SFM-OPT: CODE-LEVEL OPTIMIZATION ON SOCIAL FORCE MODEL IMPLEMENTATION

Jiajun Jiang, Kehong Liu*, Jiaqing Xie*, Peiyuan Xie**

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Social force model simulates pedestrians' behaviours based on environmental forces. Difficulties lie in the implementation of the origin paper describing social force model dynamics, and the optimization on the implementation. In this paper, we propose SFM-OPT, a bottom level optimization on social force model, aiming to accelerate the algorithm from the perspective of data structure and functional parts. Trials such as inlining, pre-computation, strength reduction, changing to structure of arrays, blocking of matrices, unrolling and using single instruction multiple data, AVX intrinsics in particular. Finally we show that our optimization has reached approximately 72 % of the maximum theoretical performance and is 4 times faster than baseline implementation.

1. INTRODUCTION

Motivation. The Social Force Model (SFM) was motivated by the requirement for a dynamic particle model that can represent the pedestrian flows and individual movement under a variety of different situations accurately [1]. SFM takes into account both social and psychological factors in addition to dynamic physics components. Group movements, individual desired movement directions, and interactions among pedestrians (repulsive and attractive) are considered as part of a joint model. The idea is followed by the gaskinetic pedestrian model [2] and force field analysis by Lewin [3]. The significance of the SFM stems from its broad practical applications. For example, it plays a pivotal role in shaping the infrastructure of public pedestrian spaces like pavements, and subway stations. It is also crucial for managing crowd evacuation from various directions under emergencies. Therefore it's essential to realize a faster real-time SFM implementation in order to prevent congestion and maintain smooth flow in time. Besides, expediting the implementation of SFM can ensure that the algorithm doesn't become a bottleneck of the system performance even when dealing with large numbers of pedestrians.

*Equal Contribution to this project

Implementing SFM poses significant challenges, primarily due to the absence of an open-source implementation of the original paper. Consequently, it becomes necessary for us to develop the source code from scratch. Additionally, the baseline implementation's time complexity is known to be $O(n^2)$, which corresponds to the approximate number of calculations required for evaluating social forces between each pair of pedestrians. As the input size increases, the execution time grows polynomially, potentially leading to a bottleneck caused by memory limitations. Another obstacle lies in the data structure utilized by the baseline implementation, which is difficult to be vectorized. It involves intricate mathematical expressions, such as calculating exponential functions and inverse square roots. These difficulties have motivated us to explore further optimizations for the baseline implementation.

Contribution. Initially, we implemented the baseline model, incorporating *acceleration terms* (desired movement directions), *repulsive forces* from pedestrians and walls. Similar to the simulation settings in the original paper, we omitted the attractive terms between pedestrians. To identify potential bottlenecks within the function, we examined the time elapsed for each term by utilizing profiling tools like VTune (or Intel Advisor) while providing the complete cache. Subsequently, we conducted preliminary optimizations on these terms by carefully selecting appropriate flags without employing vectorization. Considering the challenges posed by the un-vectorized data structure, even with vectorization flags, we attempted to transform the array of structures (AoS) into a structure of arrays (SoA). This adjustment aimed to enhance the spatial locality. Following this, we employed AVX intrinsics (SIMD) to enable vectorization and further optimize the implementation by outloop unrolling.

Related work. The former version of SFM is the sole reliance on gas-kinetic and fluid-dynamic traffic models [4], which are physically inspired. There are also several extended versions of the original SFM, which expand upon the basic SFM by incorporating the walking behavior of pedestrian social groups and simulating crowd stampedes induced by panic [5]. Nowadays, there are even deep learning mod-

els that incorporate social force modeling and simulate it using computer graphics [6]. Some libraries implementing these extensions are written in Go and Python. However, our implementation does not depend on these existing extensions or improvements but rather focuses on the original version of SFM implemented in C/C++. We have chosen to use C/C++ since they allow us to have better performance and have the possibility to use intrinsics instructions.

2. BACKGROUND ON THE ALGORITHM/APPLICATION

Acceleration term. Suppose that one pedestrian wants to reach the destination $\vec{r}_\alpha^0$, where the shortest path is applied. $\vec{r}_\alpha^*$ is the next edge of the path to reach. $\vec{r}_\alpha(t)$ is the actual position of pedestrian α at time t, the desired direction would be:

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

If there exists no disturbance to this directional behaviour, the desired velocity is given by $v_\alpha^0 \vec{e}_\alpha(t)$, where v_α^0 is the initial velocity. Therefore it leads to a difference between **desired** velocity $v_\alpha^0 \vec{e}_\alpha(t)$ and **actual** velocity $\vec{v}_\alpha(t)$. Within time τ_α, the acceleration term is given by (a = dv/dt):

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

Repulsive Force From Pedestrians. A pedestrian feels uncomfortable when he gets close to another pedestrian who is not familiar with, which leads to a repulsive effect which is defined by:

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

where $V_{\alpha\beta}$ is decreased monotonically when b is increased. The equipotential lines are ellipses which is directed into the movement. b is defined as the semi-minor axis of the ellipse, which 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 $\vec{r}_{\alpha\beta} = \vec{r}_\alpha - \vec{r}_\beta$, and $v_\beta \Delta t$ is the step length of the pedestrian β in time Δt. Under the simulation scheme, we assume $V_{\alpha\beta}(b) = V_{\alpha\beta}^0 e^{-b/\sigma}$, then:

$$\begin{aligned} \vec{f}_{\alpha\beta}(\vec{r}_{\alpha\beta}) &= -\nabla_{\vec{r}_{\alpha\beta}} V_{\alpha\beta}(b) \\ &= -\nabla_{\vec{r}_{\alpha\beta}} V_{\alpha\beta}^0 e^{-b/\sigma} \\ &= \frac{V_{\alpha\beta}^0}{\sigma} e^{-b/\sigma} \nabla_{\vec{r}_{\alpha\beta}} b \end{aligned} \tag{5}$$

$\nabla_{\vec{r}_{\alpha\beta}} b$ requires partial differentiation analysis while other terms in the equation is known. The induction is given below:

$$\begin{aligned} \nabla_{\vec{r}_{\alpha\beta}} b &= \nabla_{\vec{r}_{\alpha\beta}} \frac{\sqrt{(\|\vec{r}_{\alpha\beta}\| + \|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|)^2 - (v_\beta \Delta t)^2}}{2} \\ &= \frac{(\|\vec{r}_{\alpha\beta}\| + \|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|)(\frac{\vec{r}_{\alpha\beta}}{\|\vec{r}_{\alpha\beta}\|} + \frac{\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta}{\|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|})}{2[(\|\vec{r}_{\alpha\beta}\| + \|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|)^2 - (v_\beta \Delta t)^2]^{\frac{1}{2}}} \end{aligned} \tag{6}$$

, which could be further simplified as:

$$\nabla_{\vec{r}_{\alpha\beta}} b = \frac{(\|\vec{r}_{\alpha\beta}\| + \|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|)(\frac{\vec{r}_{\alpha\beta}}{\|\vec{r}_{\alpha\beta}\|} + \frac{\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta}{\|\vec{r}_{\alpha\beta} - v_\beta \Delta t \vec{e}_\beta\|})}{4b} \tag{7}$$

Repulsive Force From Borders. A pedestrian feels uncomfortable when he gets close to buildings or walls that he might get hurt with. It leads to an additional repulsive term from border B which is given by:

$$\vec{F}_{\alpha B}(\vec{r}_{\alpha B}) := -\nabla_{\vec{r}_{\alpha B}} U_{\alpha B}(\|\vec{r}_{\alpha B}\|) \tag{8}$$

In simulation, we assume that

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

then:

$$\begin{aligned} \vec{F}_{\alpha B}(\vec{r}_{\alpha B}) &= -\nabla_{\vec{r}_{\alpha B}} U_{\alpha B}^0 e^{-\|\vec{r}_{\alpha B}\|/R} \\ &= \frac{U_{\alpha B}^0}{R} \cdot e^{-\|\vec{r}_{\alpha B}\|/R} \cdot \frac{\vec{r}_{\alpha B}}{\|\vec{r}_{\alpha B}\|} \end{aligned} \tag{9}$$

Direction dependence. Those repulsive forces only hold when the pedestrians' behaviours are in the desired direction of motion. When another pedestrian is located behind a pedestrian, it will have a weaker influence on this current pedestrian. We take into account the direction-dependent weights:

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

Then it's possible to rewrite the above repulsive effects from pedestrians and borders:

$$\vec{F}_{\alpha\beta}(\vec{e}_\alpha, \vec{r}_\alpha - \vec{r}_\beta) = w(\vec{e}_\alpha, -\vec{f}_{\alpha\beta}) \vec{f}_{\alpha\beta}(\vec{r}_\alpha - \vec{r}_\beta) \tag{11}$$

$$\vec{F}_{\alpha B}(\vec{e}_\alpha, \vec{r}_\alpha - \vec{r}_B) = w(\vec{e}_\alpha, -\vec{f}_{\alpha B}) \vec{f}_{\alpha B}(\vec{r}_\alpha - \vec{r}_B) \tag{12}$$

We sum up those social force terms which would result in an overall acceleration term:

$$\vec{F}_\alpha(t) = \vec{F}_\alpha^0(\vec{v}_\alpha, v_\alpha^0 \vec{e}_\alpha) + \sum_\beta \vec{F}_{\alpha\beta}(\vec{e}_\alpha, \vec{r}_\alpha - \vec{r}_\beta) + \sum_B \vec{F}_{\alpha B}(\vec{e}_\alpha, \vec{r}_\alpha - \vec{r}_B) \tag{13}$$

Note that compared with the original acceleration term presented in the paper, only the attractive term is ignored.

Cost Analysis. We define n_p as the number of pedestrians, and n_B the number of borders. We decide to use the number of all floating point operations as our cost measure for performance analysis, including exponential and square root. Based on this metric, 6 flops would be needed for calculating acceleration term, 11 flops for repulsive effects from each border, and 48 flops for the repulsive effects from each of the other pedestrians. Summing them up yields $50n_p^2 + 13n_pn_B - 42n_p$ flops regarding the whole algorithm. It is now very obvious that the bottleneck lies in the computation of repulsive effects between pedestrians.

For other metrics introduced in class, such as counting each kind of operation separately, if ignoring the lower order terms, the cost should be:

$$C(n_p) = (15n_p^2) \cdot C_{add} + (24n_p^2) \cdot C_{mul} + (4n_p^2) \cdot C_{div} + (4n_p^2) \cdot C_{sqrt} + (n_p^2) \cdot C_{exp} \quad (14)$$

Asymptotic Complexities. Based on the analysis above, the asymptotic runtime for our algorithm is $O(n^2)$. Since each pedestrian only needs the space of a constant size to store all the information needed for the computation, the asymptotic spatial complexity for our algorithm is $O(n)$.

Algorithm 1: SFM 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 $\vec{v}_i^0, \vec{r}_i^0, \vec{e}_i, v_i^{max}, v_i^0$;
for $t \leftarrow 1$ **to** n_{iter} **do**
  for $i \leftarrow 1$ **to** n_p **do**
    Reset total social force: $\vec{F}_i \leftarrow 0$;
    Compute acceleration term:
    $\vec{F}_i \leftarrow \vec{F}_i + \frac{1}{\tau}(v_i^0\vec{e}_i - \vec{v}_i^t)$;
    Compute repulsive effects from pedestrians:
    $\vec{F}_i \leftarrow \vec{F}_i + \sum_{j=1,j\neq i}^{n_p} -w_{ij}\nabla_{\vec{r}_{ij}^t} V[b(\vec{r}_{ij}^t)]$;
    Compute repulsive effects from borders:
    $\vec{F}_i \leftarrow \vec{F}_i + \sum_{k=1}^{n_B} -\nabla_{\vec{r}_{ik}^t} U(\|\vec{r}_{ik}^t\|)$;
    Update:
    $\vec{w}_i \leftarrow \vec{v}_i + \vec{F}_i dt$;
    $\vec{v}_i^{t+1} \leftarrow \vec{w}_i g(\frac{v_i^{max}}{\|\vec{w}_i\|})$;
    $\vec{r}_i^{t+1} \leftarrow \vec{r}_i^t + v_i^t dt$
  end
end

3. PROPOSED METHOD

Baseline Implementation. Owing to the fact that there is neither code provided by the author nor other resources from the Internet, we first need to implement the algorithm directly from the reference paper using C++. We divided the algorithm into 3 parts according to the type of the social forces and derived the formula to calculate the gradient with analysis. We referred to the implementation of a similar paper[7] to use a class to represent each pedestrian, made some assumptions about the desired velocity of each pedestrian, and defined a class similar to Vector2d in Eigen[8], in order to ease the implementation. It is then verified by observing the same phenomenon as in the paper, and through simple uni-tests designed ourselves. This version will be used as the baseline to evaluate our following optimizations. The pseudocode of the baseline implementation is described by Algorithm 1.

Optimization 1: Inlining, Pre-computation, Strength Reduction. Before starting to optimize the code, a detailed analysis of the baseline implementation is performed. We first profiled the code with the Intel VTune Profiler and concluded that `Repulsive_pedestrian()`, a function that computes the repulsive force between pedestrians is the bottleneck. Then a roofline analysis was made with Intel Advisor, leading to the conclusion that the program is computed bound if the dataset fits in the cache, or in Intel's words, is fundamentally compute bound but also affected by memory bandwidth, as shown in Figure 1.
The roofline analysis also shows the computation has only been vectorized for a small fraction of the code, despite `-ftree-vectorize` being used for that compilation, as Intel Advisor prompts the computation is bounded by the scalar roofline. Further, we observed our baseline implementation was not able to terminate with more than 2^{14} pedestrians. Each pedestrian struct is 80 bytes, thus the max test dataset size is $2^{14} * 80 = 1310720$ bytes $= 1311$ KB, which fills L1 cache and most of L2 cache, but far from filling LLC. This confirms the compute-bound nature of the program, and led us to begin the optimization procedure on the computations performed in `Repulsive_pedestrian()`.

We first inlined all the procedure calls in the algorithm, this gives us a $n_p \times n_p$, unit-stride double loop, where n_p is the number of pedestrian structs, the inner loop was originally the body of `Repulsive_pedestrian()`. We ignored the n_{iter} loop as it's not a part of the paper's algorithm, instead, it's used to generate enough points for visualization, thus we consider it out-of-scope for the purpose of optimizing the algorithm described in the paper. Then we noticed multiple divisions by constants and cosine computation on constants in the loop:

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

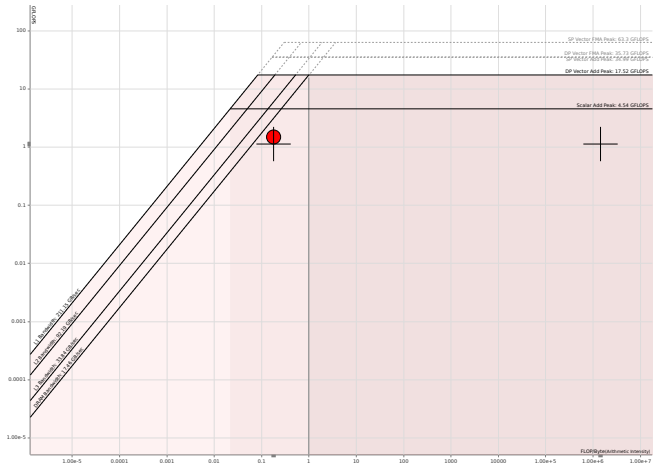

Fig. 1: Roofline Plot of Baseline Implementation

```
    ...
    Vector2d F0_alpha=(p->v0*p->e-p->v)/tau_a;
    ...
    for (int j=0; j<n_p; j++) {
        double b
            =sqrt((rab.norm()
                +rabvb.norm())
                *(rab.norm()+rabvb.norm())
                -vb.norm()*vb.norm())/2;
        f_ab
            =vab0/sigma
            *exp(-b/sigma)
            *(rab.norm()+rabvb.norm())
            *(rab/rab.norm()+rabvb/rabvb.norm())
            /(4*b);
        ...
        if (pi->e.dot(-f_ab)
            >=-f_ab.norm()*cos(phi))
            wef = 1.;
        ...
    }
    ...
}
```

Where `sigma, tau_a, phi` are constants. Since divisions are much more expensive than multiplications, we first computed the inverse of the constants outside the double loop, then replaced divisions inside the loop by multiplication to the inverses. The expensive cosine and some other constant computations are also lifted out of the loop:

```
double cos_phi = cos(100./180*M_PI);
double tau_a_inv = 1 / tau_a;
double sigma_inv = 1/ sigma;
double norm_sum = 0;
double _const = vab0*sigma_inv * 0.25;
double _const1 = UaB0/R;
for (int i=0; i<n_p; i++) {
    ...
}
```

Next, we observed the abundance of division by vector norm computations when computing `f_ab` and `b`. The norm of a `Vector2d` is defined as $\sqrt{x^2+y^2}$, combined with division, results in inverse square-root computations. Since square roots and divisions are both expensive operations, we would like to replace them with cheaper ones. Therefore we employed the famous Quake fast inverse square-root algorithm:

```
double rab_norm_inv=rab_norm_sqr;
double x2=rab_norm_inv*0.5;
std::int64_t x3=*(std::int64_t *)&rab_norm_inv;
x3 = 0x5fe6eb50c7b537a9-(x3>>1);
rab_norm_inv=*(double *)&x3;
rab_norm_inv=rab_norm_inv
            *(1.5-(x2*rab_norm_inv*rab_norm_inv));
```

Although the algorithm has more flops, individual operations are much cheaper. Finally, we unrolled the inner loop which computes the repulsive force between pedestrians 4-fold, in order to spot repeated memory accesses and potentially perform scalar replacement. While we did not find any, unrolling combined with all previous optimizations did help us achieve 1.5x performance improvement compared with the baseline implementation.

Optimization 2: Using Arrays. We then turned our attention to the pedestrian struct itself, as the roofline analysis indicates memory accesses can affect the computation. In our baseline implementation, the 80-byte pedestrian struct is of the format:

```
pedestrian {
    int stateCurrent, stateFuture, color;
    Vector2d r, v, e;
    double v0, vMax;
}

Vector2d {
    double x, y;
}
```

The n_p loop that computes the repulsive force between pedestrians operates on `r`,`v`,`e` fields of the pedestrian struct. The machine used to perform testing has a cache block size of 64 bytes. This struct clearly provides poor spatial locality, as each iteration a new pedestrian struct needs to be loaded into the cache to access the `r`,`v`,`e` fields, resulting $n_p \times n_p$ cache misses in the double loop of the algorithm. To improve spatial locality, we changed the pedestrian data representation to multiple n_p length arrays:

```
int n_p;
int* color;
Vector2d* r, * r_next, * v, * v_next, * e;
double* v0, * vMax;
```

This, in theory, should improve spatial locality, as for each pedestrian's `r`,`v`,`e` data are loaded into the cache, its three neighbouring pedestrian's `r`,`v`,`e` data are also loaded, which should reduce the number of cache misses by a factor of 4.

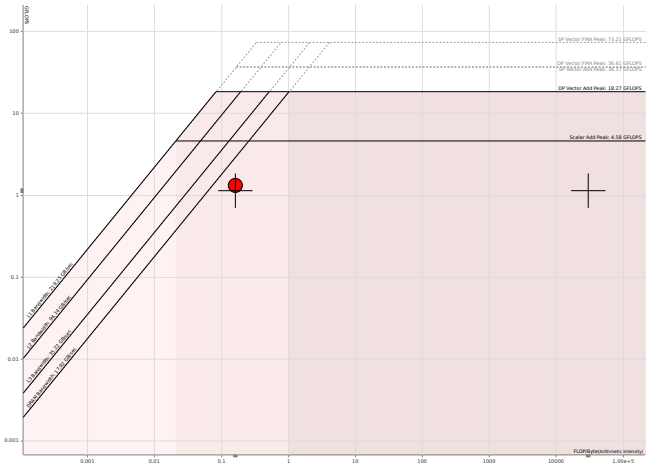

Fig. 2: Optimization 2 Roofline Plot

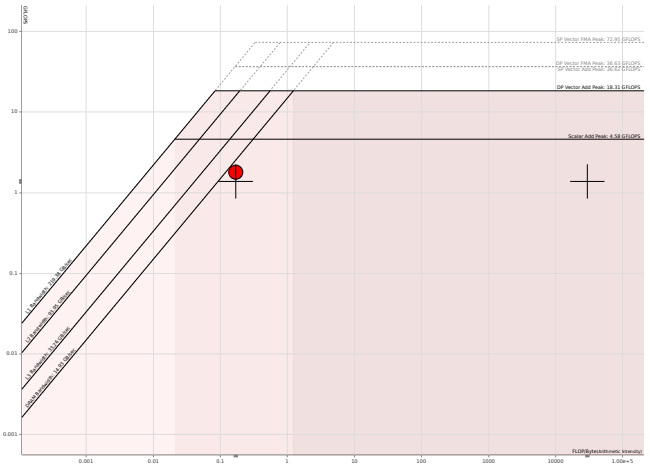

Fig. 3: Optimization 3 Roofline Plot

We then performed another set of analyses to check the effect of locality improvements. Since the number of flops stays the same but there are fewer memory transfers, we would expect a higher operational intensity. But the roofline plot appears otherwise, as shown in Figure 2. There seems to have no change in operational intensity and performance for this version of the code. To further investigate the locality issue, we used *Linux Perf tool* to check the number of cache misses between the baseline version and the arrays version, yet the result shows no improvements. We believe the reason for no locality improvement is our test dataset all fits in the cache (2^{14} pedestrians). Due to high cache bandwidth, locality does not affect the performance and speed of the computation as much as if the dataset resides in memory. One observation we made is this time, Intel Advisor suggests the computation is bounded by the vectorized roofline instead of the scalar roofline. This indicates changing to SoA enables the compiler to perform vectorization on the computation to some degree. Although using arrays itself did not bring any performance or speed improvements, it enables the compiler to do vectorization and certainly provides more room for further optimization.

Optimization 3: Combining 1 & 2, Blocking. We next combine the array data structure with Optimization 1. This time, the fast inverse square root algorithm starts to hinder the performance, as Intel Advisor prompts the computations are once again bounded by the scalar roofline. After removing the fast inverse square root algorithm, the roofline analysis results in much higher performance compared to the baseline implementation, and it is bounded by the vectorized roofline, confirming that SoA enables many compiler optimizations to take place, as shown in Figure 3.

This improvement allows us to push our dataset size to 2^{15} pedestrians. This input size fills L2 cache but far from filling the LLC, which motivates us to wonder if cache locality optimizations can further improve performance since now LLC bandwidth needs to be taken into account. Thus, we decide to consider blocking to improve temporal locality. Similar to the blocking in matrix-matrix multiplication, we unroll the outer loop as well, switching from calculating the repulsive effects between every 2 pedestrians to those between 2 blocks of pedestrians. Theoretically, it should yield better temporal locality, since it would lower the number of times needed to reload the data of each pedestrian. However, the run time is slower than the no-blocking version. We believe forcing blocking limits the power of the compiler, preventing it from reordering the whole computation in a more efficient way. Thus, we chose not to consider blocking in the following optimizations and continued with the version which only unrolls the innermost loop.

Optimization 4: AVX. Arrays provide opportunity to perform vectorization. Specifically, we use AVX256 intrinsics. Looking from the pre-results of the previous optimization methods, it seems that the compiler only performed a little part on the array vectorization, which means that there's still space for AVX256 intrinsics to do further vectorization. The previous inner loop unrolling $\times$ 4 enables one time vectorization for AVX256 intrinsics on a double array. Therefore, if we continue unrolling with AVX intrinsics, then the overall unrolling factor to the original algorithm would be 16 (4 $\times$ 4). There's three problems that we aim to tackle. First problem is that standard compiler did not support exponential calculations, therefore, we provide two ways to overcome this problem. One way is to perform element-wise exponential calculations before hand. Another way is to use Intel Compiler which provides the `_mm256_exp_pd` expression. Second problem is that if we do not unroll the outer loop, there're still parts that didn't use AVX expressions but we would like to create a pure-AVX-version of the algorithm, therefore we decided to un-

version	10	50	100
Optimization 1	0.0	0.0	0.0
Optimization 2	0.0	0.0	0.0
Optimization 3	0.0	0.0	≈ 0.04
Optimization 4	0.0	0.0	≈ 0.04

Table 1: Errors on different iterations

roll the outer-loop as well, the similar way as the blocking operations but the outer loop expressions would be all written in AVX intrinsics. The third problem is that we found that the velocity terms could be pre-computed. These could be moved out from the inner loop to a single loop with $\mathcal{O}(n)$ that performs the computation with AVX expressions.

Additionally, we removed also all Vector2D variables that are included into vectorization calculations. We only keep the force terms to be Vector2D variables. Reason is that 4 individual for-loops which means that computing force terms with AVX in parallel is not possible.

4. EXPERIMENTAL RESULTS

Unit Testing.: To test the correctness of our baseline, we did some computations by hand and made some test cases to verify whether the code produces the same results under a tolerance of 10^{-3} units. Specifically, we made some tests for the relevant functions such as pedestrian movement, repulsive force w.r.t. the buildings, and between the pedestrians.

Furthermore, the correctness of the baseline version serves as a point of reference for our optimizations. We run both baseline and optimization with the same input to testify the correctness of the latter, by taking the Euclidean distance between the pedestrians. The optimized version is regarded to be correct if the distance is within the bound of 10^{-3} units.

`-ffast-math`: the flag allows us to have faster code (from 1.5 to 2.3 times faster, considering an input of 100 pedestrians over 200 iterations), but sacrifices the accuracy by deviating about 10% from the baseline results.

#iterations: by increasing the number of iterations, the relative error of each round would accumulate. Considering 100 pedestrians and performing 10, 50, and 100 iterations, the relative errors w.r.t. the baseline is shown in Table 1. Note that the measurements are done without using `-ffast-math` to avoid extra errors.

Experimental Setup.

All experiments were performed on Intel i7-12700H @ 2.3 GHz with turbo boost disabled. Cache sizes are 80KB L1, 1280KB L2, and 24MB LLC. G++ 12.2 is used for all compilations except for AVX optimizations, in which ICPC 2021.9 is used. We tested different flag combinations as shown in Figure 4. None of the flag combinations provide much performance gain compared to another, but we proceeded with `-O3 -ffast-math -march=native -mavx2 -ftree-vectorize -m64` for all compilations as it yields the most stable result during testing.

Input sizes from $n_p = 2^3$ to $n_p = 2^{14}$, log scale, are tested for the Baseline, Optimization 1 and Optimization 2 codes, while $n_p = 2^3$ to $n_p = 2^{15}$ is tested for Optimization 3 onwards. The range is chosen because the program will not terminate for larger input sizes, and we assumed w.l.o.g, the input size is divisible by 4 to simplify AVX implementations. n_{iter} is fixed to 1 for all performance/runtime tests performed.

Performance. Figure 5 shows performance over different code versions. Input sizes of $n_p = 2^3$ to $n_p = 2^{15}$ are tested, which means all test dataset fits in LLC. The baseline implementation is only able to achieve less than 0.5 flops/-cycle. Changing the pedestrian data structure to arrays as in Optimization 2 has no impact on performance. Performing pre-computation, unrolling the inner loop that computes the repulsive force between pedestrians 4-fold, and employing the fast inverse square root algorithm as in Optimization 2 allows for a 1.5x performance gain compared to the baseline. By combining Optimization 1 and 2 as in Optimization 3, the compiler is able to perform vectorization with the optimization flags enabled, this leads to a significant 2.4x performance gain and allows us to test the input size of $n_p = 2^{15}$. Finally, by using AVX intrinsics as in Optimization 4, we are able to get a slightly more performance gain at 2.5x compared to the baseline.

Given our previous analysis (Eq. 14), the total flop count $W(n) = 48n^2$ flops. Since we used SVML for exponential function, which is closed source and for which Intel did not provide much information, we measured the gap of $exp()$ ourselves and assume it can be fully vectorized. Considering only instruction mix and ignoring dependencies between instructions, a lower bound for the runtime should be $T(n) \geq 30.25n^2$ cycles, and an associated upper bound for performance is $P(n) \leq 1.587$ flops/cycles. In this case, Optimization 4 achieves 72% of the peak performance. It is moderately close to the theoretical peak performance and the gap can be explained by the dependencies between instructions and data transfers within the cache. We believe it is unlikely for the code to achieve more percentage of the peak performance as we must take dependencies and cache transfers into account when measuring the actual performance, and these factors are unavoidable.

Speedup. Figure 6 shows speedup over different code versions. The execution time (in cycles) of the baseline implementation is set to 1, and the execution time of all other versions are normalized to the baseline execution time. Input sizes of $n_p = 2^3$ to $n_p = 2^{14}$ are tested, which means

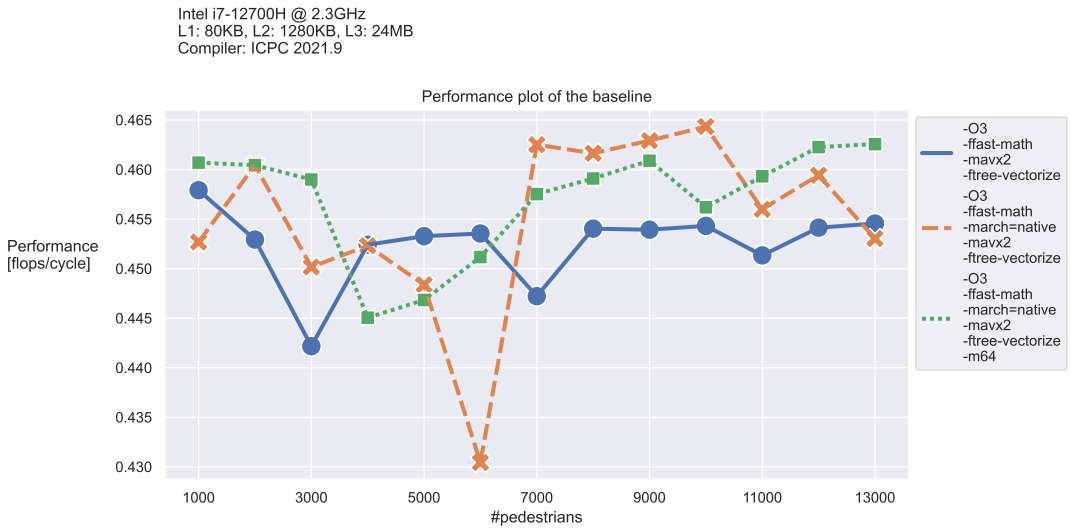

Fig. 4: Baseline Performance Under Different Compiler Flags

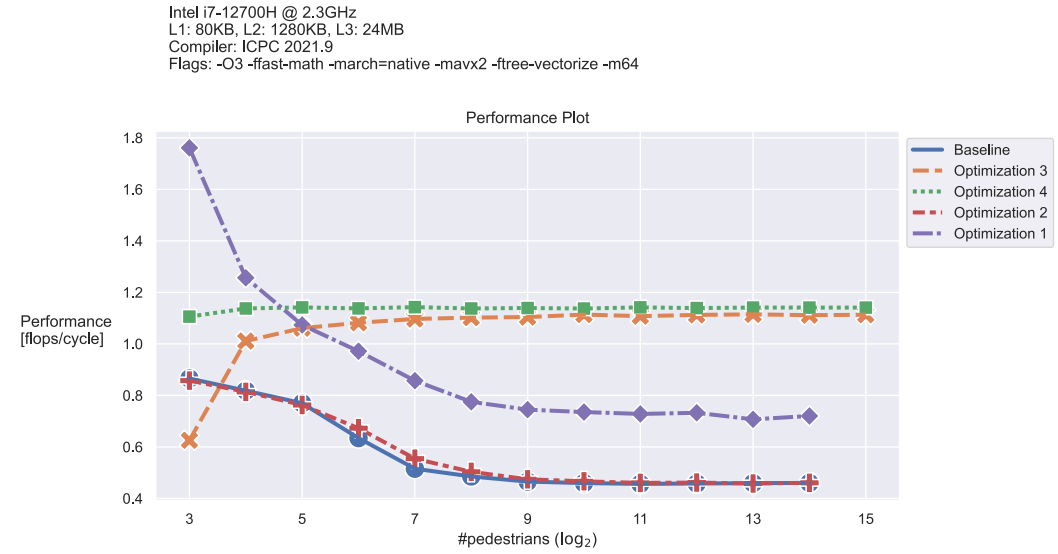

Fig. 5: Performance Plot over Different Code Versions

all test dataset fits in LLC. Changing the pedestrian data structure to arrays as in Optimization 2 has no effect on the execution time. By performing pre-computation, unrolling and using the fast inverse square root algorithm as in Optimization 1 we are able to get around 1.6x speedup compared to the baseline. By combining Optimization 1 and Optimization 2, as Optimization 3, which allows the compiler to perform vectorization in addition to the manual optimization we performed, we are able to get around 4x speed up for large input sizes compared to the baseline. Finally, by using AVX intrinsics as in Optimization 4, we are able to get slightly better than 4x speedup for larger input sizes but the difference between Optimization 3 and 4 is minimal. We believe the reason we are unable to get more speedup by employing AVX intrinsics is the compiler may have compiled the code with more sophisticated vector instructions, but by manually using intrinsics we force the compiler to compile the code to the instruction mix dictated by the AVX intrinsics, which is not noticeably better than the compiler's optimized instruction mix in this case. Since the algorithm itself contains many unavoidable expensive computations such as division, exponential and square root,

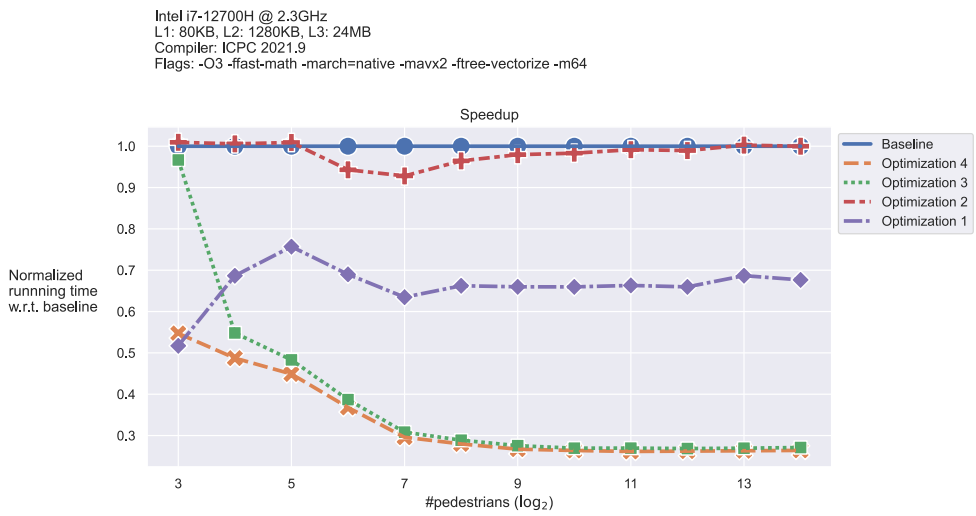

Fig. 6: Speedup over Different Code Versions

and vectorization only allow parallelizing computation but does not reduce the cost of the computation itself, we believe it is unlikely for the code to be sped up further.

5. CONCLUSIONS

We optimize the computation of SFM in four different ways, first using standard means such as unrolling, scalar replacements, changing the struct etc. Then, we implemented the vectorization using AVX intrinsics, to speed up the computations. Each optimization is built upon the previous one, and they provide strong evidence that the compiler-level and hardware-specific optimizations can improve the performance of our code.

Due to the high quality of our baseline and the dependencies of different computations, we believe that there is no other way to further improve the computation with existing tools, as can be seen from the roofline plot.

In conclusion, with our last optimization, we are able to speed up the computation by a factor of 4 w.r.t. the baseline, which also allows us to push the input size from 2^{14} pedestrians to 2^{15}. The final version of optimization has reached approximately 72 % of the maximum theoretical performance.

6. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

Jiajun Jiang. Worked with Peiyuan on the baseline implementation and memory optimization (blocking, modifying data structure) as in Optimization 2. Debugging. Worked on plotting the performance plots. Helped Jiaqing on attempting to reimplement the algorithm in pure C instead of C++. Analyzed reasons for the performance results and unexpected phenomena during testing.

Kehong Liu. Performed roofline analysis throughout the project. Worked with Jiaqing on compute/algorithmic optimizations as Optimization 3, and AVX intrinsics as in Optimization 4. Worked on collecting performance data and performance analysis. Performed cost analysis of the algorithm. Analyzed reasons for the performance results and phenomena during testing.

Jiaqing Xie. Focused on compute/algorithmic optimizations as in Optimization 1. Worked on Optimization 3 and AVX optimizations as in Optimization 4 with Kehong. Experimented on reimplementing the algorithm in pure C and analyzed potential issues and tradeoffs before deciding to discard it. Performed cost analysis of the algorithm. Lead discussion on the next steps of optimization based on analysis results.

Peiyuan Xie. Worked with Jiajun on the baseline implementation and memory optimization such as blocking as in Optimization 2. Analyzed potential memory locality issues with different data and loop structures. Debugging. Performed cost analysis of the algorithm. Implemented the visualization script. Lead discussion on directions for optimization.

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] Dirk Helbing, "Physikalische modellierung des dynamischen verhaltens von fußgängern (physical model-

ing of the dynamic behavior of pedestrians)," *Available at SSRN 2413177*, 1990.

[3] Kurt Lewin, "Field theory in social science: selected theoretical papers (edited by dorwin cartwright.)," 1951.

[4] Dirk Helbing, "Improved fluid-dynamic model for vehicular traffic," *Physical Review E*, vol. 51, no. 4, pp. 3164, 1995.

[5] Dirk Helbing, Illés Farkas, and Tamas Vicsek, "Simulating dynamical features of escape panic," *Nature*, vol. 407, no. 6803, pp. 487–490, 2000.

[6] Sven Kreiss, "Deep social force," *arXiv preprint arXiv:2109.12081*, 2021.

[7] Dirk Helbing, Illés Farkas, and Tamás Vicsek, "Simulating dynamical features of escape panic," *Nature*, vol. 407, no. 6803, pp. 487–490, Sep 2000.

[8] Gaël Guennebaud, Benoît Jacob, et al., "Eigen v3," http://eigen.tuxfamily.org, 2010.