speed), and $\vec{e}_\alpha$ being the direction towards the desired destination from the current position.

$\vec{e}_\alpha$ is calculated as

$$\vec{e}_\alpha := \frac{\vec{r}_\alpha^0 - \vec{r}_\alpha}{\|\vec{r}_\alpha^0 - \vec{r}_\alpha\|} \tag{2}$$

with $\vec{r}_\alpha^0$ being the desired destination of α and $\vec{r}_\alpha$ being their current position vector.

We set τ_α to 0.5 seconds at all times for all pedestrians.

Pedestrian Repulsion Term. When another pedestrian β approaches pedestrian α, a repulsive force appears. This force depends on the current position of β relative to α and the anticipated position of β given their current velocity and direction of motion:

$$\vec{f}_{\alpha\beta}(\vec{r}_{\alpha\beta}) := -\nabla_{\vec{r}_{\alpha\beta}} V_{\alpha\beta}[b(\vec{r}_{\alpha\beta})] \tag{3}$$

$V_{\alpha\beta}$ is a monotonic decreasing potential, and $b(\vec{r}_{\alpha\beta})$ is the semi-minor axis of an ellipse oriented in the direction of β's motion. Like the original paper, we assume that $V_{\alpha\beta}(b) = V_\alpha^0 e^{-b/\sigma}$, and therefore

$$\vec{f}_{\alpha\beta}(\vec{r}_{\alpha\beta}) := \frac{V_\alpha^0}{\sigma} e^{-b/\sigma} \nabla_{\vec{r}_{\alpha\beta}} b \tag{4}$$

with V_α^0 and σ being constants.

We also assume that

$$b := \frac{1}{2}\sqrt{\left(\|\vec{r}_{\alpha\beta}\| + \|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|\right)^2 - (v_\beta \Delta t)^2} \tag{5}$$

like original paper, and thus

$$\begin{aligned}\nabla_{\vec{r}_{\alpha\beta}} b := &\frac{\|\vec{r}_{\alpha\beta}\| + \|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|}{4b\,\|\vec{r}_{\alpha\beta}\|}\,\vec{r}_{\alpha\beta} \\ &+ \frac{\|\vec{r}_{\alpha\beta}\| + \|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|}{4b\,\|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|}\,(\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta)\end{aligned} \tag{6}$$

with $\vec{r}_{\alpha\beta} = \vec{r}_\alpha - \vec{r}_\beta$, v_β being the current speed of β and Δt being the time step used to anticipate the future position of β.

Obstacle Repulsion Term. When α approaches an obstacle B, a repulsive force also appears to prevent collision:

$$\vec{F}_{\alpha B}(\vec{r}_{\alpha B}) := -\nabla_{\vec{r}_{\alpha B}} U_{\alpha B}(\|\vec{r}_{\alpha B}\|) \tag{7}$$

$\vec{r}_{\alpha B} = \vec{r}_\alpha - \vec{p}_{\alpha B}$, with $\vec{p}_{\alpha B}$ being the point on obstacle B closest to α, while $U_{\alpha B}$ is another monotonic decreasing potential. We assume $U_{\alpha B}(\|\vec{r}_{\alpha B}\|) = U_\alpha^0 e^{-\|\vec{r}_{\alpha B}\|/R}$ like the original paper, and therefore

$$\vec{F}_{\alpha B}(\vec{r}_{\alpha B}) := \frac{U_\alpha^0}{R} e^{-\|\vec{r}_{\alpha B}\|/R} \frac{\vec{r}_{\alpha B}}{\|\vec{r}_{\alpha B}\|} \tag{8}$$

with U_α^0 and R being constants.

Total Force. Aggregating the above three terms gives us the total social force exerted on each pedestrian α:

$$\begin{aligned}\vec{F}_\alpha := F_\alpha^0 &+ \sum_\beta w(\vec{e}_\alpha, -\vec{f}_{\alpha\beta})\vec{f}_{\alpha\beta} \\ &+ \sum_B w(\vec{e}_\alpha, -\vec{f}_{\alpha B})\vec{F}_{\alpha B}\end{aligned} \tag{9}$$

The total social force acting on pedestrian α is the sum of the acceleration towards the desired destination and the repulsion forces from all other pedestrians and obstacles. To weaken the contributions of pedestrians and obstacles outside α's field of view towards the social force, the direction-dependent correction factor $w(\vec{e}_\alpha, \vec{f})$ is introduced:

$$w(\vec{e}, \vec{f}) := \begin{cases} 1 & \text{if } \vec{e} \cdot \vec{f} \geq \|\vec{f}\| \cos\varphi \\ c & \text{otherwise.} \end{cases} \tag{10}$$

2φ is α's angle of view. The inequality checks whether the angle between $\vec{e}_\alpha$ and $\vec{f}$ remains below φ, i.e., whether a pedestrian or obstacle is within α's field of view. If a pedestrian or obstacle is outside α's field of view, their contribution towards α's social force is multiplied by a damping factor c between 0 and 1.

State update. Given the social force, we can directly calculate the acceleration of pedestrian α as influenced by external factors:

$$\frac{d\vec{w}_\alpha}{dt} = \vec{F}_\alpha(t) \tag{11}$$

$\vec{w}_\alpha$ denotes α's preferred velocity, which we get with a forward-Euler step:

$$\vec{w}_\alpha(t) = \vec{w}_\alpha(t-1) + \vec{F}_\alpha(t)\,dt \tag{12}$$

We cap $\vec{w}_\alpha$ at α's maximum attainable speed $v_\alpha^{\max}$ to obtain the actual realized velocity

$$\vec{v}_\alpha(t) = g(\vec{w}_\alpha(t), v_\alpha^{\max})\ \vec{w}_\alpha(t) \tag{13}$$

with g being the clamping factor:

$$g(\vec{w}_\alpha, v_\alpha^{\max}) := \begin{cases} 1 & \text{if } \|\vec{w}_\alpha\| \leq v_\alpha^{\max}, \\ \frac{v_\alpha^{\max}}{\|\vec{w}_\alpha\|} & \text{otherwise.} \end{cases} \tag{14}$$

Finally, applying a second Euler step on $\vec{v}_\alpha$ yields α's new position:

$$\vec{r}_\alpha(t) = \vec{r}_\alpha(t-1) + \vec{v}_\alpha(t)\,dt \tag{15}$$

3. OPTIMIZATIONS PERFORMED

In this section, we describe the sequence of optimizations applied to the simulation loop. Although our optimizations might cause minor changes to the flop count, the algorithmic complexity of $\Theta(n_{iter} \cdot n_p^2)$ remains unchanged.

Baseline. First, we began by writing a baseline implementation of the simulation. For this baseline, we borrowed the utility struct `Vector2d` that another group implemented in a previous iteration of this course [2] to represent a point on a 2D plane. This struct stores the x-axis and y-axis

coordinates of a point as **double-precision floating point numbers (doubles)**. The struct also contains methods for performing arithmetic operations with its coordinates (addition, subtraction, multiplication, and division). In addition to methods for calculating the distance between two points and their dot product, we further implemented the utility classes `Pedestrian`, `Obstacle`, and `Board` on top of `Vector2d`. Each `Pedestrian` stores the current position, current velocity, desired destination, desired direction (from the current position towards the desired destination), desired velocity (in the desired direction), a desired speed, and a maximum speed. Each `Obstacle` is treated as a line with starting and ending coordinates. A `Board` is a rectangular 2D plane consisting of multiple `Obstacle`s. In our implementation, there are two obstacles: the upper and lower walls of the board. The list of pedestrians in the baseline implementation is stored as an **array of structures (AoS)**.

Algorithm 1: Baseline

Data:
Number of pedestrians n_p
Number of total iterations n_{iter}
Result:
Location of each pedestrian in each iteration r_i^t

```
Initialize v_i^0, r_i^0, e_i, v_i^max, v_i^0
for t ← 1 to n_iter do
    for i ← 1 to n_p do
        α ← pedestrians[i]
        Compute acceleration term:
          F ← acceleration(α)
        for j ← 1 to n_p do
            if i = j then continue
            β ← pedestrians[j]
            Compute pedestrian repulsion term:
              F += repulsion(α, β)
        for obstacle in obstacles do
            Compute obstacle repulsion term:
              F += repulsion(α, obstacle)
        Update: α.update_states(F)
```

We implemented separate functions for calculating the acceleration term, the pedestrian repulsion term, the obstacle repulsion term, and the correction factor. The first three correspond to `acceleration(alpha)`, `repulsion(alpha, beta)`, and `repulsion(alpha, obstacle)` in the pseudocode, while the last one corresponds to $w(\vec{e}, \vec{f})$ in (10). Within these functions, all arithmetic operations are performed on a **per-scalar** basis, i.e., there are no vectorized instructions or SIMD intrinsics. Additionally, the exponential and square root functions (`exp()` and `sqrt()`) are imported directly from the **Standard C++ Library** via the `#include <cmath>` directive. The baseline implementation aims to match the original formulas as closely as possible and as such contains no optimization.

We subsequently defined the following cost measure for our program:

$$\begin{aligned} C(n_{iter}, n_p) = &\text{adds}(n_{iter}, n_p) \cdot C_{add} \\ &+ \text{mults}(n_{iter}, n_p) \cdot C_{mult} \\ &+ \text{divs}(n_{iter}, n_p) \cdot C_{div} \\ &+ \text{sqrts}(n_{iter}, n_p) \cdot C_{sqrt} \\ &+ \text{exps}(n_{iter}, n_p) \cdot C_{exp} \end{aligned} \tag{16}$$

`adds`, `mults`, `divs`, `sqrts`, and `exps` are the numbers of the respective instruction in the entire program. The costs of all operations are set to 1 flop except for C_{exp}, which is set to 30 flops as per the course guideline.

With this cost measure, we calculated the cost of the baseline implementation to be $n_{iter} * (n_p \cdot n_p \cdot 72 + n_p \cdot 74)$ flops. Subsequent optimizations might cause the flop count to deviate from this figure. Thus, we manually recalculated the flop count whenever we added a new optimization to our program in order to derive the correct performance measurements. The performance of the baseline implementation is 1.15 flops/cycle at 5000 pedestrians and 200 iterations. Throughout this report, whenever we mention the performance of a particular version, it should be understood to mean the performance measured at 5000 pedestrians and 200 iterations, unless otherwise indicated.

Inlining. In order to later vectorize the program, it is necessary to get rid of the `Vector2d` structs and apply arithmetic operations on the doubles directly instead of via `Vector2d`'s methods. Moreover, as function calls might contain side effects, compilers tend to be conservative and treat them as a black box [3]. Thus, we might incur overhead due to the functions not being automatically inlined. In our case, the functions mentioned in the previous section were indeed not inlined, which we confirmed by inspecting the generated assembly. We manually inlined these functions as well as the utility methods provided by `Vector2d` and got rid of the structs afterwards. Subsequently, we were also able to get rid of some duplicate calculations by reusing common subexpressions. Inlining only increased performance by roughly 3.6%, yet it is nevertheless a crucial prerequisite for vectorization.

Structure of Arrays (SoA). Having gotten rid of the `Vector2d` structs, the data layout of our program was still Array of Structures (AoS). This means that we stored all the states of each pedestrian (current position, current veloc-

ity, and so on) contiguous in memory. This is not ideal for vectorization, as it means that we cannot contiguously load the same state of different pedestrians into one vector. We changed the data layout to **Structure of Arrays (SoA)**. We used one array of doubles to store the x-axis coordinates of all pedestrians, another array of doubles to store the y-axis coordinates of all pedestrians, and so on. This way, when we later perform vectorization, we can load contiguous floating-point numbers from memory into the same vectors.

AVX-256. The program was scalar and performed only a single pedestrian–pedestrian interaction per j-loop iteration, without taking advantage of the vector processing capabilities of the Floating-Point Units (FPU). As the j-loop dominated the runtime ($\Theta(n_p^2)$) and the SoA layout already stored every field with unit stride, it was a direct fit for vectorization. We processed the j-loop four lanes at a time on `__m256d` registers, computing four interactions per step. All arrays that stored pedestrian states were padded to a multiple-of-four length so that the unrolled j-loop left no scalar remainder. The padding slots contained states of dummy pedestrians placed far away (10^9 meters away from the board), so distant that the exponentially decaying repulsion force they exerted on the current pedestrian was effectively zero. To be extra safe, we masked their force contributions away when updating the accumulation vectors.

Two branches were removed to prevent blocking the vectorization (the `i == j` self-interaction skip and the vision-cone test $\vec{e}\cdot\vec{f} \geq \|\vec{f}\|\cos\varphi$ determining the weight w). We instead implemented dual-use masks that zero out the self-interaction entry as well as entries corresponding to padded β values in the two force vectors (one for forces to be added to the x-axis force accumulator vector and the other for forces to be added to the y-axis force accumulator vector). The $\alpha = \beta$ lane collapses the force denominator to zero, so we set that lane of the denominator vector to 1.0 using `_mm256_blendv_pd` to keep the division results finite. We applied the masks to the force vectors using `_mm256_and_pd` before they were added to the accumulators. The field-of-view weight $w(\vec{e}, \vec{f})$ of Equation 10 was also evaluated in a branchless manner. The vision-cone condition consisting of two individual comparisons was implemented as another mask vector. The lane values of the weight vector were set to either c or 1 based on this mask using `_mm256_blendv_pd`, which also subsumed the integration step. The two accumulator vectors were each reduced with a single horizontal sum per α, i.e., once per i-loop iteration rather than per j-loop iteration, incurring minimal costs. The obstacle loop was left scalar, since it only had two iterations per pedestrian and therefore scaled linearly rather than quadratically.

The vectorization gave a speedup of 2.9 compared to the SoA version (Section 4), increasing the performance from 1.06 to 3.09 flops/cycle. The gain fell short of the ideal $4\times$ of four-wide FP64 because the exponential in the potential was still computed by four *scalar* `exp()` calls into a scratch array per j-loop iteration. With everything around them vectorized, these serial scalar library calls became the main bottleneck of the inner loop and were evidently our next target.

exp() Approximation (PolyExp).
Profiling confirmed that the four scalar `exp()` calls per iteration dominated the inner loop. Not only did they waste the vector processing capabilities of the FPUs, they also incurred unnecessary overhead from jumps. We replaced them with a vectorized approximation function operating directly on the SIMD registers using the standard range reduction $\exp(x) = 2^n \exp(r)$ with $n = \mathrm{round}(x/\ln 2)$ and $r = x - n\ln 2$, where n comes from a round-to-nearest and r from one `fnmadd`, leaving $r \in [-\frac{\ln 2}{2}, \frac{\ln 2}{2}] \approx [-0.35, 0.35]$. On this interval, $\exp(r)$ is evaluated by a degree-13 Taylor polynomial (coefficients $1/k!$) in Horner form, i.e., thirteen fused multiply-adds. Its truncation error is bounded by $r^{14}/14! \lesssim 6 \times 10^{-18}$ at the range edge, which is well below both double- and single-precision rounding. The same polynomial keeps `exp` exact to working precision both in this FP64 kernel and in the FP32 kernel of the next step, and `exp` is never the accuracy bottleneck over the thousands of time steps of the simulation. 2^n, on the other hand, can be calculated cost-free thanks to the way floating-point numbers are represented in IEEE-754 [4]: cast n to an integer, add the bias, shift the bits leftwards into the exponent field ($(n + 1023) \ll 52$ for doubles, $(n + 127) \ll 23$ for floats), and then reinterpret the result. As such, no expensive `pow()` or other library calls are needed.

Altogether, the approximation function performs 16 multiplies and 14 adds per lane (two multiplies for the $x/\ln 2$ scaling and the final recombination, one fnmadd, and thirteen Horner fmadds), exactly the 30-flop cost we assigned to `exp`. The round-to-nearest and the integer steps of the 2^n bit trick (a conversion, a bias add, and a shift) are not floating-point operations and do not contribute to the total flop count. We also used the `inline` keyword to ensure the approximation function did not result in extra jumps. With no branching, library calls, or scratch arrays, the optimization delivered a 29% performance increase, from 3.09 to 3.99 flops/cycle (Section 4).

After this optimization, the inner loop no longer contained any transcendental function or library calls and was bounded by the remaining FP arithmetic: the FMA and sqrt-div dependency chains. At this point, our program ran entirely in FP64, carrying ~ 16 significant digits, even though the

model is only validated to 1-cm precision. Our implementation was more precise than necessary, and our next optimization traded some of this precision for more speed.

Precision Reduction. As the social force model is a simulation, it is by definition an approximation of reality. We decided to trade numerical accuracy for speed by switching to **single-precision floating-point number (floats)**. Since each float (32 bits) is half the size of a double (64 bits), we can store twice as many floats as doubles in the same register. Consequently, twice as many calculations per CPU instruction can be performed. We implemented this optimization by writing another baseline implementation using floats instead of doubles and applying the same optimizations (`Inlining -> SoA -> AVX-256 -> exp() Approximation`). Following the switch, we recorded a speedup of 2.2, from 3.99 to 8.64 flops/cycle. The speedup is slightly above 2 because this version still issued scalar libm `sqrt()` calls outside the j-loop, and `sqrt()` is faster for single-precision than double precision inputs as the former require fewer correction rounds. As such, we got a small extra boost in performance in addition to the 2x that we got from doubling the lane count.

AVX-512. AVX-512 vectors are twice the size of AVX-256 vectors. Therefore, in principle, we should be able to perform twice as many flops per instruction with AVX-512 than with AVX-256. We ported our program to AVX-512 after vectorizing the j-loop (including the scalar `exp()` calls) with AVX-256. Since most of the AVX-256 intrinsics we used also have a direct AVX-512 equivalent, the switch was straightforward. We only had to change the data type from `__m256` to `__m512` and rename the intrinsics, sometimes reordering their arguments. Following the switch, we recorded a speedup of 1.66, from 8.64 to 14.43 flops/cycle. We likely did not get a speedup of 2 because our machine's CPU has a dual-pumped 256-bit FPU data path instead of an actual 512-bit datapath. Nevertheless, the speedup was significant.

Instruction-Level Parallelism (ILP). To achieve optimal performance, we need to fully utilize the execution units of our machine. Particularly, on a single core of our machine, there are two pipes that have an FMA execution unit, and we want these two pipes to be always running. However, in our vectorized `exp()` approximation implementation (`_avx512_exp_ps()`), we have a long chain of dependent FMA operations. Therefore, if our superscalar processor runs only one `_avx512_exp_ps()` (or rather, its compiled instructions) at a time, one FMA-capable pipe would remain idle. Thus, we increased our j-loop unrolling factor L from 16 to 32 and ensured that we had two independent `_avx512_exp_ps()` calls (and therefore, two separate FMA dependency chains that could execute in parallel). Unexpectedly, this optimization only increased the performance from 14.06 to 15.57 flops/cycle. This is likely due to some other dependencies not being fully untangled, thereby preventing true parallelism from taking place.

RSQRT. The bottleneck of our program at this point was the inverse square root calculations, which involved executing a div serially after a square root, either with scalar libm calls or with SIMD intrinsics. We needed the inverse square roots for $\|\vec{r}_{\alpha\beta}\|$, $\|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|$, and $2b$. We subsequently switched to the rsqrt intrinsic (`_mm512_rsqrt14_ps` in AVX-512) whenever we needed to calculate the inverse square root. The `_mm512_rsqrt14_ps` intrinsic utilizes the hardware `vrsqrtps` instruction, accurate to about 14 mantissa bits. This is faster because the rsqrt instruction generally has much lower latency and higher throughput than the sqrt instruction (Table 1). If we need the actual sqrt, we can recover it with an additional mult, since $\sqrt{x} = x \cdot \mathrm{rsqrt}(x)$. Additionally, having the reciprocal square roots at hand allowed us to get rid of the divisions algebraically: Let $r_1 = \|\vec{r}_{\alpha\beta}\|$, $r_2 = \|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|$, and $r_3 = 2b$, with $\hat{r}_1$,$\hat{r}_2$, and $\hat{r3}$ denoting their reciprocals. Using the rsqrt intrinsic, we can calculate the three reciprocals. Afterwards, the original scaling factor $\mathrm{smc} = \mathrm{num}/(2b\, r_1\, r_2)$ can be calculated as:

$$\mathrm{smc} = (\mathrm{smc} \cdot r_1) \cdot \hat{r}_1 = (\mathrm{smc} \cdot r_2) \cdot \hat{r}_2 = \mathrm{num}\, \hat{r}_1 \hat{r}_2 \hat{r}_3$$

without involving any division. This resulted in a 29% performance increase, from 15.57 to 20.14 flops/cycle.

Other Optimizations.

- **Incremental Index Vectors.** Rather than rebuilding the lane-index vectors every iteration by broadcasting the scalar loop counters, we initialized them once before the respective loop and updated the index vectors by adding them with an offset vector every iteration.
- **Native AVX-512 Masked FMA.** Instead of masking out lanes of operands that should not contribute to the accumulator and then separately issuing an FMA to update the latter, we switched to AVX-512's masked FMA, which fuses the mask and the FMA into a single instruction. The result is identical but computed more efficiently with one fewer operation:

```
        _mm512_fmadd_ps
(_mm512_maskz_mov_ps(m, a), b, c)
              ⇓
_mm512_mask3_fmadd_ps(a, b, c, m)
```

- **Outer-Loop Unrolling.** We moved the SIMD parallelism from the inner loop to the outer loop. Previously, the inner j-loop was vectorized to process 16

lanes at a time while the outer i-loop processed one pedestrian α at a time. This did not result in significant performance increase, so we instead swapped the loops, processing 16 pedestrians at a time for the outer loop and one pedestrian at a time for the inner loop. This allowed us to vectorize all the work that was previously scalar, namely the acceleration and obstacle-repulsion terms: their costly `sqrt`, `exp` and `div` operations were vectorized sixteenfold, increasing their throughput and enabling us to replace `sqrt`/`div` with the faster `rsqrt`. The new layout also removed overhead: as the 16 partial forces now lived in independent lanes, we no longer needed the horizontal reduction (and its expensive shuffle instructions) to collapse them. The in-range mask were also removed: padded pedestrians now occupy lanes of the outer vector, so their dummy data stays confined to those lanes and can simply be ignored, leaving only the unavoidable $\alpha \neq \beta$ self-interaction mask.

- **Flush to Zero / Denormals are Zero.** In AMD Zen 5, floating-point operations resulting in subnormals might take a few extra clock cycles [5]. To avoid this penalty, we simply flushed such values to zero directly by setting the FTZ and DAZ flags in the MXCSR register.

- **Look-up Table for exp() + Hermite Approximation.** Since the pedestrian repulsion and obstacle repulsion terms require expensive exponential calculations, we experimented with a lookup table filled with 69'000 precomputed values ranging from -40.0 to 0.0 at compile time. At runtime, the final result was linearly approximated. This reduced the flop count for `exp` by $n_{iter} \cdot [n_p \cdot (n_p - 1) + n_p \cdot n_{obstacles}]$. However, linear approximation was imprecise, so we later switched to Cubic Hermite Interpolation, which allowed us to maintain 1-cm precision. This method depended on a huge lookup table, potentially incurring many cache misses. Thus, we did not follow through with this approach for the final version.

- **Far-Away Pedestrians Cutoff.** Since the repulsion force between two pedestrians that are far away is minimal (close to zero), we had the idea of skipping the repulsion force calculations entirely if the distance between two pedestrians is above a certain threshold. However, since we have no means of deriving the flop count after applying this optimization, it was not included in the end.

- **Distance Symmetry.** We realized that we were calculating the distance between each pedestrian pair twice, each time requiring one costly sqrt operation. Therefore, we thought we could avoid this recomputation by implementing a lookup table. Prior to implementing the lookup table, it had taken us $n_{iter} \cdot n_p \cdot (n_p - 1) \cdot 6$ flops to calculate the distance between every two pedestrians. After implementing the lookup table, the figure dropped to $n_{iter} \cdot n_p \cdot \frac{n_p - 1}{2} \cdot 6$ flops. However, relying on a lookup table made it difficult to vectorize our code, so in the end, we removed it.

- **Porting x86 AVX to ARM NEON.** As a bonus experiment, we implemented an optimized SIMD version using NEON intrinsics instead of AVX to explore what optimizations NEON might offer. While NEON provides some efficient instructions unavailable in AVX, such as the horizontal add `vaddvq_f32(v)`, its vector width is limited to 128 bits, far below that of AVX-512. As a result, the achieved speedup was much lower than the one realized with AVX.

4. EXPERIMENTAL RESULTS

Experimental setup. We evaluated our optimizations on the following AVX-512-capable machine:

```
Processor: AMD Ryzen AI 7 350
Memory: 32 GiB (30.6 GiB usable)
Socket(s): 1
Core(s) per socket: 8
Thread(s) per core: 2
Frequency: 2.0 GHz
Caches (sum across all cores):
  L1d: 384 KiB (8 instances)
  L1i: 256 KiB (8 instances)
  L2: 8 MiB (8 instances)
  L3: 16 MiB (1 instance)
Microarchitecture: AMD Zen 5
Operating System: Linux (6.19.14+kali-amd64)
```

Frequency refers to the chosen frequency that we capped all cores to using the `cpupower` tool.

We fixed n_{iter} at 200 and progressively tested out n_p at 50, 100, 200, 500, 1000, 2000 and 5'000. For vectorized versions, we also tested out $n_p = 10'000$. For each n_p-n_{iter} configuration, we performed 3 warmup iterations of the entire program, followed by an additional calibration iteration. Then, we ran the simulation 15 times, measuring the cycle count for each run using the `RDTSC` instruction. The median of these 15 cycle counts was chosen. The entire process is repeated 3 times for each version. We then averaged the 3 median values to be the final cycle count of the version. For each program version, we manually inspected the code to derive the actual count of each instruction. We then calculated the flop count according to the defined cost measure (Equation 16). The final performance was then calculated as W/T.

On a single core of our machine, the latency (cycles) and throughput (instructions/cycle) for each single-precision vector floating-point instruction are as follows, according to uops.info:

Instruction	Latency	Throughput	Ports
VADDPS	2	2	FP2/3
VMULPS	3	2	FP0/1
VFMADD132PS	4	2	FP0/1
VDIVPS	10	0.4	FP1
VSQRTPS	14	0.22	FP1
VRSQRT14PS	5	2	FP1/2

Table 1. **Instruction Latency and Throughput**

In our very best vectorized version, each j-loop iteration performs 23 FMAs, 19 MULTs, 6 ADDs, and 3 RSQRTs per lane, adding up to $23 \cdot 2 + 19 + 6 + 3 \cdot 2 = 77$ flops. Thus, the runtime is dominated by the FMAs and the MULTs, which can only be executed on floating-point ports 0 and 1. Thus, per j-loop iteration, $T \geq (23 + 19)/2) = 21$ cycles. The scalar peak performance based on the instruction mix is then $77/21$ flops/cycle. Given that we vectorized our code with AVX-512 (16 floats per vector), the peak performance when the program is vectorized would be $77/21 \cdot 16$, but since the machine only has a 256-bit instead of 512-bit datapath, we have to divide this number by 2. Thus, the actual peak performance based on the instruction mix of our program is $77/21 \cdot 16/2 = 29.33$ flops/cycle. The actual hardware bound would be higher at $(2 \cdot 2 + 2) \cdot 16/2 = 48$ flops/cycle, since we have 2 ports for FMAs (FP0/1) and another 2 ports for ADDs (FP2/3).

Results. The plot below shows the performance of the program when different optimization combinations are applied.

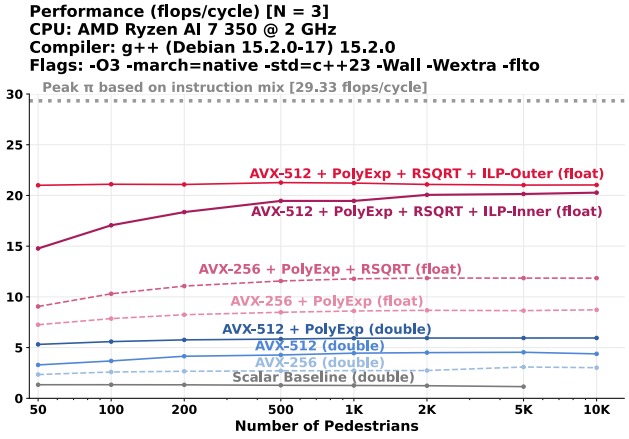


Fig. 1. Performance of different optimization combinations.

As seen in the figure, unrolling the outer loop for ILP gave a significantly larger speedup compared to unrolling the inner loop at small values of n_p. The reason is that as n_p grows, the $\Theta({n_p}^2)$ inner-loop dominates the runtime, while the outer-loop computations become negligible in comparison. Thus, optimizations to the outer loop, namely removing horizontal reductions and vectorizing residual scalar instructions, are more impactful at small n_p values. At large n_p values, these optimizations are negligible, and the small gain comes from minor inner-loop optimizations: fusing the mask with fma into one instruction and updating the index vector by addition instead of scalar broadcasting. Switching from `sqrt+div` to `rsqrt` also gave us a significant performance increase, as did simultaneously switching from AVX-256 to AVX-512 and adding ILP. Our very best version's performance stands at around 21.023 flops/cycle, approximately 71.67% of the performance bound as derived by the instruction mix and 43.8% of the hardware performance bound. This corresponds to a speedup of 18.26 from the baseline. As required, we also report the performance of our fastest non-vectorized code to be 1.19 flops/cycle, just about 3.5% better than the baseline. This (scalar) peak performance was achieved right after inlining the program (switching the data layout from AoS to SoA actually worsened performance, but we needed that step as a prerequisite for vectorization).

5. CONCLUSIONS

In this project, we implemented and optimized the social force model for simulating pedestrian dynamics. We leveraged inlining, vectorization, strength reduction, and ILP to improve our program's performance. After evaluating our optimizations on an x86 machine, we recorded a maximum speedup of 18 from the baseline, corresponding to 43.8% of the hardware performance bound and 71.67% of the instruction-mix performance bound. The largest performance gains came from vectorizing the inner loop, trading numerical accuracy for speed by using floats instead of doubles, and switching from AVX-256 to AVX-512. The optimized code is now bound by the remaining instruction mix, consisting of mostly FMAs and MULTs.

6. REFERENCES

[1] Dirk Helbing and Peter Molnar, "Social force model for pedestrian dynamics," *Physical review E*, vol. 51, no. 5, pp. 4282, 1995.

[2] Jiajun Jiang, Kehong Liu, Jiaqing Xie, and Peiyuan Xie, "Sfm-opt: Code-level optimization on social force model implementation," https://github.com/jiaqingxie/SFM-Opt/blob/main/38_report.pdf, 2023, Accessed: 2026-06-15.

[3] Markus Püschel, "Lecture: Compiler limitations," https://acl.inf.ethz.ch/teaching/fastcode/2026/slides/05-compiler-limitations.pdf, 2026.

[4] IEEE, "IEEE standard for floating-point arithmetic," IEEE Std 754-2019 (Revision of IEEE 754-2008), July 2019.

[5] Agner Fog, "Microarchitecture of intel, amd and via cpus – an optimization guide for assembly programmers and compiler makers," https://agner.org/optimize/microarchitecture.pdf, May 2026, Accessed: 2026-06-15.

7. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

Linus. First focused on non-SIMD optimization, LUT for exp, cutoff. Then ported the codebase to AVX-256. Switched from double to floats. Implementation of: exp() Approximation, RSQRT, FTZ and other minor optimizations regarding minimizing the instruction dependencies.

Quy Anh. Performed inlining to get rid of functions and structs. Ported the codebase to AVX-512 and wrote all AVX-512 variants except for the last two with outer loop unrolling and minor inner-loop optimizations. Implemented ILP for the inner loop. Implemented the switch from doubles to floats for the baseline and AVX-512 variants. Performed other minor optimizations not included in this report due to them not necessarily contributing to the best version, i.e. performing scalar replacements, implementing scalar fast inverse square root, and adding ILP to scalar float versions exploiting distance symmetry.

Ahmed. Minimized the number of flops performed in the first draft of the baseline implementation. Implemented neon simd code to see the resulting speedup from our optimization when run on ARM. Applied small optimizations to avx512 code (Incremental Index Vectors and Native AVX-512 Masked FMA) and implemented outer loop unrolling.

Smail. First focused on implementing different LUTs in non vectorized implementation and worked on ILP for AVX-256 as well as finetuning the ILP. Worked on exploiting symmetric relations (using LUTs) to reduce the flop count as well as approximating exponents using different approaches together with Linus.

8. CODE

Most performance-critical part of our fastest scalar code.

```
for (int j = 0; j < n_p; j++) {
  if (i == j)
    continue;

  double r_ab_x = p.curr_pos_x - pedestrians[j].curr_pos_x;
  double r_ab_y = p.curr_pos_y - pedestrians[j].curr_pos_y;

  double r_ab_norm = sqrt(r_ab_x * r_ab_x + r_ab_y * r_ab_y);

  double r_ab2_x = r_ab_x - pedestrians[j].s_b_e_x;
  double r_ab2_y = r_ab_y - pedestrians[j].s_b_e_y;

  double r_ab2_norm = sqrt(r_ab2_x * r_ab2_x + r_ab2_y * r_ab2_y);
  double r_ab_norm_sum = r_ab_norm + r_ab2_norm;
  double two_b_sq = r_ab_norm_sum * r_ab_norm_sum - pedestrians[j].s_b_sq;
  double two_b = sqrt(two_b_sq);
  double scalar_mult_common = (r_ab_norm_sum / (two_b * r_ab_norm * r_ab2_norm)) *
        (NEG_V0_AB_div_TWO_SIGMA * exp(two_b * NEG_TWO_SIGMA_INV));
  double scalar_mult_r_ab = scalar_mult_common * r_ab2_norm;
  double scalar_mult_r_ab2 = scalar_mult_common * r_ab_norm;

  double f_alpha_beta_neg_x = r_ab_x * scalar_mult_r_ab + r_ab2_x * scalar_mult_r_ab2;
  double f_alpha_beta_neg_y = r_ab_y * scalar_mult_r_ab + r_ab2_y * scalar_mult_r_ab2;

  double w_val = dt_neg;
  double a = (p.e_x * f_alpha_beta_neg_x + p.e_y * f_alpha_beta_neg_y) * cos_phi_inv;
  double b_sq = f_alpha_beta_neg_x * f_alpha_beta_neg_x + f_alpha_beta_neg_y * f_alpha_beta_neg_y;

  if (a >= 0 && (a * a > b_sq)) {
    w_val = c_neg_times_dt;
  }

  f_alpha_dt_x = f_alpha_dt_x + f_alpha_beta_neg_x * w_val;
  f_alpha_dt_y = f_alpha_dt_y + f_alpha_beta_neg_y * w_val;
}
```

Most performance-critical part of our fastest SIMD code.

```
for (int j = 0; j < n_p_even; j += 2) {
  const __m512 pjx_0 = _mm512_set1_ps(pos_x[j]);
  const __m512 pjy_0 = _mm512_set1_ps(pos_y[j]);
  const __m512 sbe_jx_0 = _mm512_set1_ps(sb_e_x[j]);
  const __m512 sbe_jy_0 = _mm512_set1_ps(sb_e_y[j]);
  const __m512 sb_jsq_0 = _mm512_set1_ps(sb_sq[j]);
  const __mmask16 active_0 = _mm512_cmp_ps_mask(i_idx, j_idx_0, _CMP_NEQ_OQ);
  const __m512 pjx_1 = _mm512_set1_ps(pos_x[j + 1]);
  const __m512 pjy_1 = _mm512_set1_ps(pos_y[j + 1]);
  const __m512 sbe_jx_1 = _mm512_set1_ps(sb_e_x[j + 1]);
  const __m512 sbe_jy_1 = _mm512_set1_ps(sb_e_y[j + 1]);
  const __m512 sb_jsq_1 = _mm512_set1_ps(sb_sq[j + 1]);
  const __mmask16 active_1 = _mm512_cmp_ps_mask(i_idx, j_idx_1, _CMP_NEQ_OQ);
  __m512 r_ab_x_0 = _mm512_sub_ps(pix, pjx_0);
  __m512 r_ab_x_1 = _mm512_sub_ps(pix, pjx_1);
  __m512 r_ab_y_0 = _mm512_sub_ps(piy, pjy_0);
  __m512 r_ab_y_1 = _mm512_sub_ps(piy, pjy_1);
  __m512 r_ab_n2_0 = _mm512_fmadd_ps(r_ab_x_0, r_ab_x_0, _mm512_mul_ps(r_ab_y_0, r_ab_y_0));
  __m512 r_ab_n2_1 = _mm512_fmadd_ps(r_ab_x_1, r_ab_x_1, _mm512_mul_ps(r_ab_y_1, r_ab_y_1));
  r_ab_n2_0 = _mm512_max_ps(r_ab_n2_0, v_eps_sq);
```

```
  r_ab_n2_1 = _mm512_max_ps(r_ab_n2_1, v_eps_sq);
  __m512 inv_r_ab_n_0 = _mm512_rsqrt14_ps(r_ab_n2_0);
  __m512 inv_r_ab_n_1 = _mm512_rsqrt14_ps(r_ab_n2_1);
  __m512 r_ab_n_0 = _mm512_mul_ps(r_ab_n2_0, inv_r_ab_n_0);
  __m512 r_ab_n_1 = _mm512_mul_ps(r_ab_n2_1, inv_r_ab_n_1);
  __m512 r_ab2_x_0 = _mm512_sub_ps(r_ab_x_0, sbe_jx_0);
  __m512 r_ab2_x_1 = _mm512_sub_ps(r_ab_x_1, sbe_jx_1);
  __m512 r_ab2_y_0 = _mm512_sub_ps(r_ab_y_0, sbe_jy_0);
  __m512 r_ab2_y_1 = _mm512_sub_ps(r_ab_y_1, sbe_jy_1);
  __m512 r_ab2_n2_0 = _mm512_fmadd_ps(r_ab2_x_0, r_ab2_x_0, _mm512_mul_ps(r_ab2_y_0, r_ab2_y_0));
  __m512 r_ab2_n2_1 = _mm512_fmadd_ps(r_ab2_x_1, r_ab2_x_1, _mm512_mul_ps(r_ab2_y_1, r_ab2_y_1));
  r_ab2_n2_0 = _mm512_max_ps(r_ab2_n2_0, v_eps_sq);
  r_ab2_n2_1 = _mm512_max_ps(r_ab2_n2_1, v_eps_sq);
  __m512 inv_r_ab2_n_0 = _mm512_rsqrt14_ps(r_ab2_n2_0);
  __m512 inv_r_ab2_n_1 = _mm512_rsqrt14_ps(r_ab2_n2_1);
  __m512 r_ab2_n_0 = _mm512_mul_ps(r_ab2_n2_0, inv_r_ab2_n_0);
  __m512 r_ab2_n_1 = _mm512_mul_ps(r_ab2_n2_1, inv_r_ab2_n_1);
  __m512 r_sum_0 = _mm512_add_ps(r_ab_n_0, r_ab2_n_0);
  __m512 r_sum_1 = _mm512_add_ps(r_ab_n_1, r_ab2_n_1);
  __m512 two_b_sq_0 = _mm512_fmsub_ps(r_sum_0, r_sum_0, sb_jsq_0);
  __m512 two_b_sq_1 = _mm512_fmsub_ps(r_sum_1, r_sum_1, sb_jsq_1);
  two_b_sq_0 = _mm512_max_ps(two_b_sq_0, v_eps_sq);
  two_b_sq_1 = _mm512_max_ps(two_b_sq_1, v_eps_sq);
  __m512 inv_two_b_0 = _mm512_rsqrt14_ps(two_b_sq_0);
  __m512 inv_two_b_1 = _mm512_rsqrt14_ps(two_b_sq_1);
  __m512 two_b_0 = _mm512_mul_ps(two_b_sq_0, inv_two_b_0);
  __m512 two_b_1 = _mm512_mul_ps(two_b_sq_1, inv_two_b_1);
  __m512 e_arg_0 = _mm512_max_ps(_mm512_mul_ps(two_b_0, v_neg_2s_inv), v_exp_min);
  __m512 e_arg_1 = _mm512_max_ps(_mm512_mul_ps(two_b_1, v_neg_2s_inv), v_exp_min);
  __m512 e_val_0 = _avx512_exp_ps(e_arg_0);
  __m512 e_val_1 = _avx512_exp_ps(e_arg_1);
  __m512 numer_inv_two_b_0 = _mm512_mul_ps(_mm512_mul_ps(r_sum_0, v_neg_v0_div_2s),
                                           _mm512_mul_ps(e_val_0, inv_two_b_0));
  __m512 numer_inv_two_b_1 = _mm512_mul_ps(_mm512_mul_ps(r_sum_1, v_neg_v0_div_2s),
                                           _mm512_mul_ps(e_val_1, inv_two_b_1));
  __m512 sm_r_ab_0 = _mm512_mul_ps(numer_inv_two_b_0, inv_r_ab_n_0);
  __m512 sm_r_ab_1 = _mm512_mul_ps(numer_inv_two_b_1, inv_r_ab_n_1);
  __m512 sm_r_ab2_0 = _mm512_mul_ps(numer_inv_two_b_0, inv_r_ab2_n_0);
  __m512 sm_r_ab2_1 = _mm512_mul_ps(numer_inv_two_b_1, inv_r_ab2_n_1);
  __m512 f_x_0 = _mm512_fmadd_ps(r_ab2_x_0, sm_r_ab2_0, _mm512_mul_ps(r_ab_x_0, sm_r_ab_0));
  __m512 f_x_1 = _mm512_fmadd_ps(r_ab2_x_1, sm_r_ab2_1, _mm512_mul_ps(r_ab_x_1, sm_r_ab_1));
  __m512 f_y_0 = _mm512_fmadd_ps(r_ab2_y_0, sm_r_ab2_0, _mm512_mul_ps(r_ab_y_0, sm_r_ab_0));
  __m512 f_y_1 = _mm512_fmadd_ps(r_ab2_y_1, sm_r_ab2_1, _mm512_mul_ps(r_ab_y_1, sm_r_ab_1));
  __m512 f_norm_sq_0 = _mm512_fmadd_ps(f_x_0, f_x_0, _mm512_mul_ps(f_y_0, f_y_0));
  __m512 f_norm_sq_1 = _mm512_fmadd_ps(f_x_1, f_x_1, _mm512_mul_ps(f_y_1, f_y_1));
  __m512 rhs_0 = _mm512_mul_ps(f_norm_sq_0, v_cos_phi_sq);
  __m512 rhs_1 = _mm512_mul_ps(f_norm_sq_1, v_cos_phi_sq);
  __m512 e_dot_f_0 = _mm512_fmadd_ps(eix, f_x_0, _mm512_mul_ps(eiy, f_y_0));
  __m512 e_dot_f_1 = _mm512_fmadd_ps(eix, f_x_1, _mm512_mul_ps(eiy, f_y_1));
  __m512 e_dot_f_sq_0 = _mm512_mul_ps(e_dot_f_0, e_dot_f_0);
  __m512 e_dot_f_sq_1 = _mm512_mul_ps(e_dot_f_1, e_dot_f_1);
  __mmask16 cond_0 = _mm512_cmp_ps_mask(e_dot_f_0, v_zero, _CMP_LT_OQ) &
                     _mm512_cmp_ps_mask(e_dot_f_sq_0, rhs_0, _CMP_GT_OQ);
  __mmask16 cond_1 = _mm512_cmp_ps_mask(e_dot_f_1, v_zero, _CMP_LT_OQ) &
                     _mm512_cmp_ps_mask(e_dot_f_sq_1, rhs_1, _CMP_GT_OQ);
  __m512 w_val_0 = _mm512_mask_blend_ps(cond_0, v_dt_neg, v_c_neg_dt);
  __m512 w_val_1 = _mm512_mask_blend_ps(cond_1, v_dt_neg, v_c_neg_dt);
  acc_x = _mm512_mask3_fmadd_ps(f_x_0, w_val_0, acc_x, active_0);
  acc_y = _mm512_mask3_fmadd_ps(f_y_0, w_val_0, acc_y, active_0);
  acc_x2 = _mm512_mask3_fmadd_ps(f_x_1, w_val_1, acc_x2, active_1);
  acc_y2 = _mm512_mask3_fmadd_ps(f_y_1, w_val_1, acc_y2, active_1);
  j_idx_0 = _mm512_add_ps(j_idx_0, v_two);
  j_idx_1 = _mm512_add_ps(j_idx_1, v_two);
}
```