HIGH-PERFORMANCE SOCIAL FORCE MODEL SIMULATION

Eiman Alnuaimi *Mingyuan Chi* *Moritz Kuntze* *Jonas Lauer*

Department of Computer Science
ETH Zürich, Switzerland

ABSTRACT

Simulation of the social force model is widely used for pedestrian dynamics prediction. However, it has high computation intensity. In order to accelerate this algorithm, we implemented a high-performance social force model simulation that could reach as high as 4 flops/cycle.

1. INTRODUCTION

Motivation. In urban design and planning, achieving an efficient transportation network is a paramount objective. Architects must consider predictive models to optimize their designs for identifying congestion points or minimizing travel time based on the behavior of the network's participants. While predictions can be done quite precisely for centrally controlled networks, like train networks [1], making assertions about pedestrian flow in public spaces is challenging due to their unpredictable nature, especially in crowded situations.

Pedestrian dynamics have been analogously linked to the behavior of fluids, thus leading to their examination through differential equations originally developed for fluid systems [2, 3]. Another modeling approach entails employing an equation of motion that encapsulates the human behavioral responses to their surroundings as *social forces* [4]. The social force model can predict pedestrian movement by incorporating factors such as the desired speed towards an intended destination, repulsive and attractive influences exerted by other pedestrians or objects, and a fluctuation term accounting for the inherent variability in their behavior.

Despite the algorithm simulation's success in replicating observed behavioral patterns within densely populated environments, it suffers from a non-negligible quadratic algorithmic complexity that grows with the number of pedestrians. To enable the application of the social force model as a predictive pedestrian dynamics model across real-world scenarios, we develop a performance-optimized implementation that exhibits efficient execution on contemporary hardware platforms.

Contribution. In this paper, we present an optimized social force model conceived for Intel's Skylake microachitecture. We employ various optimization techniques to improve performance and cache locality. These techniques encompass the exploration of three memory layouts; Array of Structures (AoS), Structure of arrays (SoA) and Array of Structures of Arrays (AoSoA); utilization of single-instruction/multiple-data (SIMD) vectorization intrinsics, and semi-manual register allocation. Finally, we contribute i) a quantitative comparison between Aos, SoA and AoSoA memory layouts ii) an assessment of performance on two different compilers, GCC 11.3.0 and Clang 14 with different compiler flags iii) something about the exponential function(comparing the default gnulib expo with the fast exponentiation function) iv) something about register pressure and manual allocation's impact on performance

Related work. The social force model has gained significant attention in response to population growth, primary aimed to understand pedestrian behavior. PedSim is a library written in C that provides various functions [5]. Moreover, [6] is a C++ simulation that represents an enhanced version of [4]'s algorithm. This simulation incorporates calibrated model parameters obtained from real-world experiments. Despite the utilization of Vector data structures, neither implementations have undergone further optimization.

There exists an OpenCL (GPU) implementation of PedSim that is capable of simulating 20,000 pedestrians with 8 model steps per second [7]. This implementation offers exceptional performance, however, it's incomparable to what we present in this paper due to our limited access to CPU only. [8] is a Java-based simulation of the social force model that employs parallelization through multi-threading to distribute the quadratic workload across many threads.

2. THE SOCIAL FORCE MODEL

In a condensed form[1], the social force model [4] defines three forces that determine a pedestrian α's movement:

1. **Destination attraction**. A pedestrian always has a target position $\overrightarrow{r}_\alpha^k$ it is heading for. They accelerate or decelerate their movement into that direction, approaching a desired speed v_α^0 within τ seconds. Thus,

[1] We are going to ignore the force that attracts pedestrians to places of interest here.

if α's current position is $\overrightarrow{r}_\alpha$ and their current velocity is $\overrightarrow{v}_\alpha$, one can compute this attractive force as

$$\overrightarrow{F}^0_\alpha = \frac{1}{\tau} \cdot \left(v^0_\alpha \cdot \frac{\overrightarrow{r}^k_\alpha - \overrightarrow{r}_\alpha}{\|\overrightarrow{r}^k_\alpha - \overrightarrow{r}_\alpha\|} - \overrightarrow{v}_\alpha \right),$$

$\|\cdot\|$ denoting the Euclidean norm.

2. **Pedestrian repulsion**. The model assumes that pedestrians don't tend to be too close to other pedestrians in order to avoid a potential collision. To avoid colliding with another pedestrian β, pedestrian α takes into account β's direction of movement and step width[2]

$$\overrightarrow{h}_\beta := \|\overrightarrow{v}_\beta\| \cdot \text{footstep} \cdot \frac{\overrightarrow{r}^k_\beta - \overrightarrow{r}_\beta}{\|\overrightarrow{r}^k_\beta - \overrightarrow{r}_\beta\|}$$

and computes its repulsive potential from β as

$$\overrightarrow{F}_{\alpha,\beta} = \overrightarrow{f}_{\alpha,\beta} \cdot \begin{cases} 1 & \text{if } \left(\frac{\overrightarrow{r}^k_\alpha - \overrightarrow{r}_\alpha}{\|\overrightarrow{r}^k_\alpha - \overrightarrow{r}_\alpha\|} \cdot \overrightarrow{f}_{\alpha,\beta}\right) \\ & \quad \geq \|\overrightarrow{f}_{\alpha,\beta}\| \cdot \cos(\varphi) \\ c & \text{otherwise} \end{cases}$$

with

$$b = \frac{1}{2}\sqrt{\left(\|\overrightarrow{r}_{\alpha,\beta}\| + \|\overrightarrow{r}_{\alpha,\beta} - \overrightarrow{h}_\beta\|\right)^2 - \|\overrightarrow{h}_\beta\|^2}$$

$$\begin{aligned} \overrightarrow{f}_{\alpha,\beta} &= -\nabla_{\overrightarrow{r}_{\alpha,\beta}} V_0 \cdot e^{-b/\sigma} \\ &= \frac{V_0}{4\sigma b} e^{-\frac{b}{\sigma}} \cdot \left(\|\overrightarrow{r}_{\alpha\beta}\| + \|\overrightarrow{r}_{\alpha\beta} - \overrightarrow{h}_\beta\|\right) \\ &\quad \cdot \left(\frac{\overrightarrow{r}_{\alpha\beta}}{\|\overrightarrow{r}_{\alpha\beta}\|} + \frac{\overrightarrow{r}_{\alpha\beta} - \overrightarrow{h}_\beta}{\|\overrightarrow{r}_{\alpha\beta} - \overrightarrow{h}_\beta\|}\right) \end{aligned}$$

where $\overrightarrow{r}_{\alpha,\beta}$ is the relative position between pedestrian α and pedestrian β, and$(a \cdot b)$ denotes the dot product of vector a and b (for a more detailed description and explanation, please refer to [4]). Note that the case distinction above ensures that if pedestrian β is outside of pedestrian α's angle of sight φ, α's repulsion from β is reduced by factor c, because α has limited perception on β in this case.

3. **Obstacle repulsion**. The pedestrians also don't want to collide with walls or other obstacles. Therefore, the model defines an obstacle repulsion force, that will push them away from the obstacles as they get closer, namely

$$\begin{aligned} \overrightarrow{F}_{\alpha B}(t) &= -\nabla_{\overrightarrow{r}_{\alpha B}} U_0 \cdot e^{-\|\overrightarrow{r}_{\alpha B}\|/R} \\ &= \frac{U_0}{R} \cdot e^{-\|\overrightarrow{r}_{\alpha B}\|/R} \frac{\overrightarrow{r}_{\alpha B}}{\|\overrightarrow{r}_{\alpha B}\|}, \end{aligned}$$

where U_0 and R are the maximum force and decay factor which are constants in this formula. And $\|\overrightarrow{r}_{\alpha B}\|$ denotes the euclidean distance between the pedestrian α and the closest point of obstacle B.

These forces are computed for each pedestrian and summed up, yielding the acceleration $\overrightarrow{F}_\alpha$ of the respective pedestrian at that specific time. Taking into account a maximal velocity $v^{\max}_\alpha$ of the pedestrian that they won't exceed, one obtains the following differential equations for velocity and position:

1. $$\frac{d\overrightarrow{v}_\alpha}{dt} = \overrightarrow{F}_\alpha \cdot \begin{cases} 1 & \text{if } \|\overrightarrow{F}_\alpha\| \leq v^{\max}_\alpha \\ \frac{v^{\max}_\alpha}{\|\overrightarrow{F}_\alpha\|} & \text{otherwise} \end{cases}$$
2. $$\frac{d\overrightarrow{r}_\alpha}{dt} = \overrightarrow{v}_\alpha.$$

Simulation. For our implementation, we approximate the differential equations with the semi-implicit Euler method. Namely, we compute

$$\overrightarrow{v}^{(i+1)}_\alpha \leftarrow \overrightarrow{v}^{(i)}_\alpha + \Delta t \cdot \overrightarrow{F}^{(i)}_\alpha \cdot \begin{cases} 1 & \text{if } \|\overrightarrow{F}^{(i)}_\alpha\| \leq v^{\max}_\alpha \\ \frac{v^{\max}_\alpha}{\|\overrightarrow{F}^{(i)}_\alpha\|} & \text{otherwise} \end{cases}$$

$$\overrightarrow{r}^{(i+1)}_\alpha \leftarrow \overrightarrow{r}^{(i)}_\alpha + \Delta t \cdot \overrightarrow{v}^{(i+1)}_\alpha.$$

Furthermore, like in [4], a pedestrian has a path containing their starting position, arbitrarily many intermediary goals and one final goal. Upon reaching the final goal, we decided to respawn the pedestrian to their initial state, heading for their first goal again. This ensures that the number of pedestrians stays constant over the entire simulation, which makes the runtime independent of the pedestrian's paths and thus greatly simplifies our analysis.

Lastly, we restrict ourselves to obstacles of the form of straight lines $l = (\overrightarrow{a}, \overrightarrow{b})$ from point $\overrightarrow{a}$ to point $\overrightarrow{b}$, as this makes it easy to compute its closest point $\overrightarrow{c}$ to a pedestrian at position $\overrightarrow{p}$ as

1. $$\overrightarrow{ba} = \overrightarrow{b} - \overrightarrow{a}, \quad \overrightarrow{pa} = \overrightarrow{p} - \overrightarrow{a}, \quad \overrightarrow{pb} = \overrightarrow{p} - \overrightarrow{b}$$
2. $$j = \frac{\overrightarrow{ba}_x \cdot \overrightarrow{pa}_x + \overrightarrow{ba}_y \cdot \overrightarrow{pa}_y}{\|\overrightarrow{ba}\|^2}$$
3. $$\overrightarrow{d} = \begin{cases} \overrightarrow{a} & \text{if } j \leq 0 \\ \overrightarrow{b} & \text{if } j \geq 1 \\ \overrightarrow{p} - \left(\overrightarrow{a} + j \cdot \overrightarrow{b}\right) & \text{otherwise.} \end{cases}$$

A formal derivation of this calculation is given in appendix **??**.

The overall simulation loop then roughly looks like it is shown in Algorithm 1.

Cost Analysis and First Performance Bound.

First define you cost measure (what you count) and then compute or determine on other ways the cost as explained

[2]Note that the paper does not explicitly say how to compute this quantity. We decided to introduce the footstep-constant and not to reuse the simulation time step size Δt for this purpose, because we think that a pedestrian's step width should not depend on how we discretize time in our simulation.

```
simulated_time ← 0
while simulated_time < duration do
    Respawn finished pedestrians
    for each pedestrian α do
        Compute destination attraction
        for each pedestrian β ≠ α do
            Compute repulsion of α from β
        end
        for each obstacle B do
            Compute repulsion of α from B
        end
        Compute semi-implicit Euler step for α
    end
    simulated_time += step width
end
```

Algorithm 1: Test

in class. In the end you will likely consolidate it into one number (e.g., adds/mults/comparisons) but be aware of major imbalances as they affect the peak performance..

Also state what is known about the complexity (asymptotic usually) about your problem (including citations).

3. INVESTIGATING MEMORY LAYOUTS AND COMPILER VECTORIZATION

The AoS, SoA and AoSoA memory layouts are the three most basic layouts for a set of compound data and represent three different ways of interleaving data fields. Assuming each data element (e.g. each pedestrian) can be represented as a structure composed of primitive types (e.g. `double`), the AoS layout stores the data as an *Array of Structs*, i.e. it stores a complete data record contiguously in the memory preceding the next data record. The SoA layout inverts this interleaving pattern, storing a *Structure of Arrays* instead: It stores contiguous arrays for each field of the data record. The benefit of the latter approach is that it allows contiguous memory loads of data with identical semantic meaning, allowing for easier application of SIMD operations: Whereas an aligned load of four doubles in the AoS layout would have yielded a mix of structure fields, an aligned load of four doubles in the SoA layout yields a vector of four values of the same field, but of different pedestrians.

The AoSoA (Array of Structures of Arrays, or "tiled SoA") layout combines and generalizes the AoS and SoA approaches: It combines a typically fixed number (the block size, K) of data records into blocks stored in an SoA fashion and then stores the blocks themselves in a simple array (in an AoS fashion). This has two main advantages: i) It reduces to the AoS layout when $K = 1$ and to the SoA implementation when $K = N$, where N is the number of data records. This allows for a generalzied implementation, enabling the developer to choose the final layout based on performance benchmarks for varying K; and ii) for K being a multiple of the SIMD vector size, it allows for the same aligned, equal-meaning vector reads as the SoA layout while maintaining some degree of locality of data belonging to the same pedestrian (or group of pedestrians), which may yield better cache locality and better performance depending on the dataset.

Experimental Setup. In order to investigate the capabilities of automatic compiler vectorization, we implemented the social force model with all three memory layouts described above and benchmarked the three implementations using both GCC 11.3.0 and Clang 14, each with five optimization profiles: All optmizations (`-O3`), including FMAs, vectorization and link-time optimizations (`FullOpt`), all optimizations, except LTO (`FullOptNoLTO`), all optmiziations, except LTO and FMAs (`VecNoFMA`), all optmizations except LTO and vectorization (`FMANoVec`) and all optmizations except LTO, FMAs and vectorization (`NoVecNoFMA`). All benchmarks were performed on an Intel Core i7-6700, clocked at 3.4 GHz with Turbo Boost disabled, running Ubuntu Server.

Results.

4. MANUAL VECTORIZATION

As compiler vectorization did not produce a significant improvement in performance, we implemented a hand-vectorized version of the AoSoA implementation using Intel Intrinsics. While this implementation is straightforward for most parts of the social force algorithm, some implementation details are nontrivial and highlighted in the following sections.
AoSoA base unit and Alignment. In order to simplify the Intrinsics-based implementation, `__m256d` was chosen as the base unit of the AoSoA fields, instead of `double` and all structs as well as memory allocations were aligned to 32-byte boundaries to ensure that all struct fields could always be loaded into registers using aligned loads. **Shuffling.**

5. IMPROVED PERFORMANCE ESTIMATION

In the remaining parts of the paper [[report?]], we address the question where this discrepancy between our achieved performance and the maximum performance bound of section **??** comes from and how we might be able to close this gap. For this section, let us first consider the performance bound.

As described in section **??**, the performance bound is solely based on the instruction mix. The reason for that are the following two assumptions:

a) We are compute-bound, since we have $\Theta(n^2 + nm)$ operations to be performed on $\mathcal{O}(n + m)$ data, n be-

ing the number of pedestrians, m the number of obstacles.

b) For our bottleneck, pedestrian repulsion, have n independent loops whose iterations are commutative and associative, because they just accumulate (i.e. sum up) the repulsion from every other pedestrian. Thus, there should be always enough instructions ready to be executed to fill the pipeline.

However, it turns out that we are actually not that much compute-bound if we take cache sizes into consideration. The working set of our pedestrian repulsion nested loop consists of 48 B per pedestrian we iterate over in the inner loop, plus 192 B for the outer loop pedestrian (due to the shuffling) plus constants. Once this size exceeds the size of a cache, we need to load some of the values in each iteration of the inner loop to that cache. For the L3 cache of size 2 MiB, this happens if $n > 43,686$, for L2 (256 KiB) if $n > 5457$ and for L1 (32 KiB) if $n > 678$ pedestrians. Because the bandwidths for loads and stores of those caches are not high when considering vector operands[3], we should consider loads and stores in our analysis.

Note that because of the shuffling, the outer loop working set consists of (24 + constants) vectors, so there is no way we can keep all of them in registers. Indeed, an inspection of the generated assembly [**?**] shows that about 40% of the instructions for pedestrian repulsion have a memory operand, a majority of these being part of the outer loop working set, i.e. values that don't change in the inner loop and thus, in principle, wouldn't need to be loaded in the inner loop. Therefore, to improve our performance bound, the first question becomes: Which values need to be loaded in each inner loop iteration, in order to minimize loads into registers?

To that end, we can take a look at the pedestrian repulsion dependency graph, to see how many registers are required by the computation alone (for now, let's ignore the fact that we are actually computing four potentially interleaving pedestrian repulsions in each inner loop iteration because of the shuffling). It turns out that it requires at least 9 registers, as it is argued in more detail in appendix **??**. This leaves us with 7 registers in which we can keep 7 of the 8 force accumulators, as these should be the most important variables to keep, since they are the only ones that are read and written (all other variables are just read). So this strategy should be optimal, as it minimizes stores and loads. In return, we must consider all other vectors of the working set to be loaded for each pedestrian repulsion computation (meaning four times in the innermost loop).

Naturally, the question arises why interleaving the four pedestrian repulsions cannot save us some of the loads, as it seems redundant to load the same vector that is needed four times in the same inner loop iteration four times within that iteration. The reason is that when interleaving the pedestrian repulsions, the increase in the number of loads, incurred by the additional registers required to store the intermediary values of different pedestrian repulsions, always dominates the decrease in the number of loads that are saved because some vector load is used by multiple pedestrian repulsions (see appendix **??** for more details). What follows from this conclusion is that unrolling any of the loops will not increase the achieved performance. Moreover, when we try in the following to improve our performance bound, we can always assume the inner loop's instructions to simply consist of a sequence of $n - 1$ pedestrian repulsions.

Scheduling μops. Now that we have decided which values need to be loaded, let's take a closer look at the second assumption, namely that the n independent inner loops will always give us enough instructions that are ready to be executed to fully occupy ports 0 and 1.

First of all, it is not correct to assume that the processor can only execute one pedestrian repulsion at a time only because they, as we just concluded, do not interleave and use the same registers. Skylake CPUs are capable of performing out-of-order execution and register renaming. Therefore, even if two instructions write to the same *architectural* register, they can be executed in any order, the results being temporarily stored in some of the 168 *physical* registers [**?**] before the most recent of them is (or both are) written back to the architectural register.

However, the amount of instruction reordering a Skylake processor is capable of is limited by size of the Reorder Buffer (ROB), which has a capacity of 224 μops [**?**]. The ROB is a FIFO (first in, first out) queue that contains the sequence of instructions the processor is currently executing. Once the first μop in the FIFO (the head) has finished, it is removed and the next μop in program memory is attached to the end of the queue[4]. Any μop in the ROB can be executed once all of its operands have been computed (i.e. have been written to the physical register file).

On the other hand, this means that if the distance between two instructions in program memory is more than 224 μops, they cannot be reordered. Hence, because we assume the pedestrian repulsion computations not to interleave (in program memory) and each pedestrian repulsion contains

[3] 2 vectors load + 1 vector store per cycle for Register-L1 traffic, 2 vectors per cycle for L1-L2 traffic, 1 vector per cycle for L2-L3 traffic, 1/2 vector per cycle for L3-RAM traffic [**?**]. Note also that the latency of loads and stores is 5 cycles for 256 bit vectors [**?**], so always loading and storing data from/to L1 cache won't fill the pipeline with 4-cycle-latency ALU flops.

[4] Note that these management operations on the ROB data structure can be implemented quite efficiently and won't cause any delays. Especially, many μops can be loaded into the ROB within one CPU cycle.

[[TODO]] μops (see appendix **??**), it follows that no more than [[3]] pedestrian repulsions can be executed at the same time.

To learn how this observation influences the achievable performance of our program, we tried to derive an optimal schedule for the inner pedestrian repulsion loop that includes the loads we have determined before and respects the ROB size. However, deriving this schedule by hand turned out to be somewhat infeasible and error prone.

Therefore, we created an automatic scheduler [**?**] that follows a few simple, reasonable rules when deciding on which μop to schedule next (described in more detail in appendix **??**). Instead of letting the automatic scheduler derive the complete schedule for all $n-1$ pedestrian repulsions of the inner loop, we used the observation that, because the scheduler is deterministic and stateless, the generated schedule will repeat, once enough pedestrian repulsions are scheduled. Thus, we equipped the scheduler with a mechanism to detect this repetition. From this periodic schedule we were then able to calculate in which cycle the number of finished μops is equal to the number of μops of the complete inner loop and we used this number as a runtime estimate to replace the pedestrian repulsion part in our previous instruction mix bound.

6. RESULTS

Performance Comparison. For different K sizes in AoSoA memory layout with SIMD, we compare their performance in the roofline plot with two versions compiled from Clang and GCC with all flags on. As shown in Figure 1, the

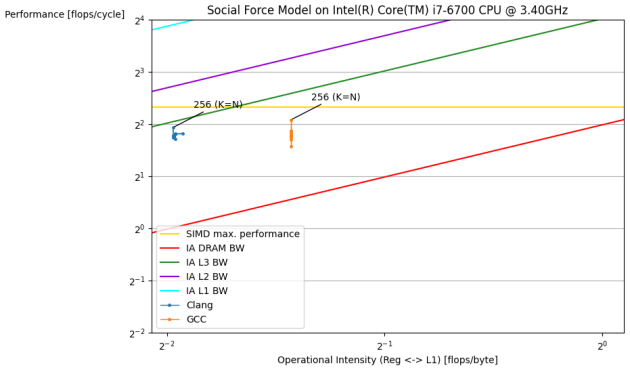


Fig. 1: Roofline plot with varying K size

Clang compiler tend to optimize the memory access better than the GCC compiler. One could also find that when the K equals to the number of pedestrians N, it achieves the highest performance. It means that the SoA memory layout could achieve the best performance in this problem.

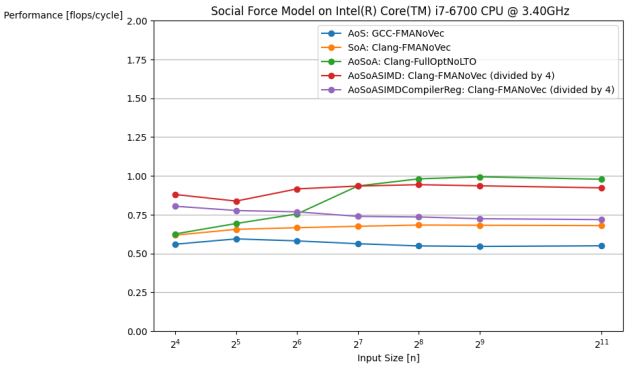


Fig. 2: Performance comparison for different implementation

In Figure 2, we picked the best performance among all compiler-flags combinations for all kinds of memory layouts and compare them under different numbers of pedestrians. It's obvious that our AoSoA memory layout with a semi-human register assignment is always better than the compiler register assignment.

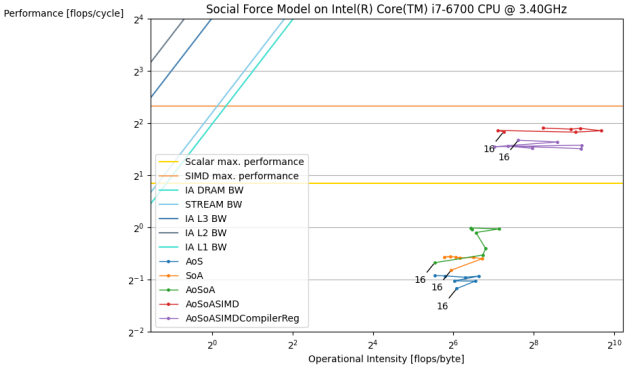


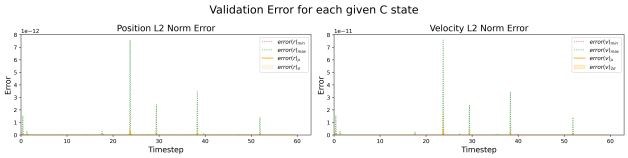
Fig. 3: Roofline plot with fixed K size

We also plots those implementations in the roofline plots. As shown in Figure 3, our SIMD implementation tend to reach the computation bound, but still a small gap among them. It may result from the register pressure and limitation of ROB, which is fully discussed in Section 5

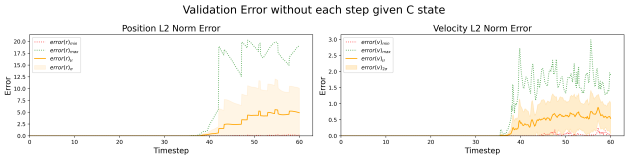
Accuracy Comparison.

In order to ensure the accuracy of our optimization, we develop a validation system in Python. It could calculate the difference in pedestrians' positions and velocities for single-step and for accumulation between the C implementation and the Python implementation.

For example, Figure 4 shows the errors for the most intuitive AoS memory layout. As can be seen, the error accumulates over time but is still acceptable. Since the social-force model is a kind of n-body problem that is chaotic. The growing error is inevitable. The small disturbance may be



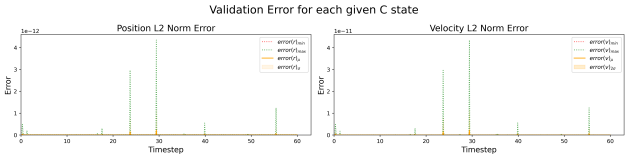
(a) AoS single step error



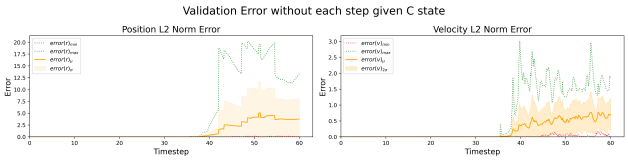
(b) AoS accumulate error

Fig. 4: position and velocity L2 single step 4a and accumulate 4b Error for AoS memory layout

caused by the difference in the implementation of cos and exp in C and Python.



(a) AoSoA single step error



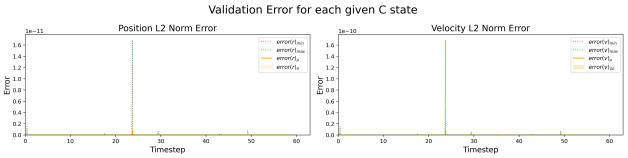
(b) AoSoA accumulate error

Fig. 5: position and velocity L2 single step 5a and accumulate 5b Error for AoSoA Scalar memory layout

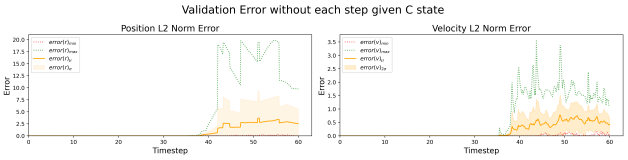
Different memory layouts don't introduce any computational errors. Figure 5 demonstrates the error of AoSoA scalar memory layout. It's similar compared to Figure 4, confirming the correctness of our implementation.

Switching from scalar to SIMD introduces almost no error. As can be seen from Figure 5 to Figure 6, the single step and accumulate error remains almost identical and sometimes even lower compared to the scalar version. It shows that our shuffle techniques in SIMD are equivalent to the scalar version.

Using fast exp increases the speed but brings small error at the same time. From Figure 6 to Figure 7, the difference is still within our acceptance range. Compare Figure 7 to the initial version of Figure 4. The error only varies at a very tiny scale. Therefore, we could assert that our optimized model is correct.

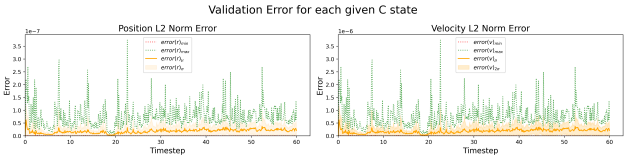


(a) AoSoASIMDCompilerReg single step error

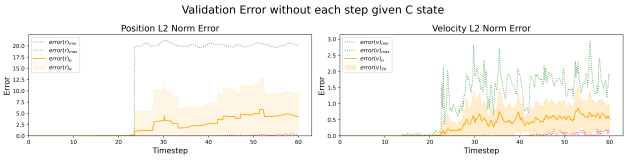


(b) AoSoASIMDCompilerReg accumulate error

Fig. 6: position and velocity L2 single step 6a and accumulate 6b Error for AoSoA SIMD memory layout with compiler register assignment



(a) AoSoASIMD single step error



(b) AoSoASIMD accumulate error

Fig. 7: position and velocity L2 single step 7a and accumulate 7b Error for AoSoA SIMD memory layout with semi-human register assignment

7. CONCLUSIONS

Here you need to briefly summarize what you did and why this is important. *Do not take the abstract* and put it in the past tense. Remember, now the reader has (hopefully) read the paper, so it is a very different situation from the abstract. Try to highlight important results and say the things you really want to get across (e.g., the results show that we are within 2x of the optimal performance ... Even though we only considered the DFT, our optimization techniques should be also applicable ....) You can also formulate next steps if you want. Be brief.

8. FURTHER COMMENTS

Here we provide some further tips.

Further general guidelines.

- For short papers, to save space, I use paragraph titles instead of subsections, as shown in the introduction.
- It is generally a good idea to break sections into such smaller units for readability and since it helps you to (visually) structure the story.
- The above section titles should be adapted to more precisely reflect what you do.
- Each section should be started with a very short summary of what the reader can expect in this section. Nothing more awkward as when the story starts and one does not know what the direction is or the goal.
- Do not use subsubsections.
- Make sure you define every acronym you use, no matter how convinced you are the reader knows it.
- Always spell-check before you submit.
- Be picky. When writing a paper you should always strive for high quality. Many people may read it and the quality makes a big difference. In this class, the quality contributes to the grade.
- Books helping you to write better: [9] and [10].

Graphics. For plots that are not images *never* generate (even as intermediate step) jpeg, gif, bmp, tif. Use eps, which means encapsulate postscript, or pdf. This way it is scalable since it is a vector graphic description of your graph. E.g., from Matlab, you can export to eps or pdf.

Fig. 8 is an example plot that I used in a lecture. Note that the fontsize in the plot should not be any smaller. On the other hand it is also a good rule that the font size in the plot is not larger than the one in the caption (otherwise it looks ugly).

Up to here you have 8 pages.

9. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

In this mandatory section (which is not included in the 8 pages limit) each team member should very briefly (telegram style is welcome) explain what she/he did for the project. I imagine this section to be between one column and one page (absolute maximum).

Include only

- What relates to optimizing your chosen algorithm / application. This means writing actual code for optimization or for analysis.
- What you did before the submission of the presentation.

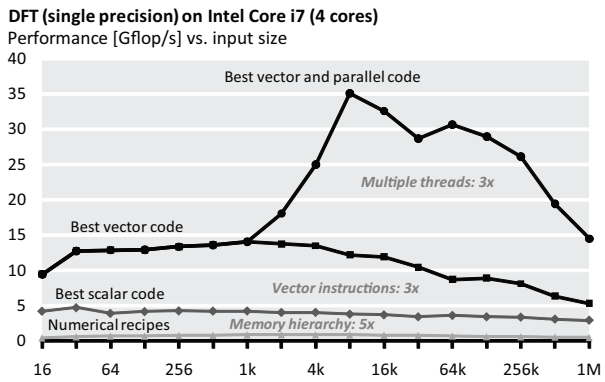


Fig. 8: Performance of four single-precision implementations of the discrete Fourier transform. The operations count is roughly the same. *The labels in this plot are about the smallest you should go.*

Do not include

- Work on infrastructure and testing.
- Work done after the presentation took place.

Example and structure follows.

Marylin. Focused on non-SIMD optimization for the variant 2 of the algorithm. Cache optimization, basic block optimizations, small generator for the innermost kernel (Section 3.2). Roofline plot. Worked with Cary and Jane on the SIMD optimization of variant 1, in particular implemented the bit-masking trick discussed.

Cary. ...

Gregory. ...

Jane. ...

10. REFERENCES

[1] U. T. Zimmermann and T. Lindner, *Train Schedule Optimization in Public Rail Transport*, pp. 703–716, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[2] L. F. Henderson, "On the fluid mechanics of human crowd motion," *Transportation Research*, vol. 8, no. 6, pp. 509–515, 1974.

[3] D. Helbing, "Physikalische modellierung des dynamischen verhaltens von fußgängern (physical modeling of the dynamic behavior of pedestrians)," 1990.

[4] D. Helbing and P. Molnár, "Social force model for pedestrian dynamics," *Phys. Rev. E*, vol. 51, pp. 4282–4286, May 1995.

[5] C. Gloor, "Pedsim," `https://github.com/chgloor/pedsim`, 2012.

[6] F. M. Nasir and S. Mohamad, "Socialforce-model," https://github.com/fawwazbmn/SocialForceModel, 2017.

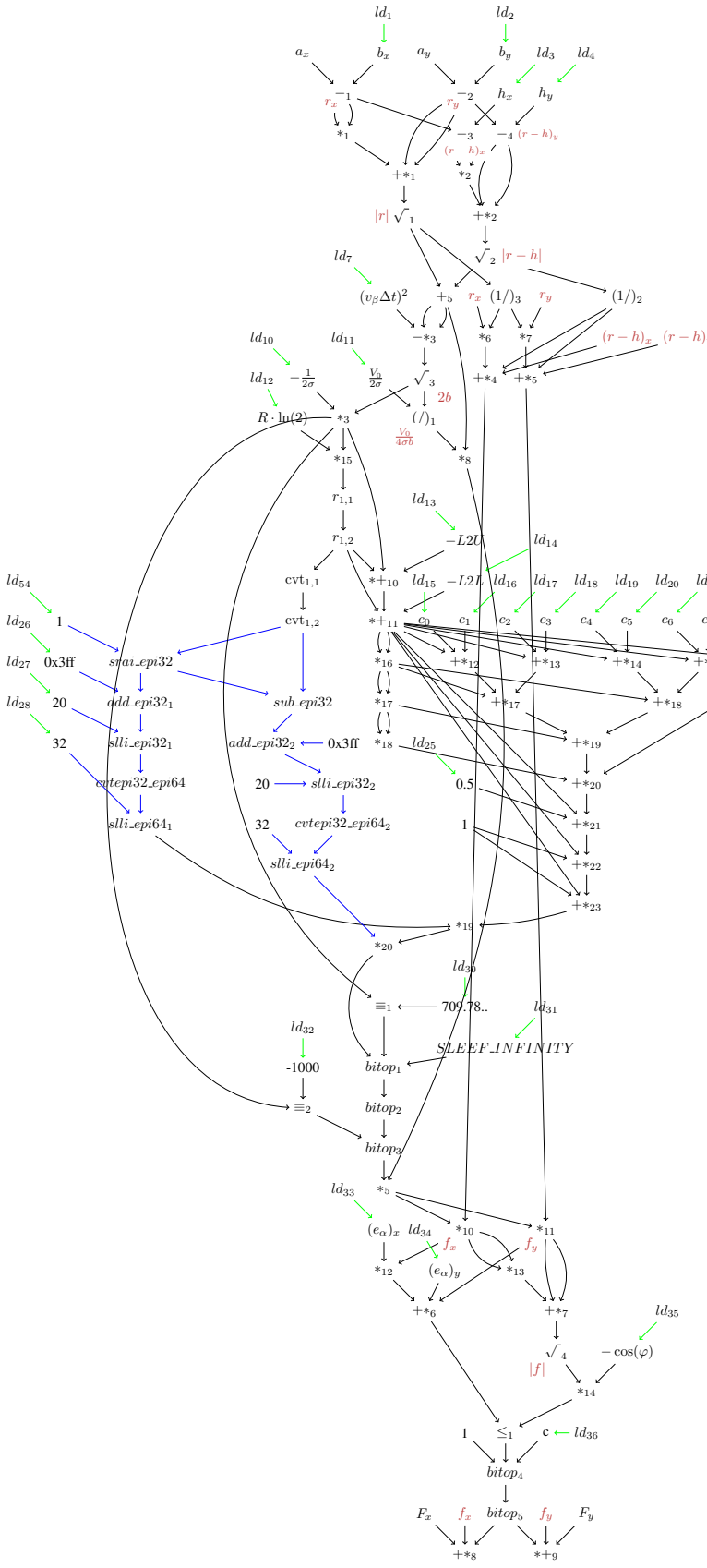
[7] Sven Lauterbach, "Performanceoptimierung von fußgängersimulationen durch einsatz von parallelisierungstechniken," 2017.

[8] Guillaume Payet, "Developing a massive real-time crowd simulation framework on the gpu," 2016.

[9] N.J. Higham, *Handbook of Writing for Mathematical Sciences*, SIAM, 1998.

[10] W. Strunk Jr. and E.B. White, *Elements of Style*, Longman, 4th edition, 2000.

A. TEST


ld1
ld2
ax
bx
ay
by
ld3
ld4
r_x
r_y
h_x
h_y
(r − h)_x
(r − h)_y
|r|
|r − h|
ld7
(v_β Δt)^2
(1/)_1
(1/)_2
(1/)_3
ld10
ld11
ld12
R · ln(2)
2b
ld13
−L2U
ld14
ld15
−L2L
ld16
ld17
ld18
ld19
ld20
ld54
ld26
ld27
ld28
0x3ff
20
32
1
cvt1,1
cvt1,2
r1,1
r1,2
c0
c1
c2
c3
c4
c5
c6
smai_epi32
add_epi32
slli_epi32
cvtepi32_epi64
slli_epi64
sub_epi32
add_epi32_2
slli_epi32_2
cvtepi32_epi64_2
slli_epi64_2
0.5
ld25
ld30
709.78..
ld31
SLEEF_INFINITY
ld32
-1000
bitop1
bitop2
bitop3
ld33
ld34
(e_α)_x
(e_α)_y
f_x
f_y
ld35
− cos(φ)
|f|
1
≤1
c
ld36
bitop4
F_x
bitop5
F_y