A FAST IMPLEMENTATION OF THE SOCIAL FORCE MODEL FOR SIMD SINGLE-CORE SYSTEMS

Matthias Matti, Tommaso Pegolotti, Lino Telschow, Simon Zurfluh

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

In this paper, we present how an efficient implementation of the social force model from Helbing and Molnar can be implemented for SIMD[1] single-core systems. Starting from a straightforward C implementation, different optimization strategies are added incrementally to accelerate the simulation. We identified the bottlenecks of the simulation and propose methods to address them. Our optimizations are applied for modern variants of the x86 architecture family, but can also be adapted for other ISAs[2] which support SIMD instructions. The process is documented and founded with performance benchmarks and other measurements. In the end we achieved a run-time speedup by a factor of about 19 compared to a naive C implementation.

1. INTRODUCTION

As part of the Advanced Systems Lab course at ETH Zürich, we implement a straightforward simulation of pedestrians on a walkway; following the theory in Helbing's social force model [1]. After validating the correctness of the implementation we apply optimization techniques. We compare different methods and their effect on performance and runtime. A final and most improved implementation is given at last.

The model uses differential equations to describe pedestrian movement in space. Because of its nonlinear structure, Helbing's model is a fitting example on how to improve performance with structural changes in the implementation. To increase performance we use optimization methods from the course lecture, such as: precomputation, scalar replacement, and the use of intrinsic instructions. More optimizations are discussed in Chellappa et al [2].

Related work. The social force model presented by Helbing and Molnar is widely used in pedestrian simulation, e.g. in Crowd analysis [3] and detection of abnormal crowd behavior [4]. These simulations have experienced a rising interest in real time applications, where crowds can be analyzed and compared to the model. A parallelized implementation on multi-core machines, that allows real time simulation for tens of thousands of pedestrians was proposed by Quinn et al [5]. In contrast to multi-core systems our focus is only on single-core optimizations.

To the best of our knowledge there has not been a published paper about single-core optimizations in social force simulations. Helbing's social force model was given as a possible subject for the project and therefore multiple groups have chosen to work on the same topic. This means that our implementation is only one way of many to increase the performance and reduce run-time.

2. SOCIAL FORCE MODEL

In this section we give background information on the different parts of Helbing's social force model [1] and provide a cost analysis.

According to the model, pedestrians in large crowds behave as if they are under the influence of a social force that indirectly affects their movements. The model is based on Lewin's idea of social fields [6] that represent the external stimuli level based on individual objectives. The different incentives lead a person to walk in a preferred direction.

Notation. The state of a person α at a time t is described with its *position* $\vec{r}_\alpha(t)$ and its *actual velocity* $\vec{v}_\alpha(t)$. The *speed* $(v_\alpha(t))$ is defined as the euclidean norm of the velocity vector.

People prefer to walk at their own pace. This is simulated by the *desired speed* v^0_α.

A single point is used as a *destination* $(\vec{r}^0_\alpha)$, which is either the final point that a pedestrian wants to reach or an intermediate point in a path that a pedestrian is traversing. A *desired direction* $\vec{e}_\alpha(t)$ is calculated such that pedestrians walk towards their destination.

$$\vec{e}_\alpha(t) := \frac{\vec{r}^0_\alpha - \vec{r}_\alpha(t)}{\|\vec{r}^0_\alpha - \vec{r}_\alpha(t)\|} \tag{1}$$

[1]Single Instruction stream, Multiple Data stream
[2]Instruction Set Architecture

Based on these quantities, Helbing et al.[1] describe terms that determine the movement of a pedestrian. Their implementation considers 3 different terms: an **acceleration** term, a **people repulsion** term and a **border repulsion** term.

Acceleration Term. The *acceleration term* $\vec{F}^0_\alpha(t)$ models the active approach of a pedestrian α to his desired speed at the beginning of the simulation or after a deceleration. It is computed from the difference between the actual velocity and the desired velocity. The relaxation time τ_α allows the pedestrian to smoothly approach its desired velocity.

$$\vec{F}^0_\alpha(\vec{v}_\alpha, v^0_\alpha \vec{e}_\alpha) := \frac{1}{\tau_\alpha}(v^0_\alpha \vec{e}_\alpha(t) - \vec{v}_\alpha(t)) \tag{2}$$

People Repulsion Term. The model describes a repulsive effect in form of a private circle around every person, because people tend to keep a certain distance from each other when moving in space. This effect of a person β on person α is described by:

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

where $V_{\alpha\beta}(b)$ describes a monotonic decreasing function with equipotential lines, and $\vec{r}_{\alpha\beta} := \vec{r}_\alpha - \vec{r}_\beta$. The equipotential lines have the form of an ellipse to provide enough space for the next step of each pedestrian. Furthermore, b denotes the semi-minor axis of the ellipse and is given by:

$$2b := \sqrt{(\|\vec{r}_{\alpha\beta}\| + \|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|)^2 - (v_\beta \Delta t)^2}, \tag{4}$$

where $\vec{r}_{\alpha\beta} := \vec{r}_\alpha - \vec{r}_\beta$.

If pedestrian β is positioned outside α's *angle of sight*, then the people repulsion potential is multiplied by an influence factor c with $0 < c < 1$. This models the weaker influence of events not seen by the pedestrian α and is taken into account by the function w. Thus, the repulsive effect of β on α is given by:

$$\vec{F}_{\alpha\beta}(\vec{e}_\alpha, \vec{r}_{\alpha\beta}) := w(\vec{e}_\alpha, -\vec{f}_{\alpha\beta})\vec{f}_{\alpha\beta}(\vec{r}_{\alpha\beta}) \tag{5}$$

The *people repulsion term* is the sum of the repulsive effects between α and every other person β.

Border Repulsion Term. Similar to the people repulsion term the *border repulsion term* is the sum of the repulsive potentials between α and all the borders that are part of the simulation. The repulsive effect evoked by a border B on a person α is described by:

$$\vec{F}_{\alpha B}(\vec{r}_{\alpha B}) := -\nabla_{\vec{r}_{\alpha B}} U_{\alpha B}(\|\vec{r}_{\alpha B}\|), \tag{6}$$

with $U_{\alpha B}(\|\vec{r}_{\alpha B}\|)$ describing a monotonic decreasing function and $\vec{r}_{\alpha B}$ being the shortest distance between α and border B. Whether the border is in front or behind a person has no effect on the repulsive potential in Helbing's model.

Social Force Model. The sum of the different terms describes the motivation of a person to move in a certain direction at time t.

$$\vec{F}_\alpha(t) := \vec{F}^0_\alpha(\vec{v}_\alpha, \vec{v}^0_\alpha \vec{e}_\alpha) + \sum_\beta \vec{F}_{\alpha\beta}(\vec{e}_\alpha, \vec{r}_{\alpha\beta}) + \sum_B \vec{F}_{\alpha B}(\vec{e}_\alpha, \vec{r}_{\alpha B}) \tag{7}$$

The social force model is then defined by

$$\frac{\vec{w}_\alpha}{dt} := \vec{F}_\alpha(t) + fluctuations \tag{8}$$

where $\vec{w}_\alpha$ is the preferred velocity. The addition of a fluctuation term models random variations of the behavior. The realized motion relates the actual velocity $\vec{v}_\alpha$ and the preferred velocity $\vec{w}_\alpha$,

$$\frac{d\vec{r}_\alpha}{dt} = \vec{v}_\alpha(t) := \vec{w}_\alpha(t) g(v^{\text{max}}_\alpha / ||\vec{r}_\alpha||), \tag{9}$$

which completes the model of pedestrian dynamics. Here, g limits the actual velocity.

For a full description of the model we refer to Helbing et al.[1]

Asymptotic Analysis. Here, we analyze the asymptotic cost for one time step for the social force model as described above.

To simplify the calculation we first analyze the computational cost to simulate one pedestrian α out of n. The desired direction and the acceleration term can be comped in a constant number of instructions. The same holds for the computation of the repulsive potential between two pedestrians or a pedestrian and a wall. The repulsive term however, consists of $n - 1$ repulsive potentials between α and every other pedestrian, and B repulsive potentials between α and every wall in the simulation. This results in an asymptotic cost of $\mathcal{O}(n + B)$ or $\mathcal{O}(n)$ when the number of borders is assumed to be constant for any input size. Performing the integration steps and computing the new position for α is again possible in a constant number of instructions.

In total, the asymptotic run-time cost of the model for 1 person is in $\mathcal{O}(n)$, this results in $\mathcal{O}(n^2)$ when running the computation for n people.

3. IMPLEMENTATIONS

In this section the straightforward implementation and the different optimization techniques used to increase performance and lower run-time are described. After the straightforward implementation our work splits into two parts. The focus of one part lies on optimizations that ignore vectorized instructions, while the other part revolves around SIMD optimizations. To compare both end-solutions the non-SIMD version is vectorized. The best performing implementation is then further refined to the final implementation.

Scenario. The scenario we consider for our implementation is a walkway bordered by two walls. There are two groups of people, the ones who walk from the left side of the walkway to the right side; and the ones who walk in the opposite direction. The walkway has a width of 4 meters and a length of 50 meters, which corresponds to the dimensions given in Helbing et al [1].

Straightforward Implementation. The straightforward version simulates the behavior of the pedestrian for each time step. Social forces are calculated for all persons and based on these the positions are updated: first the desired direction and acceleration terms are computed for each pedestrian. Then the people repulsion terms between each pair of pedestrians are stored in a matrix $\mathbf{R}$ of size $n \times n$. For every pair of persons α, β the matrix entries are defined as

$$R(\alpha, \beta) := \begin{cases} \vec{F}_{\alpha\beta}(\vec{e}_\alpha, \vec{r}_{\alpha\beta}) & \text{if } \alpha \neq \beta \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

Note that the diagonal contains zeros.

Border repulsion terms are computed and stored in the same way. Before updating the position of every person with formula (9), social forces are obtained with formulas (7,8). The repulsive effect of other pedestrians on pedestrian α is calculated by $\sum_\beta R(\alpha, \beta)$.

Constants and repulsive potentials follow the values used in Section IV. of [1].

Integration. To simplify computations, we assume all values to stay constant during a time step. This results in an inaccurate integration scheme for large Δt.

When choosing a value for the time step Δt, some problems arise. Due to numerical inaccuracies and the discrete integration scheme, the model suffers from oscillations when pedestrians get close to each other or arrive at their destinations. To alleviate those issues we decreased the size of Δt to 0.1 seconds and positioned the destinations outside of the walkway area. We refer to [7, 8] for further possibilities to avoid such oscillations.

Integrating $\vec{F}_\alpha(t)$ over the time step results in the preferred velocity $\vec{\omega}_\alpha(t)$ (8). For simplicity, we omit the addition of random fluctuations. Similarly, the new position is the result of integrating $\vec{v}_\alpha(t)$ over Δt again (9).

Testing and Validation. Besides test cases, a finite-differences approach is used to verify the correctness of the implemented functions. Furthermore, a visualization of the simulation helped to observe the behavior of pedestrians and to compare it with the results of Helbing et al.[1]. Similar to the paper, we can observe the phenomenon of people forming lanes as shown in Fig. 1. This straightforward implementation is then used to test optimized versions for correctness.

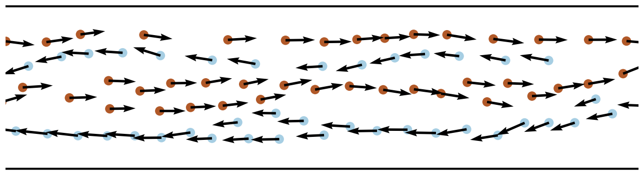

Fig. 1. Snapshot of the visualization. Red points walk from left to right; blue point in the opposite direction. The arrows indicate the velocity of the pedestrians. The formation of horizontal lanes is visible.

Memory Layout and Complexity. Scalar quantities, e.g. the speed values v_α, are stored in arrays of length n, where n is the number of people. Vectorial quantities, e.g. the positions $\vec{r}_\alpha$, are stored in two dimensional arrays of size $2 \times n$. The first row stores the x component and the second row the respective y component. However, to store all people repulsion terms, we need an array with $2n^2$ entries. Aligning all arrays to 32-byte storage boundaries enables efficient use of AVX and AVX2[3] instructions.

The memory complexity of the straightforward implementation is in $\mathcal{O}(n^2)$. To measure the memory usage in practice we used the profiling sub-module Instruments in Xcode [4]. For 2^{10} people we measured a memory footprint of 16 MB and for 2^{14} people we got 4 GB, which shows how memory inefficient the straightforward implementation is.

Run-Time Analysis. The structure of the code suggests that the bottleneck of the algorithm is the computation of the people repulsion terms. Profiling the simulation with 1024 people with gprof [9], callgrind [5], Intel Advisor [6], and Xcode showed that 94% of the run-time is spent in the calculation of these terms.

Exponential Function. Given that AVX and AVX2 does not provide an exponential instruction and repulsive potentials are part of the bottleneck, we decided to implement our own exponential function. Since the inputs to the function are always non-positive, we need an approximation that is accurate and fast for this input range only.

Our implementation is based on the limit definition of the exponential function

$$e^x = \lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n, \tag{11}$$

and is inspired by this blog post. [7] The blog post suggests that the approximation $e^x \approx (1 + \frac{x}{n})^n$ may be suitable for large n if the required input range is small. To validate this, we empirically determined the n that minimizes the absolute error between the standard C library exponential

[3]Advanced Vector Extensions and Advanced Vector Extensions 2
[4]https://developer.apple.com/xcode/
[5]https://www.valgrind.org/
[6]https://software.intel.com/content/www/us/en/develop/tools/advisor.html
[7]https://codingforspeed.com/using-faster-exponential-approximation/

function and this approximation. The maximum absolute error is minimized for $n = 2^{12}$, for arguments in the range $[-200, 0]$. The corresponding maximum absolute error is $5.0 \cdot 10^{-5}$.

Cost Measure. The cost measure counts the number of floating point additions, multiplications, division, square root and exp instructions. The baseline implementation has a cost of

$$\begin{aligned} C(n) := &\ \text{adds}(n) + \text{mults}(n) \\ &+ \text{divs}(n) + \text{sqrts}(n) + C_{exp} \cdot \text{exps}(n), \end{aligned} \tag{12}$$

where C_{exp} is defined as the number of instructions needed to execute an exponential instruction.

3.1. Standard C Optimizations

This section presents the optimizations without SIMD instructions. Our approach was to apply different techniques to the straightforward implementation, and then splitting up to explore different structures and unrolling factors. The final version of each split is described more in detail.

The first step was to apply simple arithmetic optimization such as precomputation of constant values and transformation of divisions to multiplications. However, these techniques could not be applied extensively to our problem due to the low amount of constant factors.

The following step was to inline all the function calls, merge the loops that iterate over each person and reorder the computation. This new structure made it possible to reuse loaded values and reduce the number of total load and store instructions, thus increasing the temporal locality of the algorithm.

Because of this restructuring the optimized implementations, compared to the basic version, do not use a matrix to store the social force terms. After computing the terms locally, they are directly added to the social force applied to the person. This reduces the memory complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$.

These two optimization steps led us to two different structures which later developed into stdc_2_4 and stdc_2_5_1.

Stdc_2_4. Stdc_2_4 consists of only one for-loop per time step. First the social force is calculated and then the position and desired direction are updated. With this approach individual values of a pedestrian can be loaded from memory at the beginning and stored locally. While calculating the social force and the new position the values of this pedestrian remain local and can be reused without additional memory accesses.

However a drawback of this approach is the interference of already updated positions when calculating the people repulsion terms. In the straightforward implementation we solved this by calculating the new positions after the social forces of all pedestrians were computed. In stdc_2_4 this is not possible anymore. Therefore additional arrays are needed to store the state of the model at the previous time step, while the state of the current time step is gradually updated.

Every second time step reads the values directly from the temporary arrays, this prevents additional memory traffic. Because otherwise the old information must be overwritten with new data after every time step.

Stdc_2_5_1. Stdc_2_5_1 consists of two main phases, first the magnitude of the interaction between pedestrians is computed, then the position is updated accordingly.

Since on Coffee Lake micro-architecture multiplications, additions and FMAs[8] have a throughput of 2 and a latency of 4 cycles, we settle on using an unrolling factor of 8 and 8 accumulators. After various trials we could confirm that indeed it gives the best performance. Because of this unrolling factor a simplifying assumption we use is that the input needs to be a multiple of 8.

In order to optimize the computation of the social force we refactored the algorithm by considering $\vec{F}_{\alpha\beta}(\vec{e}_\alpha, \vec{r}_{\alpha\beta})$ and $\vec{F}_{\beta\alpha}(\vec{e}_\beta, \vec{r}_{\beta\alpha})$ simultaneously rather than calculating the terms individually. This lets us reuse values and, together with the accumulators, improves the locality of the implementation since only the values of one person β need to be loaded for 8 pedestrians (α_1 to α_8).

3.2. SIMD Optimizations

This section presents the optimizations with SIMD using the AVX and AVX2 instruction set [9]. Our approach was to start with an initial vectorization of the straightforward implementation and then optimize this version incrementally. The result of this are 3 SIMD versions, where vectorize_1 is the vectorization of the straightforward implementation and vectorize_3 is the most optimized version. In order to compare the non-SIMD to vectorize_3, the best standard C version stdc_2_5_1 was also vectorized.

Assumptions. For simplicity we only considered input sizes where the number of people is divisible by the number of elements fitting in a SIMD register, i.e. 4 for doubles and 8 for floats. Moreover, as mentioned in the previous section, all arrays are aligned to 32-byte memory boundaries such that SIMD instructions can be used efficiently.

Doubles vs. Floats. Up to this point all versions were implemented with doubles. When vectorizing the straightforward implementation, we learned that it may be advantageous to use floats. The use of floats allows more parallelism because 8 instead of 4 numbers fit into a 256-bit vector register. Besides that, there exist AVX instructions which are only available for floats, e.g. rsqrt_ps, rcp_ps, and may be useful for optimizing our code. Indeed, the cal-

[8]Fused multiply–add

[9]https://software.intel.com/sites/landingpage/IntrinsicsGuide/

culation of repulsion terms between people includes both norm calculations and vector normalization, which require the use of expensive square root and division operations, i.e. sqrt_pd/ps and div_pd/ps. By using rsqrt_ps, which approximates the reciprocal square root, and rcp_ps, which calculates the reciprocal, aforementioned operations can be computed more efficiently considering latency and gap/inverse throughput, see Table 1.

Instruction	**Latency [Cycles]**	**Gap [Instr. / Cycle]**
sqrt_ps	12	3
sqrt_pd	15-16	4-6
div_ps	11	3
div_pd	13-14	4
rsqrt_ps	4	1
rcp_ps	4	1

Table 1. Latency and gap of AVX square root and division instructions operating on 256-bit vector registers on Intel's Coffee Lake micro architecture. [10]

The disadvantage of floats compared to doubles is the lower precision and the smaller range of possible values. However, extensive tests have shown that the maximum error occurring in a simulation step is within a reasonable range. Furthermore, the simulation does not express different phenomena when using floats. This is the reason why we decided to focus on SIMD versions with floats.

In the following the incremental optimization process of these SIMD versions is described.

Vectorize_1. This version is a direct vectorization of the straightforward implementation. The only non-trivial aspect in this initial vectorization is the computation of the people repulsion terms. The corresponding pseudo code is given in Algorithm 1. To avoid if-statement in the innermost loop, the diagonal entries are set to 0 in a separate loop.

Algorithm 1: Implementation of Eq. (10), single precision SIMD version

```
for $\alpha = 0$; $\alpha < n$; $\alpha += 1$ do
    for $\beta = 0$;$\beta < n - 7$; $\beta += 8$ do
        R($\alpha$,$\beta : \beta + 7$) = $\vec{F}_{\alpha\beta:\beta+7}(\vec{e}_\alpha, \vec{r}_{\alpha\beta:\beta+7})$
for $\alpha = 0$; $\alpha < n$; $n += 1$ do
    R($\alpha$,$\alpha$) = 0 // diagonal entries
```

Vectorize_2. This version uses vectorize_1 as base and performs optimizations on the level of intrinsics functions. FMAs are used as much as possible to get rid of chained additions and multiplications. As mentioned above, the use of rsqrt_ps and rcp_ps allows to omit expensive square root and division instructions and calculate norms and normalize vectorial quantities more efficiently.

Vectorize_3. In previous versions, the people repulsion terms were stored in an array with $2n^2$ entries. To be able to use larger input sizes, we wanted to reduce the memory footprint of the algorithm to $\mathcal{O}(n)$. This required a way to calculate $\sum_\beta \vec{F}_{\alpha\beta}(\vec{e}_\alpha, \vec{r}_{\alpha\beta})$ without the matrix R. With the help of an accumulator register these sums can be pre-computed and later added to the social force. By separating the computation of the diagonal terms from the off-diagonal terms, we managed to vectorize the whole computation of the people repulsion terms using only an array of length $\mathcal{O}(n)$. Since we are using SIMD registers, 8 terms $\{\vec{F}_{\alpha\beta}, \dots \vec{F}_{\alpha\beta+7}\}$ can be calculated at the same time. However, to keep expensive branching and shuffle instructions to a minimum, the calculation of terms close to the diagonal, i.e. terms on a 8×8-block-diagonal, was separated from the calculation of the off-block-diagonal terms. This optimization reduces the number of data loads and stores considerably and allows the cache to be used more efficiently. Moreover, the accumulator register helps to increase temporal locality and avoid stores in the innermost-loop.

Reducing the memory complexity to $\mathcal{O}(n)$ allowed us to run the simulation for larger input sizes. According to Xcode, the memory usage of vectorize_2 for $8 \cdot 3^6$ people, which is the maximal input size we used for the benchmarks for the versions with a memory complexity of $\mathcal{O}(n^2)$, was 260 MB. Using the same input size for vectorize_3 lead to a memory usage of 0.92 MB. Moreover, we were able to run the code for input sizes up to $8 \cdot 3^9 \approx 157'000$ people. Here, the limiting factor for the input size was the run-time, since it took multiple hours to complete the benchmark. The memory usage for vectorize_3 was only 21.3 MB.

Vectorize_2_5_1. This version is the vectorization of the non-SIMD version stdc_2_5_1. Since stdc_2_5_1 uses an unrolling factor of 8, the vectorization was straightforward. The only intricate detail was the computation of people repulsion terms close to the diagonal. The block-diagonal approach as in vectorize_3 was used to compute them. Moreover, we used as many FMA and rsqrt_ps instructions as possible.

3.3. Most Optimized Version

In this section we discuss which optimizations we used to implement the final and most optimized version.

Before applying the last optimization steps, we had to decide which version should serve as the code base. We had to choose between vectorize_2_5_1 and vectorize_3. We decided in favour of vectorize_3, since its performance was higher by 2 Flop/Cycle and its run-time lower.

Vectorize_4. Similar as in the standard C optimizations, merging loops helped to reuse values and increase temporal locality. By reordering some operations in the innermost loop of the people repulsion calculation, we were able to reduce the length of the critical path of the calculation and increase instruction level parallelism.

4. EXPERIMENTAL RESULTS

In this section we evaluate and analyze our optimized implementations.

Experimental setup. The system we used to run our experiments on is specified in Table 2. The benchmarks were executed on MacOS. For the purpose of generating roofline plots we used Intel Advisor on Windows, since it only runs on Windows and GNU/Linux. Using two systems is not a problem since Intel Advisor uses it's own model to measure performance, which is not directly comparable to the one used in our benchmarks. However, the performance measurements of both systems deliver very similar results. To avoid CPU throttling, we run all benchmarks at base frequency (i.e. disabled Turbo Boost).

CPU:	Intel Core i7-9750H
Micro architecture:	Coffee Lake
Number of cores:	6
Number of threads:	12
CPU base frequency:	2.6 GHz
CPU max. frequency:	4.5 GHz
Turbo Boost support:	yes
L1 data cache:	32 KB, 8-way set associative
L2 cache:	256 KB, 4-way set associative
L3 cache:	12 MB, 16-way set associative
OS:	MacOS 10.15.4
OS (Intel Advisor):	Windows 10 Pro 64 bit 1909
C compiler:	gcc 9.2.0
compiler flags:	-O3 -mavx2 -mfma -ffast-math

Table 2. Specification of the test system

Input sizes. Versions vectorize_3, vectorize_4 and vectorize_2_5_1 were benchmarked for input sizes up to $8 \cdot 3^9$ people, the others up to $8 \cdot 3^6$ people. The input sizes for the straightforward version, vectorize_1 and vectorize_2 are limited by their memory footprints of $\mathcal{O}(n^2)$: not enough memory could be allocated at run-time. For the other versions, the run-time was the limiting factor as it took multiple hours to complete a benchmark.

We also timed the implementations with powers of two as input sizes. However, we recognized that these inputs are not suitable for benchmarking, since an aliasing effect can be observed for large input sizes. This issue is discussed in subsection "Strided Memory Access".

Benchmarking. The benchmark framework we have built performs two phases. It first performs a calibration step in which it finds the amount of runs that are needed to measure a minimum number of cycles for a simulation. This aims to minimize fluctuations of the measurements for faster runs. After the calibration, the simulation is timed repeatedly using the Time Stamp Counters (TSC) with the tsc_x86[10] header. Since the number of flops is constant for each time step we have settled on running the simulation for 20 time steps in order to keep the benchmark run-time contained. Once these simulations are ran, we consider the median of the measurements.

Results. We now discuss the results obtained after running the implementations on the specified system and framework. We use two different straightforward implementations as baselines: one of them uses doubles and the other uses floats. The reason for this is that we want to compare optimized versions only to the straightforward version that uses the same precision, i.e. the SIMD versions with the straightforward implementation using floats and the non-SIMD versions with the straightforward implementation using doubles.

The straightforward implementation reaches for all input sizes a performance of around 0.60 Flop/Cycle using doubles and 0.45 Flop/Cycle using floats. The two versions reach 16% and 11% respectively of the theoretical scalar FMA peak performance of 4 Flop/Cycle.

First, we look at the standard C optimized versions and compare them to the straightforward implementation. These versions all use doubles. As can be seen in Fig. 2, both stdc_2_4 and stdc_2_5_1 are almost twice as fast compared to the straightforward implementation.

As already mentioned the difference between these two version resides in their structure which causes different memory access patterns. We believe that the difference in performance between stdc_2_4 and stdc_2_5_1 is due to the higher instruction level parallelism given by the unrolling factor and the better spatial locality obtained by keeping the computation of the social force and the update of the position separated. Version stdc_2_5_1 reaches 27% of the theoretical scalar FMA peak performance.

Next, we discuss the results of the SIMD versions and the final most optimized implementation as shown in Fig. 3. These versions are all based on floats. As mentioned earlier, the less optimized versions have been benchmarked for input sizes up to $8 \cdot 3^6$, due to the high memory usage due to the quadratic people repulsion array. Despite the high memory consumption, vectorize_1 has about a 13 times higher performance than the straightforward version. Furthermore, we see with vectorize_2, that using FMAs and rsqrts increases the performances by more than one 1 Flop/Cycle.

[10]https://acl.inf.ethz.ch/teaching/fastcode/2020/homeworks/hw01files/tsc_x86.h

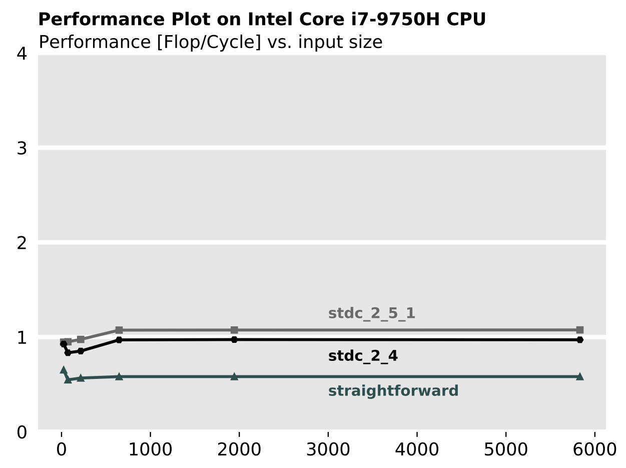

Fig. 2. Performance of the final two optimized implementations without SIMD instructions. Compared to the straightforward implementation both optimized versions achieved a speedup of about 2x.

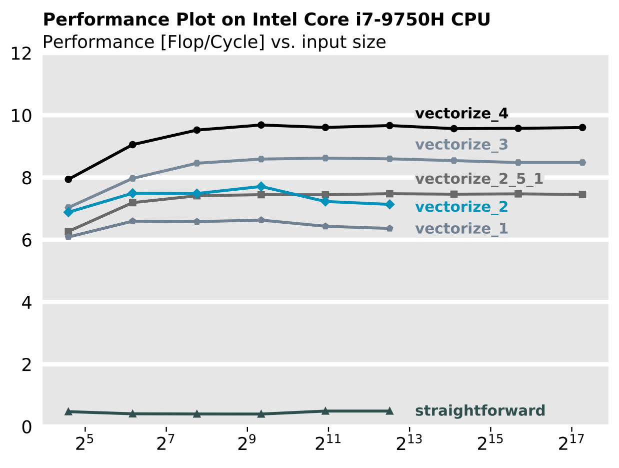

Fig. 3. Performance of all vectorized implementations. The versions 1, 2 and straightforward cannot be benchmarked for higher input sized due to their quadratic memory usage. The FMA SIMD peak performance is not shown here for visual reasons.

The implementation vectorize_2_5_1 has a slight performance increase compared to vectorize_2, but is slower than vectorize_3. A reason for this difference in performance is the implementation of how the people repulsion term is computed. Version vectorize_2_5_1 only iterates over the upper triangular matrix and directly computes and stores the values from the lower triangular matrix. However, this approach implies store instructions in the innermost loop, which may cause the performance difference compared to vectorize_3. Vectorize_3 overcomes this issue by separately computing the values for the upper and lower triangular matrix and therefore only needs to add up the results locally.

Another performance increase of more than 1 Flop/Cycle is possible with the changes made in vectorize_4. The final implementation reaches about 30% of the theoretical SIMD FMA peak performance for floats of 32 Flop/Cycle.

Switching our focus now from performance to run-time, we can see from Fig. 4 the relative speedup from the straightforward implementation to stdc_2_5_1 and vectorize_4. In particular we see that our best standard C implementation is twice as fast as the basic version and our final version reaches a peak of 24 times for smaller inputs and 19 times for bigger sizes. The run-time for non-vectorized implementations can be decreased further by cutting of the computation of the repulsion forces when the interacting parties are too far away from each other. We added this technique to the straightforward version and observed a reduction in run-time. The gained speedup, however, is dependable on the interactions between pedestrians and therefore hard to quantify.

Roofline Model. To estimate the value of our optimizations we have used the Intel Advisor Tool, it implements a cache-aware roofline model, presented in [11]. This means it counts all data movements from L1, L2, L3-caches and main memory. Therefore, the arithmetic intensity stays constant for a given algorithm. This stays in contrast to the classical roofline model, which only measures memory traffic. Thus, the operational intensity might vary with changing problem size or when cache optimizations are applied.

In addition to this, Intel Advisor empirically estimates theoretical peak performances for scalar and vector instructions, as well as bandwidth estimates for L1, L2, L3-caches and main memory.

From Fig. 5 we can infer that the optimizations we applied to stdc_2_5_1 and vectorize_4 let us overcome the DRAM bound and ultimately made them L3 bound. The performance values are consistent with our benchmarks and do not need additional explanation.

Further measurements with Intel VTune showed that, considering the same number of people, the number of load and store instructions is higher in stdc_2_5_1 than in the straightforward version. This behavior may be caused by the high number of variables used, that increase the number of register spills. However these memory management instructions operate on data stored in cache, hence we observe a higher performance despite having lower arithmetic intensity.

Strided Memory Access. In this paragraph we describe why we have chosen to sample the input size from $\{8 \cdot 3^n \mid n \in \mathbb{N}\}$. We started by using powers of 2 instead of 3. When timing versions vectorize_3, vectorize_4 and vectorize_2_5_1, we observed a sudden performance drop of up to 1.5 Flop/Cycle for an input size of 2^{14}.

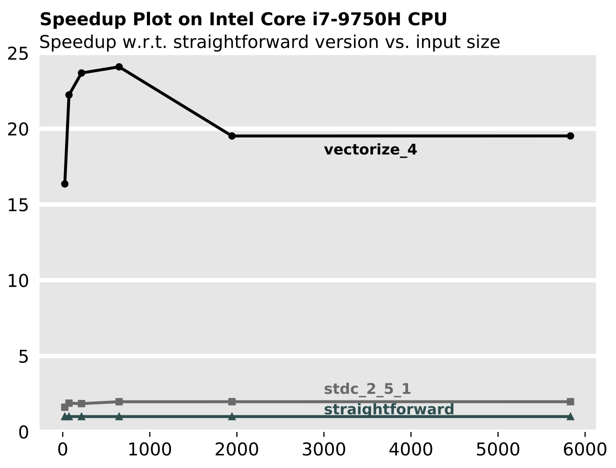

Fig. 4. Speedup of stdc_2_5_1 and vectorize_4 compared to their respective straightforward implementations.

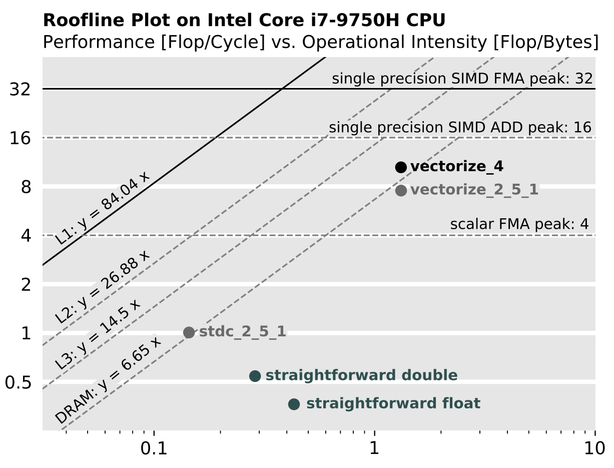

Fig. 5. Cache aware roofline model plotting the performance and arithmetic intensity of some of our best versions compared to both baseline implementations.

As described in Section 3, vectors are stored as two dimensional arrays of size $2 \times n$, where the x components reside in the first and the y components in the second row of the array. Since the L2 cache has a size of 256 KB, a cache line size of 64 bytes and is 4-way-set-associative, the corresponding cache lines for the x and y components of respective vector map to the same cache set in the L2 cache, when the input size is a multiple of 2^{14}. Since there are multiple vector arrays, our hypothesis is that this strided access pattern causes an increase of conflict misses in the L2 cache. Measurement of the cache hit/miss rates with Intel VTune match our hypothesis. To further test the hypothesis we timed above versions for input sizes in $8 \cdot 3^k \mid k \in \{0, \dots 9\}$, where corresponding x and y components do not map to the same cache set in the L2 cache. As visible in Fig. 3, no performance drop is observable. From these observations we concluded that the memory layout is not optimal and using separate arrays for respective components would solve this issue. However, as we encountered this problem towards the end of the project, we did not have enough time to change the memory alignment, as this would have resulted in a change of the whole setup. In order to benchmark our optimizations in a way, such that they can show their full potential, we decided to use input sizes $8 \cdot 3^k, k \in \{0, \dots 9\}$.

Experiments with different compilers. Besides using gcc 9.2.0 we also tried icc 19.1.1 and clang 11.0.3 compilers in combination with different flags.

Clang delivered a significantly lower performance for smaller inputs, but reached the same level of performance given by gcc once the input reached a size of approximately 1.7k people.

Regarding icc we were mainly interested in using the SVML instructions set and its mathematical functions to further boost our performance. However, the use of these instructions did not match our expectation as the performance was lower than by using our implementations.

In conclusion, the binaries produced by both of these compilers did not deliver a significant performance difference, hence we decided to only use results obtained via gcc for consistency.

5. CONCLUSIONS

The systematic approach described in this paper shows the effectiveness of single-core optimizations.

Often a straightforward implementation can not be thoroughly optimized by the compiler alone, therefore it is important to restructure code and algorithm to promote the application of more effective optimizations.

Our straightforward implementation of Helbing and Molnar's social force model includes optimization blockers, high dependency trees and computationally intensive instructions, e.g. divisions, square roots and exponentials.

Non-vectorized optimization techniques achieved 27% of scalar FMA peak performance and a 2 times run-time speedup compared to a straightforward implementation. By including vectorization, we were able to increase the run-time speedup up to 19 times and reach 30% of the SIMD FMA peak performance for floats.

6. CONTRIBUTION OF TEAM MEMBERS

Matthias. Implemented the border repulsion function in straightforward implementation. Used run-time profiling tools, i.e. gprof,callgrind. Worked on SIMD versions vectorize_1, 2 and 3 with Lino. In particular, focused on using an accumulator register for the computation of the people repulsion terms in vectorize_3, described in 3.2. Worked together with Tommaso to find an appropriate exponential function suitable for vectorization, described in Section 3. Built final vectorize_4 with Tommaso. Roofline Plot. Python script for benchmarks, visualization. Analyzed the influence of random seeds on calculation error.

Tommaso. Implemented the computation of the repulsion term among pedestrians in the straightforward version. Built testing environment for unit testing and checking correctness of optimizations. Wrote the benchmarking framework. Worked together with Simon on standard C optimizations and later finalized version 2.5.1. Finalized vectorized version 4 with Matthias. Roofline plot. Profiled using XCode. Worked on the python scripts used for plotting.

Lino. Implemented some functions in the straightforward implementation. Worked together with Matthias on the vectorized versions. Analyzed with Matthias the precision of floats and doubles for the simulation. Was responsible for executing all benchmarks. Worked with Intel Advisor to generate roofline plot data. Used Intel VTune and Xcode to analyze the memory usage and the memory access patterns.

Simon. Implemented the acceleration term and the integration steps of the straightforward version. Worked together with Tommaso on the non-vectorized optimizations and later finalized the version 2.4. Restructured the implementation of the people repulsion term. Restructured initialization to enable optimizations. Analyzed the influence of random seeds on calculation error.

7. REFERENCES

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

[2] Srinivas Chellappa, Franz Franchetti, and Markus Püschel, "How to write fast numerical code: A small introduction," in *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 2007, pp. 196–259.

[3] Weiliang Zeng, Peng Chen, Hideki Nakamura, and Miho Iryo-Asano, "Application of social force model to pedestrian behavior analysis at signalized crosswalk," *Transportation research part C: emerging technologies*, vol. 40, pp. 143–159, 2014.

[4] R. Mehran, A. Oyama, and M. Shah, "Abnormal crowd behavior detection using social force model," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 935–942.

[5] Michael J Quinn, Ronald A Metoyer, Katharine Hunter-Zaworski, et al., "Parallel implementation of the social forces model," in *Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics*, 2003, vol. 63, p. 74.

[6] Kurt Lewin, *Resolving social conflicts and field theory in social science.*, American Psychological Association, 1997.

[7] Tobias Kretz, "On oscillations in the social force model," *Physica A: Statistical Mechanics and its Applications*, vol. 438, no. C, pp. 272–285, 2015.

[8] Gerta Köster, Franz Treml, and Marion Gödel, "Avoiding numerical pitfalls in social force models," *Physical Review E*, vol. 87, no. 6, pp. 063305, 2013.

[9] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick, "Gprof: A call graph execution profiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, New York, NY, USA, 1982, SIGPLAN '82, p. 120–126, Association for Computing Machinery.

[10] Agner Fog et al., "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus," *Copenhagen University College of Engineering*, vol. 93, pp. 383, 2019.

[11] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

A fast implementation of the social force model for SIMD single-core systems

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Matti	Matthias
Pegolotti	Tommaso
Telschow	Lino
Zurfluh	Simon

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
Zürich, 12.6.2020	[illegible]
	Tommaso Pegolotti
	[illegible]
	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.