FAST IMPLEMENTATION OF SOCIAL FORCE MODEL FOR PEDESTRIAN DYNAMICS

Juan Diaz Sada, Thierry Backes, Lam Nguyen Thiet

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

This reports presents a fast implementation of a simulation of the Social Force Model algorithm. We present our analysis of performance bottlenecks in the baseline simulation and our various successful and failed optimizations attempts.

1. INTRODUCTION

Modeling pedestrian behaviour in crowds and buildings can provide valuable insights and help with the design of buildings, public spaces, and even festival grounds. One way of modeling crowd movement is trough a microscopic simulation where each agent is modeled individually. While this simulation might provide accurate results, it is computationally expensive. In addition, the timesteps in the simulation have to be computed in order and can't be trivially parallelised. We present and analyze multiple single-core optimizations that together improve the run time of a particular simulation.

2. BACKGROUND: SOCIAL FORCE MODEL

In this section we introduce the Social Force Model [1], specify our implementation, and perform a bottleneck and cost analysis.

Social Force Model. The model attempts to describe the motion of pedestrians as if they are subject to multiple forces, the so called *social forces*. The motion of a pedestrian is fully described by four of them:

Desired direction force Attraction to the desired location

Agent repulsive force Repulsion to other agents

Border repulsive force Repulsion from physical borders

Attractive force Attraction to other agents

For a time step t, the total force $\vec{F}_\alpha$ that acts on an agent α is the sum of the four individual forces (same order as above):

$$\vec{F}_\alpha(t) = \vec{F}_\alpha^0 + \sum_\beta \vec{F}_{\alpha,\beta} + \sum_B \vec{F}_{\alpha,B} + \sum_i \vec{F}_{\alpha,i}$$

where β sums over the other agents, B over all the borders, and i over all the points of interest. To decrease complexity, we did not implement the attractive force. For brevity we do not explain the forces in detail and refer the original paper [1]. The simulation exhibits the following pattern:

Algorithm 1: Structure of the simulation

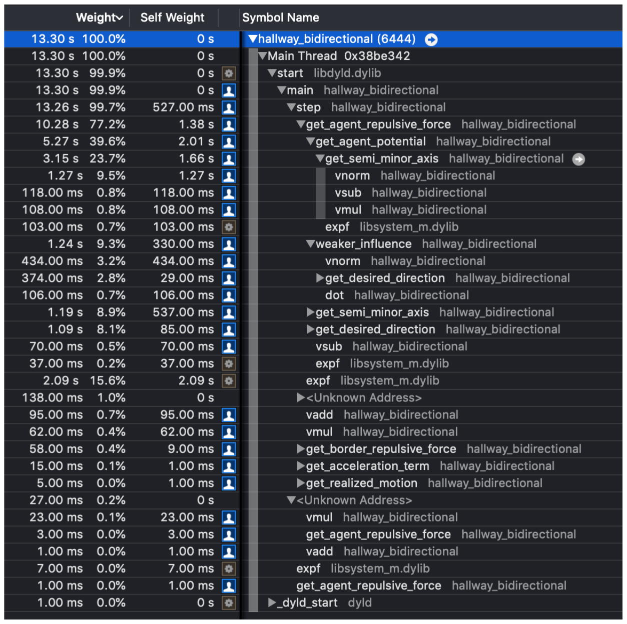
```
foreach timestep $t$ do
    foreach agent $\alpha$ do
        foreach agent $\beta$ do
            compute $\vec{F}_{\alpha,\beta}$
        foreach border $B$ do
            compute $\vec{F}_{\alpha,B}$
        compute $\vec{F}_\alpha(t)$
        $velocity\_buffer \leftarrow velocity\_new$
        $position\_buffer \leftarrow position\_new$
    foreach agent $\alpha$ do
        $velocity \leftarrow velocity\_buffer$
        $position \leftarrow position\_buffer$
```

The new state for each agent is stored in a buffer and updated when all the agents finished computing their new state for a given timestep. Due to this, the order of agents for a specific timestep is irrelevant.

Implementation. The baseline implementation is a straight forward implementation of the algorithm in C. We could not find and open source project in C that implements exactly and only this paper. No consideration of speed was taken, and the focus was on correctness. All of the constants are identical to the original paper except for the $\Delta t = 2$ from equation (4) in the original paper. We set this equal to $\Delta t = dt = 0.2$, the time that we advance in each timestep. To check for correctness, we compared our implementation to a python implementation from `svenkreiss` from Github [2].

Our implementation has two build methods. In a normal build, we store the position of each agent for each time step and write them at the end of the simulation to file. The output can then be visualized. This is in contrast to the bench marking, where we do not store the old positions for each agent and do not have any I/O. We do not care about any historical state, and the state of the agent only consists of its parameters and current position. This also means that we do not load and parse different scenes from text files, but have a header file for each scene and compile a specific binary for each scene. Besides those two changes, the compilation for normal and bench marking builds is identical (except for possible additional compiler flags).

Bottleneck analysis. The first step of the baseline analysis is done with a profiler. As expected, the bottlenecks are identical on the different machines and we only report data from one specific machine in figure 1.



```
     Weight     Self Weight   Symbol Name
 13.30 s 100.0%       0 s     ▼hallway_bidirectional (6444)
 13.30 s 100.0%       0 s      ▼Main Thread  0x38be342
 13.30 s  99.9%       0 s        ▼start  libdyld.dylib
 13.30 s  99.9%       0 s          ▼main  hallway_bidirectional
 13.26 s  99.7%  527.00 ms          ▼step  hallway_bidirectional
 10.28 s  77.2%    1.38 s             ▼get_agent_repulsive_force  hallway_bidirectional
  5.27 s  39.6%    2.01 s               ▼get_agent_potential  hallway_bidirectional
  3.15 s  23.7%    1.66 s                 ▼get_semi_minor_axis  hallway_bidirectional
  1.27 s   9.5%    1.27 s                    vnorm  hallway_bidirectional
118.00 ms  0.8%  118.00 ms                   vsub  hallway_bidirectional
108.00 ms  0.8%  108.00 ms                   vmul  hallway_bidirectional
103.00 ms  0.7%  103.00 ms                 expf  libsystem_m.dylib
  1.24 s   9.3%  330.00 ms              ▼weaker_influence  hallway_bidirectional
434.00 ms  3.2%  434.00 ms                 vnorm  hallway_bidirectional
374.00 ms  2.8%   29.00 ms                ▶get_desired_direction  hallway_bidirectional
106.00 ms  0.7%  106.00 ms                 dot  hallway_bidirectional
  1.19 s   8.9%  537.00 ms              ▶get_semi_minor_axis  hallway_bidirectional
  1.09 s   8.1%   85.00 ms              ▶get_desired_direction  hallway_bidirectional
 70.00 ms  0.5%   70.00 ms               vsub  hallway_bidirectional
 37.00 ms  0.2%   37.00 ms               expf  libsystem_m.dylib
  2.09 s  15.6%    2.09 s              expf  libsystem_m.dylib
138.00 ms  1.0%       0 s             ▶<Unknown Address>
 95.00 ms  0.7%   95.00 ms             vadd  hallway_bidirectional
 62.00 ms  0.4%   62.00 ms             vmul  hallway_bidirectional
 58.00 ms  0.4%    9.00 ms            ▶get_border_repulsive_force  hallway_bidirectional
 15.00 ms  0.1%    1.00 ms            ▶get_acceleration_term  hallway_bidirectional
  5.00 ms  0.0%    1.00 ms            ▶get_realized_motion  hallway_bidirectional
 27.00 ms  0.2%       0 s           ▼<Unknown Address>
 23.00 ms  0.1%   23.00 ms             vmul  hallway_bidirectional
  3.00 ms  0.0%    3.00 ms             get_agent_repulsive_force  hallway_bidirectional
  1.00 ms  0.0%    1.00 ms             vadd  hallway_bidirectional
  7.00 ms  0.0%    7.00 ms           expf  libsystem_m.dylib
  1.00 ms  0.0%    1.00 ms           get_agent_repulsive_force  hallway_bidirectional
  1.00 ms  0.0%       0 s        ▶_dyld_start  dyld
```

Fig. 1. Time profiler output on the baseline implementation of `hallway_bidirectional` on the `MB2` machine (more details in chapter 4) with `-O3 -g -march=native` and its default compiler.

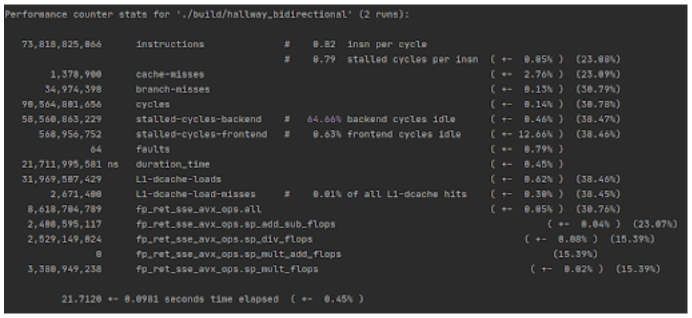
We can see that the majority of the time is is spent in the `get_agent_repulsive_force` function. This also reflects the scene setup as described in detail in chapter 4. In addition to this, we spend a lot of time in our custom math functions, like `vadd`, or `vnorm`. The functions are simply a wrapper around standard math operations, to work with our `vec2` struct:

```
typedef struct{
    float x;
    float y;
} vec2;
```

Another big bottleneck are the `libm` math functions, in particular `expf` and `cos`.

In order to figure out in greater detail, how the program behaves and where potential bottlenecks are, we used the linux `perf` tool with different event counters as we can see in 2. The same scene as above was run with

```
perf stat -e instructions,cache-misses,
branch-misses,cycles,
stalled-cycles-backend,
stalled-cycles-frontend,
faults,duration_time,
L1-dcache-loads,
L1-dcache-load-misses,
fp_ret_sse_avx_ops.all,
fp_ret_sse_avx_ops.sp_add_sub_flops,
fp_ret_sse_avx_ops.sp_div_flops,
fp_ret_sse_avx_ops.sp_mult_add_flops,
fp_ret_sse_avx_ops.sp_mult_flops
```



```
Performance counter stats for './build/hallway_bidirectional' (2 runs):

  73,818,825,866      instructions             #    0.82  insn per cycle
                                               #    0.79  stalled cycles per insn  ( +-  0.05% )  (23.08%)
       1,370,900      cache-misses                                                ( +-  2.76% )  (23.09%)
      34,974,398      branch-misses                                               ( +-  0.13% )  (30.79%)
  90,564,801,656      cycles                                                      ( +-  0.14% )  (30.78%)
  58,560,863,229      stalled-cycles-backend   #   64.66% backend cycles idle     ( +-  0.46% )  (38.47%)
     568,956,752      stalled-cycles-frontend  #    0.63% frontend cycles idle    ( +- 12.66% )  (38.46%)
              64      faults                                                      ( +-  0.79% )
  21,711,995,581 ns   duration_time                                               ( +-  0.45% )
  31,969,587,429      L1-dcache-loads                                             ( +-  0.62% )  (38.46%)
       2,671,400      L1-dcache-load-misses    #    0.01% of all L1-dcache hits   ( +-  0.38% )  (38.45%)
   8,618,704,789      fp_ret_sse_avx_ops.all                                      ( +-  0.05% )  (30.76%)
   2,400,595,117      fp_ret_sse_avx_ops.sp_add_sub_flops                                   ( +-  0.04% )  (23.07%)
   2,529,149,024      fp_ret_sse_avx_ops.sp_div_flops                               ( +-  0.08% )  (15.39%)
               0      fp_ret_sse_avx_ops.sp_mult_add_flops                                 (15.39%)
   3,380,949,238      fp_ret_sse_avx_ops.sp_mult_flops                               ( +-  0.02% )  (15.39%)

        21.7120 +- 0.0981 seconds time elapsed  ( +-  0.45% )
```

Fig. 2. `perf` output on the baseline code with scene `hallway_bidirectional`. Compiled with `-O3 -g -march=native` on the `AMD` machine.

We could not find much documentation about the different counters. Using strace, we found that `cache-misses` is a syscall to `perf_event_open` with the config of `PERF_COUNT_HW_CACHE_MISSES`, which measures last level cache accesses. Note that this might also include *prefetches and coherency messages*. Compared to all the L1 data cache loads, there are barely any cache misses for this particular scene (it also holds true for other scenes).

Another thing we notice is that the are more than twice as many total instructions than floating point operations. This means that we do a lot of work that doesn't look *useful*. Lastly, there are a lot of stalled backend cycles. There are many elements that hide behind the term *backend*. Generally, it means that the back end, which is the actual floating point computation engine, is either waiting for new instructions, new data, or waiting for instructions to retire. As we do not have cache issues, we assume that the backend either waits for computations to finish, or that there are too many dependencies such that we can't dispatch instructions to the floating point unit fast enough.

To conclude, the main bottlenecks are: Time spent in agent-to-agent computation, custom math functions, libm functions, massive instruction overhead.

Cost Analysis. For a cost analysis, we need the number of floating point operations, and the amount of memory transferred between the main memory and the last layer cache.

To find out the number of floating point operations of the baseline, we used three different methods. One method replaces all the mathematical operations with a function call, and each time the function is called we increment a counter. To be more precise, we replace `c = a+b` with `c = add(a,b)`, and the function `float add(float, float);` does the addition operation and increments a counter. For the second method we created a formula that can calculate the flops based on the input scene. The formula is:

$$((14a) + (105a(a-1) + (47b \cdot a) + (16a)) \cdot n$$

where a is the number of agents, b is the number of borders, and n is the number of time steps that we perform. The third option is using `perf`, the linux performance monitoring tool and measuring the event counter `fp_ret_sse_avx_ops.all`.

The formula for the `hallway_bidirectional` scene estimates $12,603,002,800$ operations. Incrementing counters produces $12,607,974,130$ operations on machine `MB1`. The issue comes with instrumentation. Different compilers and different flags give drastically different flop counts. Table 1 shows the different results

compiler	floating point operations
gcc -O0	$14,019,650,583$
clang -O0	$14,014,810,792$
gcc -O3	$8,617,606,326$
clang -O3	$18,011,768,928$

Table 1. Different compiler and flags produce different floating point operations according to `perf` for the same scene on machine `AMD`.

We encounter a similar issue for the memory transfer. Two different methods were used to measure the cache accesses and memory transfer, namely `perf` and `cachegrind`. While `perf` uses event counters during the runtime, `cachegrind` simulates the cache based on the machine architecture. From `perf` we get $1,378,900$ L3 cache misses and $2,671,400$ L1 data cache misses and a total of $31,969,587,429$ L1 data cache loads. `cachegrind` on the other hand provides the output that is summarised in table 2

type	value
Memory reads (Dr)	$86,219,028,170$
L1 data read misses (D1mr)	$812,839,371$
L3 data read misses (DLmr)	$2,125$
Memory writes (Dw)	$35,830,012,825$
L1 data write misses (D1mw)	$678,456$
L3 data write misses (DLmw)	577

Table 2. Cachegrind output for machine `AMD` on the baseline code and the scene `hallway_bidirectional`.

It is not clear how to interpret the vastly different results of `perf` and `cachegrind`. The only result that looks similar are the L1 data cache loads from `perf` and the Memory writes (Dw) from `cachegrind`. The scene has 200 agents 2 borders and 3000 steps. Each agent stores 13 floats (in the benchmarking version, we assume no padding in the struct layout) and each border is 4 floats, so a total of 2608 floats, or 10432 bytes. As the machine `AMD` has a last layer cache of 33554432 bytes (according to `cachegrind`), we assume that the Dr and Dw values from `cachegrind` do not represent actual main memory accesses but represent main memory accesses *if* there is no cache.

The operational intensity $I(n)$ is defined as $I(n) = \frac{W(n)}{Q(n)}$ where $W(n)$ is the number of floating point instructions, and $Q(n)$ is the number of bytes transferred between the main memory and cache. We take the number of floating point operations from the formula, which is $12,603,002,800$ as it is close to the `gcc -O0/clang -O0` as well as the counted value. For the memory transfer, we look at the last layer cache misses as they are a proxy for memory transfer. The Zen architecture has an $L3$ cache line of 64 bytes [3]. Using the `cachegrind` L3 misses in bytes: $(2125 + 577) \cdot 64 = 153728$ then we get an operational intensity of $\frac{12603002800}{153728} \approx 81982$. With the L3 cache misses of `perf` we get a memory transfer of $1378900 \cdot 64 = 88249600$ bytes and an operational intensity of $\frac{12603002800}{88249600} \approx 142$. This is a substantial difference, but in both cases the program is compute bound rather than memory bound. Especially, if we consider the fact that `perf` reported about 0.01% of all L1 cache accesses to be misses.

3. OPTIMIZATIONS

In this section we explain the different optimizations that we did on top of the baseline implementation. We start with some basic C optimizations, continue with inlining, then go over the cache layout, proceed with loop unrolling, and end with vectorisation. All the code is run with `-O3 -march=native`, if not stated otherwise.

General C optimizations. The general C optimizaitons include scalar replacement, code motion, and precomputation. They do not really touch any of the bottlenecks, except that we precompute a cos value, rather than compute it in the function call. Additionally, we removed some pointers from our baseline code. From figure 3 we can see that there is barely any improvement in run time for the selected scenes on each machine compared to the baseline implementation.On average this optimization achieved only 1.02x speed up.

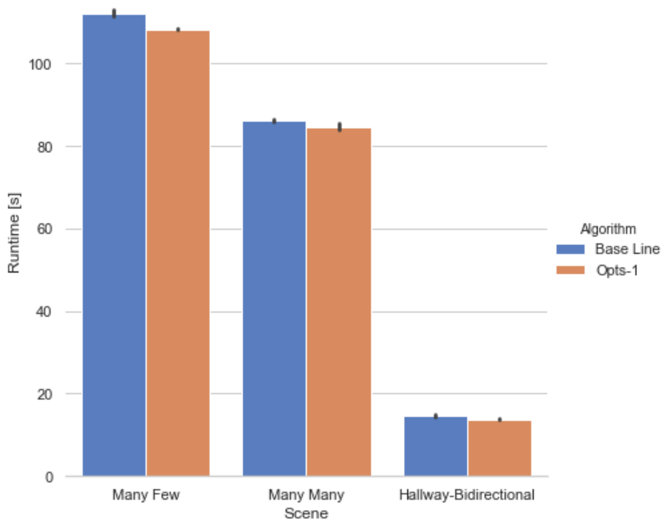


Fig. 3. Run-time bar plot for general C Optimizations, run on `MB1`.

Code inlining. During the baseline analysis we found that the baseline code spends a lot of time in our custom math functions, and also has many instructions that are not floating point computations. Our custom math functions are just a wrapper to compute easier with vectors, this means that `vadd` simply performs two additions and returns a new struct. There is however large overhead generated in the assembly for each function call of those vector functions as we can see in figure 4

```
push    %rbp
push    %rbx
sub     $0x98,%rsp
vmovss  0x108(%rsp),%xmm3
vmovq   0xe8(%rsp),%xmm1
vmovq   0xb0(%rsp),%xmm0CH
vmovss  %xmm3,0x8(%rsp)
callq   0x5555555560a0 <vsub>
```

Fig. 4. Overhead of function call which does two subtractions. Those functions are not inlined by the compiler.

The idea is that function inlining drastically reduces the overhead of a function call and therefore reduces the amount of total instructions. Doing this also might help the compiler to somehow optimize the mathematical operations rather than treating the function as a black box.

In general, the compiler inlines code. The baseline however had multiple functional units, which prohibited code inlining. We used the compiler flag `-lto` which enables link time optimizations to test our claim that inlining results in a speedup. One of those optimizations is code inlining across multiple functional units. This indeed produced a speedup, and upon inspection of the generated assembly, the functions were properly inlined. It is however not clear if the resulting speedup is due to inlining itself, or any other optimization that the flag activated.

To get a clearer picture about inlining, we therefore removed the flag and forced the compiler to inline the. To verify that the compiler respected the `inline` keyword, we used the flag `-Winline`. Inlining achieved on average a 1.8x speed up as we can see on 5.

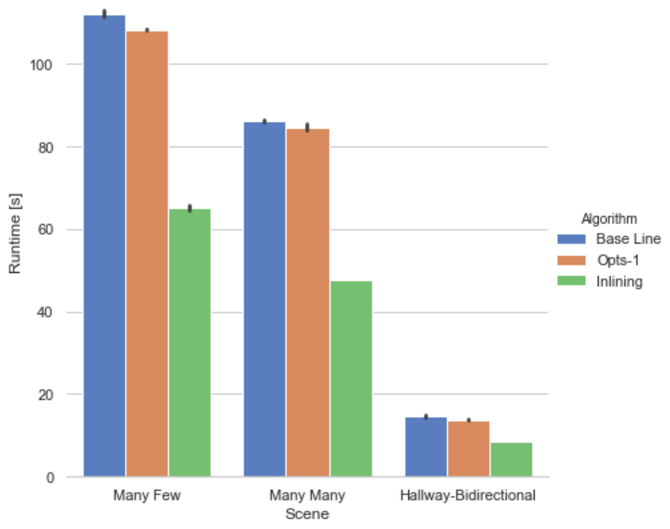


Fig. 5. Run-time bar plots after code inlining on `MB1`.

Using `perf` to figure out how inlining helped, we can see in figure 6 that the number of instructions is nearly halved, but also the number of floating point operations is less. It is not clear to us why there are less floating point operations.

C code unrolling. In this section we present our results with unrolling the innermost loop of the computation. The unrolling was done on top of all the general C optimizations.

```
39,027,208,668   instructions:u
6,957,332,218    fp_ret_sse_avx_ops.all:u
```

Fig. 6. Two perf counters after code inlining for the scene `hallway_bidirectional`.

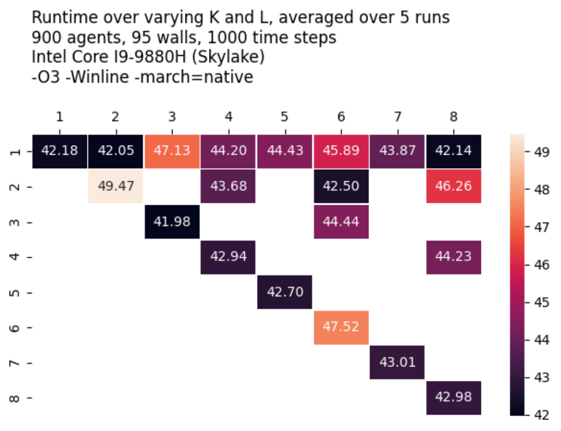


Fig. 7. Runtime plot for unrolling, ran on `MB3`. The runtimes were fetched from the Unix command `time` using the `user` value.

There are many reasons why one would apply unrolling for the Social Force Model algorithm :

- The computation of $\sum_\beta \vec{F}_{\alpha,\beta}$ and $\sum_B \vec{F}_{\alpha,B}$ for any two given α do not depend on each other for a single time-step t
- There are expensives operation with high latency that can be leveraged for ILP, such as `div` or `exp`.
- The program is compute bound.

We created a code generator in Python that generates alternatives and we would manually pick the best parameters **K** (numbers of accumulators) and **L** (unrolling factor). We didn't do scalar replacement, but we presume it wouldn't have a great influence as most of the time is spent on computation.

As seen in Figure 7 we were not able to get significant results. We have the best average runtime for **K** = 2 and **L** = 2 but the rest of the runtime is inconsistant.

We had a few leads for explaining this :

- The very low throughput of the expensive instructions forces **K** and **L** to be around 1.
- The compiler already does the optimization.
- Bugs...

Struct permutations for improved cache layout. Although we have strong belief that the program is compute, and not memory bound, it might still be favorable to have a better cache layout such that transfer between the different cache layers is reduced. It turns out that this is very difficult to measure, and we can only measure the L1 hits/misses, L3 hits/misses and runtime. We mainly focus on runtime, as a surrogate measure for the cache layout.

The struct of an agent holds its full state for a given timestep. During the computations, some values are generally used together such as the current position and the desired destination. Whereas other values are used only once or twice, such as the buffers for the position and velocity. The idea is that we want to fit all the values that we need together into the same cache line. As the memory of a struct is laid out linearly (with potential padding) in memory, reordering the struct will therefore also change how they are laid out in memory and fetched by the cache.

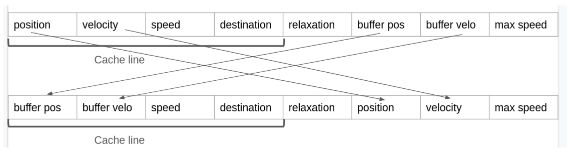


Fig. 8. Visualisation of a different cache layout and how this is mapped to a cache line.

It is not clear without a detailed computation graph to know when which values are accessed together, therefore we created a code generator that generated every possible permutation of the 7 struct elements and measured the runtime on the (smallest and) fastest scene. Those are in total 5040 runs which take a considerable amount of time, therefore only two measurements were done.

In figure 9 we sorted the two runs based on the runtime and plotted them together. We can see that there is a potential significant difference between the fastest and the slowest struct permutations. To back up this claim, we compared the permutations that led to the best results for both runs, and if the memory layout is the reason for the speedup, then we should see similar struct layouts for both runs. However, if the difference is due to the normal variance of measuring performance code (such as cold cache, threads moving to another core, interrupts, background tasks,...) and the struct layout doesn't have any influence, then the layouts should be randomly different.

We aggregated the layouts of the 40 best runs for both measurements and counted the occurrences of each variable for the different fields. As it turns out, there aren't many similarities. Simply comparing this by eye however is not a valid measure. In order to make sure that we average out the overhead of launching a process and having a cold cache, we did new measurements on a larger scene. Instead of running all the permutations, we are running the 150 fastest, and 150 slowest struct layouts that we got from the first run on a larger scene. The assumption is that an efficient struct layout is an inherent problem of the computation and not of the scene.

Figure 10 compares the runtime for both struct layouts.

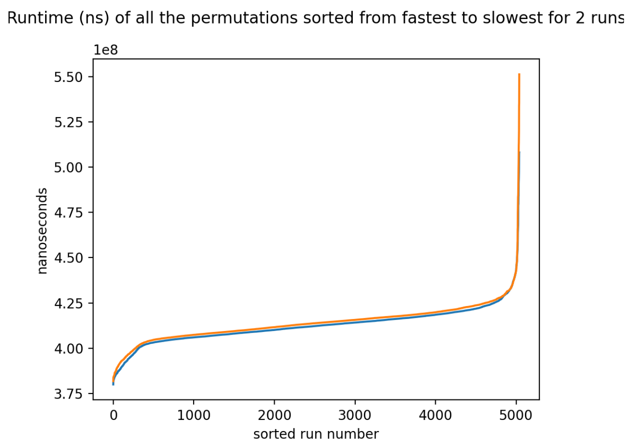


Fig. 9. Runtime of each permutation sorted and plotted, for two repetitions.

It turns out that there is no significant difference between the two

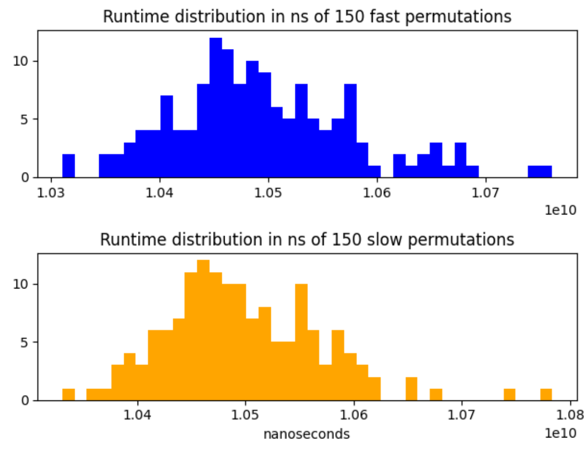


Fig. 10. Runtime distribution of the fastest(top) and slowest (bottom) permutations on the scene ...

Different memory layout: Struct of Arrays. In order to manually vectorise our code we had to re-write the way the agents state is stored. Previously, we used an array of structs, which means that each agent is laid out linearly after the other, and for a given data, it's values are laid out continuously in memory. The AVX intrinsics to load data, such as `load_ps` however require the data to be laid out continuously in memory. Using this function to load data means that we fill a vector of different attributes for the same agent, rather than filling a vector with the *same* attribute but for *different* agents. Instead of using `load_ps` we could use `gather` functions which can load values from arbitrary memory locations. Without measuring it, we are the opinion that this is a poor choice and decided to change the memory layout rather than working around it. This issue is visualized in figure 11.

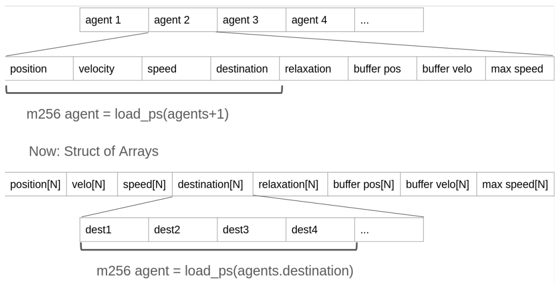


Fig. 11. Visualization of `load_ps` on two different agent data layouts

The new memory layout now is a struct of arrays, and the index in each array represents the data for a specific agents. Before we do manual vectorization however, we wanted to know if this new layout could improve the runtime. On the one hand, a completely different memory layout could improve the cache layout, but on the other hand, this layout could be favorable for the compiler to automatically vectorize our loops. Therefore, we measured the performance for the new layout and also tried loop unrolling for this layout. The results show us that on average the new struct of arrays layout achieves a 1.18x speed up compared to the array of struct layout.

AVX intrinsics. Changing the program that is written in the struct of array representation to use avx256 is straightforward. We can unroll most loops to work on 8 values in parallel. Our goal was to ensure the computation of the repulsive agent force was vectorized since it is the main bottleneck in the code and independent within each time step. We vectorized the entire code base. The biggest hurdles were missing math functions, and debugging. There is no equivalent for the exponential function in the intrinsics header file. We decided to use a header file from AVX mathfun [4] which implements common math functions such as cosine, and expo in pure AVX. The average speed up with AVX is 11.42x as you can see in figure 12.

4. EXPERIMENTAL SETUP

In this section we go over the experimental setup.

Experimental setup. We used several machines and tried out different compilers (and versions) to compile the scenes. In particular, we used:

AMD AMD Ryzen Threadripper 2920X 3.5GHz. L1 cache:32KB (8-way), L2 cache: 6MB, L3 cache: 32 MB (32-way) gvv 10.1.0, clang 10.0.0 .

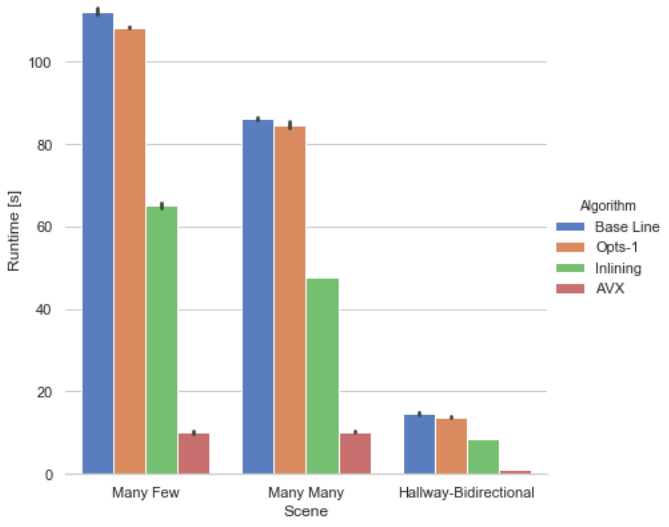


Fig. 12. Run time bar plots for all optimizations

MB1 Macbook Pro, Intel Core I7 Kaby Lake, Peak (Scalar):4 flops/cycle, 3.10GHz, L1 cache: 32.0 KB (2 way), L2 Cache: 256 KB (2 way), L3 Cache: 4.00 MB (1 way) compiler:`clang-1000.11.45.5`

MB2 Macbook Pro, Intel Core i7 (i7-5557U) 3.1 GHz, L1 cache: 32.0 KB, L2 Cache: 256 KB, L3 Cache: 4.00 MB, compiler: `clang-1100.0.33.8`

We fixed several (5?) scenes that we used during development to test the optimizations against. They were used to check for correctness, but also evaluate the optimisations. The idea was that the scenes focus on different part of the simulation, such that we don't overfit the optimisations for one specific scene. The scenes are:

1. Hallway Bi-Directional (200 agents, 2 borders, 3000 steps)
2. Many Few (1000 agents, 2 borders, 1000 steps)
3. Many Many (900 agents, 25 borders, 1000 steps)

The issue with the scenes are that they are very small compared to the cache size. The biggest scene has $1000 * 13 * 4 + 2 * 4 * 4 = 52032$ bytes about 52KB, while the smallest L3 cache that we have has 4mb. The biggest scene however already takes several minutes to run. Increasing the scene to be larger than the L3 cache size means that each run takes several hours for the unoptimised code. We still tried to accommodate for the cache sizes by fixing one scene (the borders), drastically reducing the number of timesteps, and increasing the number of agents as you can see in figure 13.

Results.

We also tried to measure multiple runs per optimization to account for outliers. All the bar plots show error bars that are computed with bootstrap consistences. However, for certain optimizations and scenes this was not feasible in a reasonable amount of time and we only did about 5 measurements, whereas the fastest scenes might have up to 20 measurements.

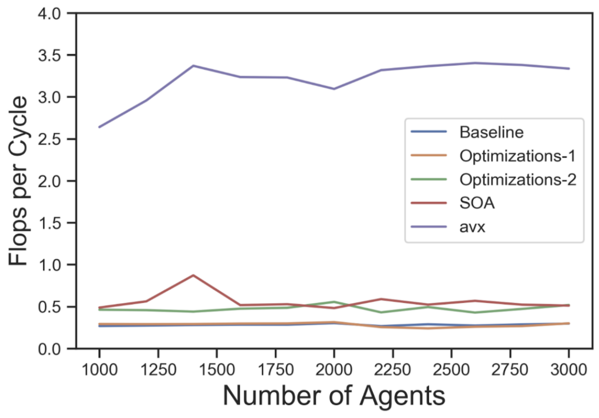


Fig. 13. x-axis: varying amount of agents, y-axis: flops per cycle. Multiple lines, where each is a different optimization. (Ran on `MB1`) Flops were calculated from instrumented code.

5. CONCLUSIONS

In this paper we presented a fast implementation of a simulation of the Social Force Model algorithm. We achieved a roughly 1.8x speed up in our scalar optimizations and up to 12x speed up for vectorization using AVX256 intrinsics. Our focus was on optimizing the main bottleneck in the baseline code, namely the repulsive force between the different agents as explained in section 2. Our scalar optimizations consisted of code motion improvements, precomputations, and most significantly code inlining which removed a large portion of the overhead of our function calls and achieved the highest speed up for our scalar optimizations. Since our main bottleneck is computing the force between agents and this computation is independent within a time step, we vectorized the code in order to parallelize the computations performed during this force calculation per time step and naturally this approach achieved the highest speed up.

6. FURTHER COMMENTS

One of the main limitations we faced in our experimental setup is the lack of tooling. `perf` only runs on linux, but the linux machine we had access to has an AMD cpu with a completely different microarchitecture than the intel CPUs. Additionally, `perf` is very difficult to interpret. It is not clear, and also not documented, how to interpret the different event counters. For cache simulations `cachegrind` is an alternative, however this never represents how the program *really* behaves as it is just a simulation, takes nearly an hour to run, and gives contradictory results compared to `perf`. In addition, the intel tools such as `vtune` or `intel advisor roofline` don't allow local binary analysis on mac os. While they run on linux, they provide sparse information in our setup, beacuse the AMD cpu is not supported. In particular, the roofline plots produced by `intel advisor roofline` looked very wrong as they contained loops above several roofline bounds.

We also tried other tools to measure specific information, account for memory layout changes, and create a roofline plot. An alternative to the intel performance measuring tools is `amd uprof` which runs on linux on the AMD machine. This tool however measures exactly the same information as `perf`, and doesn't provide more functionality than it. It's as complex to use as `perf`, because it doesn't provide detailed information what the measured events actually refer to. `Stabilizer`[5] is a tool which randomizes the memory layout during runtime to make sure that the optimizations are really the cause of the speedup, and not a new (random) and more favorable memory layout (e.g. function doesn't span two pages anymore). We tried several days to get a working version by downgrading LLVM, running it on old ubuntu versions or in isolated docker containers but never got it working. The last tool that we tried is the `Extended Roofline Model` [6] which should create detailed roofline models. We ran into similar outdated issues as with `Stabilizer` and never got the tool to run, despite downgrading LLVM and modifying source code to include apparently missing headers.

As already mentioned, we could not find any high-performance implementation of the exact same algorithm and thus were unable to compare our results with other work.

7. CONTRIBUTIONS OF TEAM MEMBERS

This section describes the work that each team member did. In addition to the individually listed work, each team member worked on bug hunting for the baseline implementation.

Juan. Worked along with Thierry on general C optimisations and AVX vectorisation. Worked alone on the struct of arrays, and did most of the measurements and plotting. Instrumented code to count flops.

Thierry. Implemented the full simulation in C, created scene loading, scene generation, and a benchmarking infrastructure. Worked along with Juan on general C optimisations and AVX vectorisation. Worked alone on struct reordering, inlining, and with `perf`, as he is the only one who has access to a linux machine. Tried out different tools as described in section 5.

Lam. Implemented and analysed loop unrolling. Created a tool to visualize the simulation output.

8. REFERENCES

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

[2] svenkreiss, "Social force model python implementation," `https://github.com/svenkreiss/socialforce`.

[3] "Wikichip zen microarchitecture," `https://en.wikichip.org/wiki/amd/microarchitectures/zen#Memory_Hierarchy`.

[4] "Avx mathfun header," `https://github.com/reyoung/avx_mathfun`.

[5] "Stabilizer," `https://github.com/ccurtsinger/stabilizer`.

[6] "Extended roofline model," `https://github.com/caparrov/ERM`.