FASTERSLAM: AN ACCELERATED SINGLE-CORE 2D FASTSLAM IMPLEMENTATION

Romeo Valentin, Philipp Lindenberger, Nikolaos Tselepidis, Jonathan Lehner

Advanced Systems Lab, Spring 2020
Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

We propose an optimized implementation of FastSLAM with known data association for a single core. A benchmark driven optimization approach is presented, with detailed studies on function performance and input scaling. We introduce a new memory layout for the particles and a memory-efficient resampling method. Densely vectorized AVX2-implementations of the prediction and observation step, based on 2D linear algebra kernels, trigonometric function approximations and pseudo-random number generation, are presented. We discuss the efficiency of our proposed optimizations in terms of performance and operational intensity. To validate our method, we test our proposed method on the Victoria Park Dataset.

1. INTRODUCTION

Motivation. Autonomous mobile robots play an increasingly important role in industry as well as in daily life. Warehouse logistics, driver assistance systems, search and rescue robots, as well as autonomous vehicles are some of the main applications. For these robots, self-localization is still one of the main challenges, especially when located in unknown environments. Noisy sensor data as well as limited computational resources create a need for robust and efficient algorithms. This need is further increased by the fact that these robots usually run a variety of algorithms at the same time, which makes performance increase of individual algorithms even more valuable.

In 2002, Thrun et al. [1] proposed the FastSLAM algorithm as a computationally feasible solution to the problem of simultaneous localization and mapping. While this algorithm has a reduced time complexity, a naive implementation will achieve limited performance due to bottlenecks such as bad cache locality, inefficient instructions and unnecessary function calls.

Contribution. In this paper we present an efficient implementation of the FastSLAM algorithm for 2D point cloud data and known data association optimized for the *Intel Skylake* microarchitecture. We show how certain parts of the algorithm can be restructured in order to reduce computations and boost performance, how certain data structures can be modified in order to increase efficiency and improve data locality, and how certain math operations can be accelerated using low order approximations where possible.

We analyzed the code to identify bottlenecks, which we optimized step by step. Overall, we managed to achieve approximately a $60\times$ speedup over a popular open-source C++ implementation[1] based on Eigen.

Related work. In 1986, Smith et al. [2] first introduced the SLAM problem as a way to create a map from sensor measurements with uncertain robot poses. In 2001, Dissanayake et. al [3] introduced the first highly successful SLAM algorithm based on a single Extended Kalman Filter (EKF SLAM) that keeps track of all the features at once. Although successful, the algorithm suffers from poor scalability with a large number of features, making it computationally infeasible for many real-world applications. In 2002, Thrun et al. [1] introduced the FastSLAM algorithm as a more computationally efficient solution by exploiting conditional independence between the features given the robot path. This algorithm was then further improved by the authors, resulting in the FastSLAM 2.0 [4] algorithm, which improves sample generation in certain situations, but at the expense of algorithmic complexity. In this work we only consider the original FastSLAM algorithm.

2. BACKGROUND

The main idea of the Fast Simultaneous Localization and Mapping (FastSLAM) algorithm is that, given a set of control inputs u_t and feature measurements z_t at a number of time steps $t \in \{1, \dots, T\}$, the algorithm provides a robot pose estimation $s_t = (x_t, y_t, \phi_t)$ at each time step (*localization*) while building a position estimation of all the landmarks/features (*mapping*). For our algorithm we consider a list of two dimensional features $\theta^{(i)} = (x^{(i)}, y^{(i)})$ with $i \in \{1, \dots, N_f\}$, which can be measured by a sensor, yielding the measurements $z_t^{(j)} = (r_t^{(j)}, \alpha_t^{(j)})$. Although the fea-

[1] `github.com/yglee/FastSLAM`

Algorithm 1: FastSLAM

Input : Controls and measurements
Result: Feature and robot pose estimations

```
while not arrived do
    u_t ← obtain_controls(t, ...);
    predict_update(p, u_t, Δt) ∀p in particles;
    if observe then
        z_t ← obtain_measurements(t, ...);
        observe_update(p, z_t) ∀p in particles;
            weights ← compute_weights(z_t);
            update_features(particles, z_t);
                KF_update(particles, z_t);
            resample_particles(weights);
    end
end
```

tures themselves are unknown to the SLAM-algorithm (as they are to be estimated from the measurements), we assume we have known data association, i.e. we know the association of measurements found at different timesteps to the same feature.

Additionally, we have a robot motion model that is given by some function $h(s_{t-1}, u_t)$ as well as a feature measurement model $g(s_t, \theta_t)$. Both models are assumed to be disturbed by some normally distributed noise $\delta_t \sim \mathcal{N}(0, P_t)$ and $\epsilon_t \sim \mathcal{N}(0, R_t)$ respectively, yielding a probabilistic description at time t of the robot's state

$$p(s_t|s_{t-1}, u_t) = h(s_{t-1}, u_t) + \delta_t, \tag{1}$$

and of the feature measurements

$$p(z_t|s_t, \theta) = g(s_t, \theta) + \epsilon_t \ . \tag{2}$$

In order to deal with the uncertainty, a particle filter algorithm is used. Instead of having a single estimate of the pose s_t and feature locations θ, many guesses are kept as realizations of the random variables, propagated using the motion model, weighted by their likelihood given the measurements and resampled according to their weights. Each of those guesses $S_t^{[m]}$, called a particle, consists of a pose estimation and an Extended Kalman Filter (EKF) for the position of every known feature, i.e.

$$S_t^{[m]} = (s_t^{[m]}; w_t^{[m]}; \mu_t^{[m],1}, P_t^{[m],1}; \mu_t^{[m],2}, P_t^{[m],2}; \dots) \ ,$$

where $s_t^{[m]}$ and $w_t^{[m]}$ represent the estimated pose and particle weight and each pair $< \mu_t^{[m],j}, P_t^{[m],j} >$ represents an Extended Kalman Filter for the jth tracked feature. $\mu = (x, y)$ denotes the tracked position together with corresponding covariance matrix P.

In Algorithm 1, we present the three main components of FastSLAM, namely `predict_update`, `observe_update`, and `resample_particles`.

`predict_update` updates the state of each particle by applying the motion model and adding some noise sampled from the $\mathcal{N}(0, P_t)$.

`observe_update` takes all measurements at a given time step and uses them to *(i)* compute the weight for each particle, *(ii)* initialize a new EKF for each new feature and *(iii)* update previously known feature estimations with new measurements using an EKF update for each feature. This function is only called if *observe* is true, which for us is the case every eight iterations. Physically that means that we assume we can give a control input at eight times the frequency that we can measure with our sensor.

`resample_particles`, which is part of `observe_update`, redraws with replacement N_p new particles from the available N_p particles according to their previously computed weight. This method has a starvation criteria that triggers when a few particles hold the majority of the weight, and only then we resample.

The input functions `obtain_controls` for odometry and `obtain_measurements` for feature observations are either mapped to input data or can be simulated from ground truth for testing purposes.

Cost analysis. Our cost measure is defined as the sum of the call count of basic instructions multiplied with their respective flop costs, i.e. $c_{total} = \sum_{op} N_{op} \cdot c_{op}$. The call count N_{op} is measured through manual instrumentation of every function, and we defined costs for double-precision math instructions (`add`, `mul`, `div`, `sqrt`, etc), trigonometric functions and for `rand`. The respective costs were obtained by measuring the relative cost of these operations w.r.t. `mul` and `add`. Integer operations were not considered. The experimental hardware setup used is described in Section 4.

3. METHOD

In this section we introduce FasterSLAM, an optimized implementation of the FastSLAM algorithm, that utilizes vector intrinsics and is targeted at fast execution on a single core. First, we discuss how we set up the memory layout to improve the cache locality of the overall algorithm. Furthermore, we present how we optimized the low-level functions, i.e. functions that don't depend on many subfunctions. Finally, we explain how we integrated the low-level optimizations in the main functions of the FastSLAM algorithm in order to boost performance.

Benchmark-driven improvements. In order to identify the hotspots, i.e. the main parts of the code that constitute bottlenecks, we first extensively benchmarked our "naive" baseline implementation using the profiling tools Valgrind [5] and GProf [6]. Then, as hotspots were targeted and removed, we kept iterating between code improvement and benchmarking, introducing manual instrumentation for exact hotspot analysis.

Algorithm 2: resample_particles

$\mathcal{I} \leftarrow$ draw_particle_indices(w);
$\mathcal{C} \leftarrow$ count_memory_dependencies($\mathcal{I}$);
$i \leftarrow$ find_particle_without_dependencies($\mathcal{C}$);
while i *is valid* **do**
    copy_into($\mathcal{P}[i]$);
    remove_dependency_from_C($\mathcal{P}[i]$, $\mathcal{C}$);
    $i \leftarrow$ find_particle_without_dependencies($\mathcal{C}$);
end

3.1. Low-level optimizations

In this section we present how we set up the memory layout in order to improve cache locality, and we introduce an efficient resampling algorithm that alleviates the need for continual heap allocation. Then, we discuss our design of specialized routines for 2x2 matrix operations, and we introduce a Chebyshev `sin` approximation on a bounded domain. Finally, we discuss our choice of a pseudo random number generation algorithm.

Particle memory layout. The initial profiling of the first baseline implementation indicated that poor data locality as well as frequent memory allocations and deallocations were leading to substantial slowdowns of the overall code. We propose an efficient memory layout for the structure holding the particles. While in a straightforward implementation one would naturally store the weight w_i and the estimated pose (x_i, y_i, ϕ_i) inside the data structure of the particle $S_t^{(i)}$, instead we store all the weights w_i as well as the estimated poses (x_i, y_i, ϕ_i) consecutively in memory, and keep the corresponding *addresses* in the structure holding the corresponding particle. This corresponds to the well known structure-of-arrays vs array-of-structures discussion. By choosing the structure-of-arrays design we leverage cache locality for algorithms that sequentially access the weights or poses of multiple particles. It should be stated that, in the initialization of the algorithm, we allocate a large buffer of predefined size for each particle's feature estimates to avoid frequent reallocations.

Resampling algorithm. The other bottleneck in terms of memory allocations is the resampling algorithm. The basic resampling algorithm first draws with replacement a set of N_p particle indices according to the weights attached to the N_p existing particles S_{t-1}. Next, a temporary deep copy of all the particles $S_{tmp} \overset{copy}{\leftarrow} S_{t-1}$ needs to be created. Finally, the particles with indices that have been drawn in the first step get copied from the temporary storage back into the main storage, overwriting the old values and creating the new set of particles $S_t \overset{copy}{\leftarrow} S_{tmp}$.

This method has the obvious disadvantage that a full temporary deep copy of the particles needs to be created for each resampling, including allocating heap memory for the temporary storage. We propose using a directed-acyclic graph (DAG) to keep track of the memory dependencies between the particles and removing the need for a temporary copy, as well as for any heap allocations. This way we can find particles that don't have a dependency upon them, change their memory content and remove their dependency from the graph. In this way we can process every particle in the DAG until all copies have been made. A pseudocode implementation is provided in Algorithm 2.

Fast 2x2 matrix operation kernels. Profiling the baseline code indicated that the linear algebra operations, and specifically the matrix-matrix multiplications, constitute an additional important bottleneck for the overall algorithm. Since we consider features in 2D, all linear algebra operations are actually performed on 2×2 matrices and 2×1 vectors. This allowed us to implement specialized matrix computations for 2×2 matrices inspired by the interfaces utilized in BLAS [7], as well as to allocate all matrices and vectors in the stack instead of dynamically in the heap.

For single matrix and vector inputs we implemented and vectorized using AVX computations like $C = A \cdot \text{op}(B)$ and $C \mathrel{+}= A \cdot \text{op}(B)$, where $\text{op}(X) = X, X^T$, for X either a vector or a matrix. Apart from vectorized functions for single matrix and vector inputs, we also implemented computations that operate on *batches*, i.e. four different inputs at a time. This is especially useful for use in unrolled loops of higher level functions and allows for better utilization of AVX instructions. Specifically, we implemented a batch $y = x^T A x$ computation, that operates simultaneously on four matrices and four vectors, as well as a batch inverse matrix computation that computes four inverses of 2×2 matrices at a time. In order to identify the "vectorization pattern" in the batch computation of $y = x^T A x$, we expanded the equation and made use of the notion that $y = x^T A x = (Ax)^T \cdot x$, when A is symmetric. This expansion allowed us to derive an explicit expression which can be implemented efficiently using AVX instructions. For the vectorized version of the batch inverse function, after loading the four matrices into four registers, we transpose the contents of the registers to enforce a vector pattern which cannot be identified otherwise, and proceeded with the use of vectorized muls and permutes, allowing for an efficient vectorized implementation.

Fast sin/cos approximation. Another bottleneck in the baseline implementation is the computation of `sin` and `cos` which occur in the motion model $h(s_{t-1}, u_t)$ in `predict_update`. Since the `sin` and `cos` differ simply by an input shift of $\frac{\pi}{2}$ we only discuss the `sin` implementation. One of our major insights is that, since the motion update step presented in Eq. (1) is disturbed by noise, it is not necessary to perform a highly accurate `sin` computation like the one that the `cmath` implementation provides. Further-

more, if we assume that we can bound our input angles to some bounded domain like $(-\pi, \pi]$ we can leverage this to create a low order approximation only valid on this domain. This assumption will be discussed in the next section. This allows us to utilize a lower order function approximation like a polynomial approximation valid on the bounded domain to approximate the `sin` computation in a fast way.

We chose a Chebyshev interpolation of `sin` to the 11^{th} degree, which can be written as $\sin(x) = \sum_{i=0}^{5} c_{2i+1} x^{2i+1}$ and efficiently evaluated using Horner's scheme, i.e. by applying repeated multiplication and addition of the coefficients with x:

$$sin(x) = c_1 + x(c_3 + x(c_5 + x(c_7 + x(c_9 + x(c_{11} + x))))).$$

This equation can be computed efficiently using AVX intrinsics in two ways: First, FMA instructions can be used to efficiently fuse the multiplication and addition computations after computing the initial $(c_{11} + x)$. Second, when computing multiple `sin` values at the same time, AVX can be used to compute four FMA instructions at the same time for multiple inputs. Using these improvements this method is theoretically capable of almost achieving the theoretical peak performance of a Skylake architecture.

Angle normalization. As discussed in the previous section, the `sin` approximation requires the input to be in the range $(-\pi, \pi]$. We can transform any input angle to that range by using a `mod` operation, which is expensive. An alternative method is to continually subtract 2π while the input is greater than π (and vice versa for less than $-\pi$), which can give a better performance for angles close or within the desired range. Furthermore, if we know that our angle is computed as some $\alpha + \Delta\alpha$ with $\alpha, \Delta\alpha \in (-\pi, \pi]$ it is actually sufficient to just use a single `if-elseif` clause to check if 2π need to be added or subtracted. This gives rise to an efficient normalization scheme that only works on specific inputs, but is sufficient for our case.

Fast random number generation. In `predict_update`, we generate a set of random numbers for the state prediction of each particle (as a realization of the random variable δ_t), and this turned out to be a major bottleneck.

Thus, we benchmarked several common pseudo-random number generator algorithms, including `wyhash64`, `xorshift128+`, and `pcg32`, to identify the most efficient generator for our application. Although not all algorithms are statistically valid, experiments showed that for this application all mentioned algorithms are sufficiently random (see the validation section). After benchmarking, we decided to use a vectorized implementation of `xorshift128+`, which showed the highest speedups.

3.2. Higher-level function optimizations

In this section, we explain how we use the presented ideas together with code reordering techniques, manual loop unrolling and inlining to speed up the FastSLAM algorithm.

predict_update. The `predict_update` function is one of the two core functions of the FastSLAM algorithm. It updates the particle states based on the motion model, while also adding gaussian noise. We used heavy inlining in order to maximize precomputations. In order to vectorize it efficiently, we unrolled the loop over four particles at a time. We inlined `multivariate_gauss` and vectorized it using our fast random number generator and linear algebra routines. Moreover, we utilized our fast trigonometric functions based on the Chebyshev polynomial approximations as well as the vectorized angle normalization routine. In order to efficiently compute the updates, we used register transposition to enable the use of FMAs.

KF_update. In the `KF_update` function, the EKF matrices for every feature in each particle get updated, making it one of the most called functions. Since the `KF_update` function is called inside a loop that iterates over all particles in `observe_update`, we designed an unrolled implementation that takes as input four particles at a time, and makes use of our fast matrix operation kernels including the fast batch inversion function. It should be noted that unrolling the function by a factor of four made it possible to compute simultaneously two matrix-vector updates of the form $x_i = x_i + W v_i$, where $x_i, v_i \in \mathbb{R}^2$, $i = 1, 2$, using a single matrix-matrix multiplication kernel, i.e. $X = VW^T + X$, making more efficient use of the AVX registers.

observe_update. Analysis using GProf and Valgrind showed that this function is the main bottleneck in terms of runtime, contributing 60% of the total cycles in the baseline implementation. Thus, most of our optimization efforts were focused into this function. First, we simplified the code and reordered some computations to make vectorization easier. Then, coordinated with optimizations in `KF_update` and `compute_jacobians`, we performed efficient reordering of instruction calls, 4x unrolling on particles, reuse of computations (e.g. transposes) across functions and consistent register patterns to minimize permutes and shuffles. Here, we also used the previously introduced batch computations for matrix inverses and determinants. Since many optimizations required us to transpose 256-bit registers and compute matrix transposes (mapped to one 256-bit register), we used an ordering of $(0, 2, 1, 3)$ for most floating point registers, which significantly reduced permutations. Functions where no AVX-intrinsics were available, like the exponential and atan2, we implemented approximations similar to sin, using the open source library AVX Mathfun[2] as a starting point.

[2]Giovanni Garberoglio, `software-lisc.fbk.eu/avx_mathfun/avx_mathfun.h`

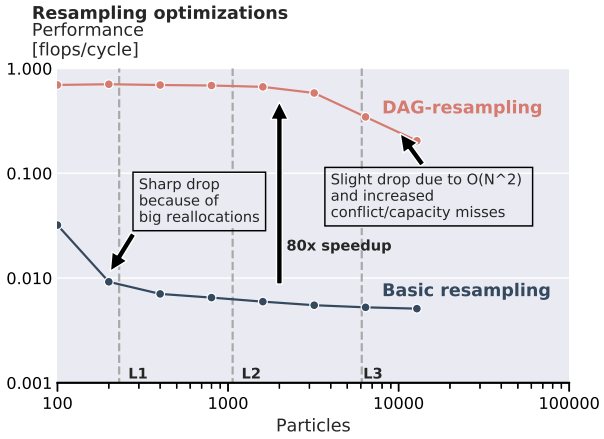

Fig. 1: **Resampling performance scaled with particles, $N_f = 20$ features.** Our DAG-algorithm scales significantly better than then the base resampling method. Steady drop in performance due to increased cache misses and a non-optimal algorithm. Gray dashed lines show cache capacities.

4. EXPERIMENTAL RESULTS

4.1. Experimental setup

Our experiments were conducted on a Dell XPS 15 9550 with an Intel Core i7-6700HQ @ 2.6 GHz. We disabled TurboBoost for our measurements to get consistent benchmarking results. The processor belongs to the Skylake microarchitecture with cache sizes of 4x 32 KB for each L1 instruction cache and for each data cache, 4x 256 KB for the L2 caches and 4x 1.5 MB for the L3 cache. The peak performance of a single core is $\pi_{SIMD} = 16\frac{flops}{cycle}$, which is achieved by utilizing pipelined FMA instructions on both FMA ports at the same time. The peak memory bandwidth is $\beta = 6.5\frac{bytes}{cycle}$, which was determined by running the Stream benchmark [8]. When not noted otherwise, we compiled with *gcc 7.4.0* and the flags *-O3* and *-march=native* enabled. We also tested *icc* and *clang* compiler but did not get a consistently different results compared to *gcc*.

Benchmarking architecture. We used *rdtsc*, a time stamp counter that accurately counts cycles on Intel CPUs, to benchmark our functions. Each main function was instrumented manually to count the exact instruction mix of each function given the inputs, and we counted the estimated memory movement (reads + writes), ignoring conflict misses. Still, our memory estimations tend to a bit higher than what cachegrind suggests. Combined, we aim to give precise measures for speedup, performance and operational intensity by wrapping each function in a benchmark.

4.2. Results of low-level functions

Performance of resampling. A performance plot of our resampling algorithm for different number of particles is presented in Fig. 1. The blue line represents the naive implementation, where we create a deep copy of all particles and sample from there. The orange line is our improved resampling method that does memory movements inplace, thus not requiring additional heap allocations in each call. The data moved per particle is proportional to

$$Q = N_f(2+4)+1+3+9\,[doubles] = 48N_f+96\,[Bytes].$$

In this benchmark we tested with $N_f = 20$ features known to each particle. We observe that for the basic resampling method, we already see a sharp drop at $N_p = 200$ particles. Our machine has an L1 Cache size of 256 KB (first gray vertical line), which is exceeded slightly above 200 particles, thus justifying the sharp drop in the baseline implementation. For our improved method, we see a much smaller performance drop, and for a small number of features like this we achieve a speedup of around $80\times$ versus the basic resampling method (on our memory layout). Still, we also see a slight decrease in performance in the the improved resampling method. This has mainly two reasons: First, looking for particles without dependencies takes $\mathcal{O}(N^2)$ time, and since these are only int operations (comparisons on a $N \times 1$-array, which we excluded from our measurements), the performance will drop when increasing the number of particles. Second, the proposed method suffers from non-consecutive memory accesses (due to some particles being not redrawn), thus introducing conflict misses. Finally, capacity misses are also increasingly problematic since we exceed the L3 cache capacity above 6000 particles.

Trigonometric functions. As expected, the approximated `sin` implementation gives both a big speedup compared to the *cmath* implementation as well as a high performance in general, but at the expense that we only achieve an accuracy of about 10^{-8}. When computing the `sin` function on a large vector of doubles that are assumed to be in $(-\pi, \pi]$, we achieve a peak performance of about $14.38\frac{flops}{cycle}$, which is close to the theoretical peak performance of $16\frac{flops}{cycle}$ when only using FMA instructions. The speedup is approximately $14\times$ compared to the *cmath* implementation. Introducing the simplified angle normalization still maintains a high speedup of about $12.5\times$, when the implementation is written using AVX FMA instructions.

When the `sin` implementation is done without using explicit AVX instructions, the speedup decreases to $4\times$, indicating that the Horner's scheme combined with manual FMA instructions and unrolling for four computations at a time make a big difference. Additionally introducing full angle normalization instead of partial angle normalization drops the speedup by another 25%, leaving us at approxi-

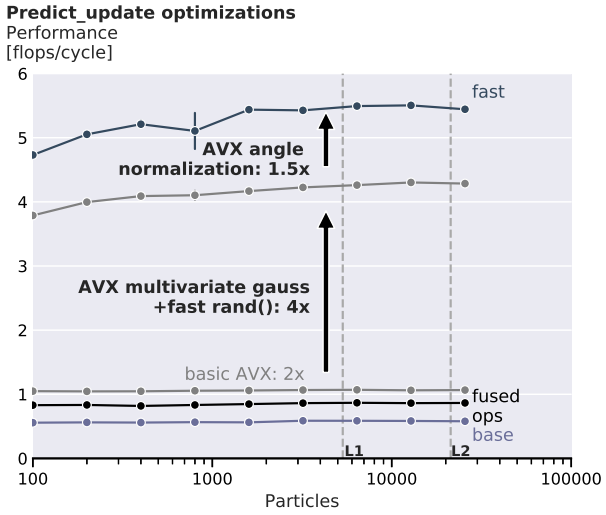

Fig. 2: **Comparison of different predict_update optimizations.** $P_{opt} = 11.21 \frac{flops}{cycle}$. This plot shows the performance of different optimization stages of the `predict_update` function. Black line is the new baseline with reduced flops. Gray dashed lines show cache capacities.

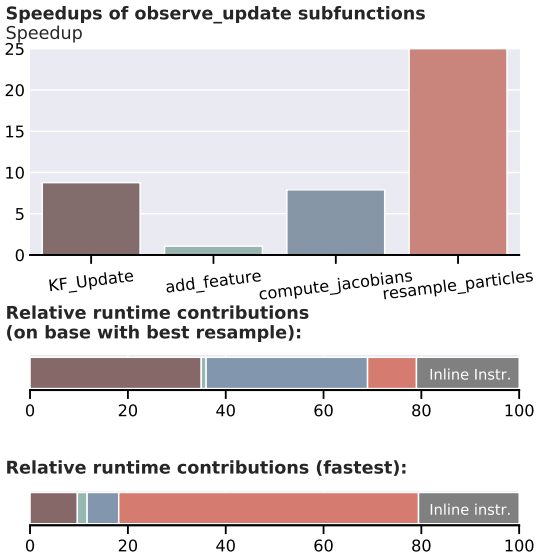

Fig. 3: **Speedups and runtime contributions within observe_update.** Top chart shows speedup of subfunction, middle chart shows relative runtime contributions in % on the base implementation but with our faster resampling, and the bottom chart shows relative runtimes in our fastest implementation. Relative runtime data was obtained on $N_p = 1000$ particles and output is from GProf.

mately a $2.8\times$ speedup compared to the *cmath* implementation, but at the cost of lower accuracy.

Linear algebra functions. For all of the vectorized linear algebra functions we achieve performance results between 2.2 and 3.1 flops/cycle. Considering the small input sizes and low number of computations, as well as the sequential dependencies, we consider this reasonable.

Fast pseudo-random number generators. Both for the scalar and vectorized implementations, `xorshift` yields the biggest improvements, with speedups of $S = 3.75$ and $S = 9.8$ over standard `rand()`, respectively.

Cache analysis. We evaluated our full simulation with the cache profiler Cachegrind, which is part of Valgrind. It simulates the first-level and the last level caches and counts cache hits and misses. Since our processor has three cache levels, it checks the L1 instruction and L1 data cache, as well as the L3 cache. The L3 cache is the lowest cache above the main memory. We are most interested in last level cache misses, since fetching memory from main memory is the most expensive. We compared Cachegrind results with 100 to 12800 particles. With our final code, the L1 data cache miss rate remains well below 4% CMR for all amounts of particles tested, with only a slight but steady scaling with particles, supporting our locality claims. Reads and writes are mostly balanced, with a tendency towards more reads when using more particles. The last level cache instruction misses, i.e. the L3 misses, are below 0.02% up to 25600 particles, even though the computations clearly exceed the L3 capacity of 6MB.

4.3. Results of higher-level functions

Optimizations and performance in predict_update. Our best implementation of `predict_update` achieves 47% of the theoretical peak performance of $11.21 \frac{flops}{cycle}$, determined by the function's instruction mix and the theoretical throughput (sequential dependency was not considered). The impact of each of the steps to achieve this improvement is illustrated in Fig. 2. It should be noted, that due to the approximation of `sin` and some simple pre-computations, the work in the optimized implementations was reduced by 10% compared to the base implementation – though, from the first optimization listed (i.e. the black line), the flop count remains equal. The use of vectorized trigonometric functions and basic vectorization hardly yielded any performance improvements, mainly due to the low relative cost of these functions. 4x unrolling on the particles, which enabled more efficient vectorization, along with the use of our fast random number generators yielded a significant performance boost. Utilizing the simplified normalization in the `sin` approximations and vectorizing the (reduced) normalization on the state angle led to another performance boost, finally resulting in a $11\times$ speedup of the fastest implementation of `predict_update` against its baseline. Also note that the performance of this function remains steady when scaling with the number of particles.

Optimizations and performance in observe_update. In Fig. 3, the analysis and optimization efforts are summarized. Note that we use the improved resampling algorithm in our baseline implementation. The top figure shows the speedup of the main subfunctions against their own baseline. Both `KF_update` and `compute_jacobians` achieve good speedups against their baseline. Both function's best optimizations were unrolled on the number of particles. The highest speedup was obtained in `resampling`. The middle and lower chart shows the relative contributions of submethods before and after our optimization efforts. While resampling particles was not a bottleneck after changing it to the DAG algorithm, after optimizing the other methods in `observe_update`, it is now the main bottleneck again. The main steps towards our fastest method are shown in Fig. 4, where we scale different optimization stages with the number of particles. We achieve a total speedup of around 8x in `observe_update`. By fusing some operations, e.g. `feature_update` and `weight_computation`, we reduced the work by approximately $\frac{1}{3}$, which is accounted for with the black baseline. All above have the same work. In all optimizations, we observe three performance drops, exactly when the capacities of the three cache levels (L1, L2, L3) are reached, which is as expected.

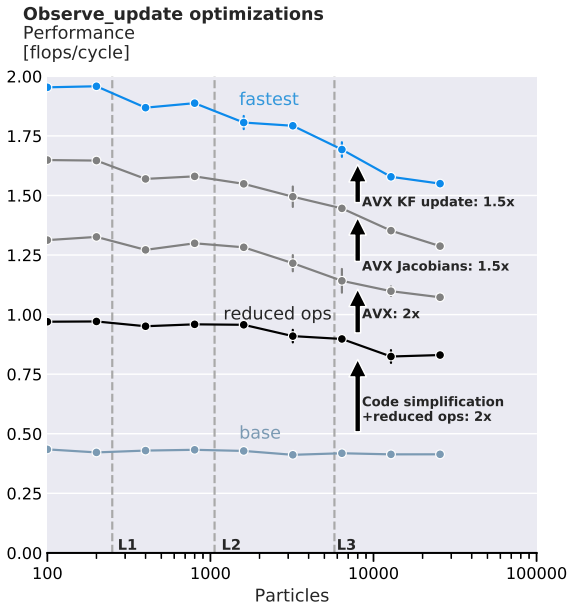

Fig. 4: **Comparison of different observe_update optimizations.** $P_{opt} = 6.88 \frac{flops}{cycle}$. This plot shows the performance of different optimization stages of the `observe_update` function. Black line is the new baseline with reduced flops.

Overall, our best `observe_update` is still only at around 25% of its optimum SIMD peak performance of

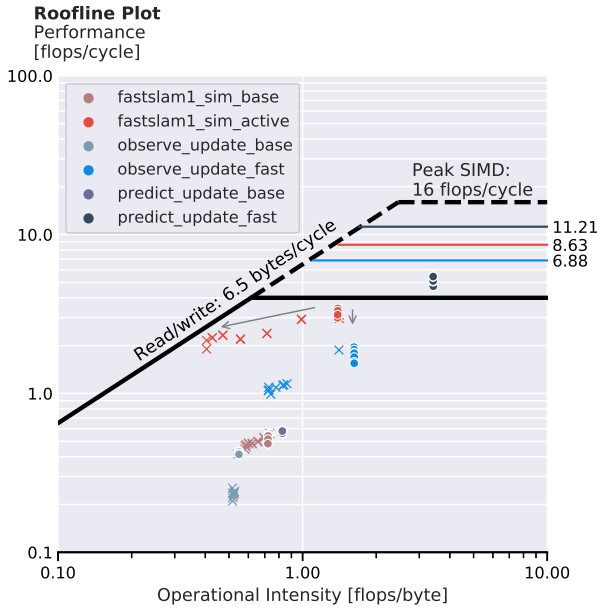

Fig. 5: **Roofline plot with scaling against particles (•) and features (×).** Roofline of the proposed simulation and both main subfunctions, `predict_update` and `observe_update`. Ratio of these functions in the simulation was 8 predictions followed by one observation. Colored flat lines show maximum theoretical performance based on the final instruction mix of each method, but without considering sequential dependencies.

$P_{opt} = 6.88 \frac{flops}{cycle}$. The reason for that is on the one hand the data-movement of resample particles, and on the other hand many required permutations (across the 128-bit lane) and blends, both in the subfunctions and inline instructions.

Fastslam1 simulation. Fig. 5 shows the roofline plot for our main simulation and its two main components. For all functions, the active implementation was greatly improved in terms of operational intensity and performance compared to the base implementation, which also allowed some functions to be compute- instead of memory-bound. When scaling with the number of particles (• markers), we observe that all functions remain stable in the roofline, both in terms of performance and operational intensity. On the other hand, when scaling with the number of features (× markers, arrow denotes increasing input size), both `observe_update` and `fastslam1_sim` move towards the memory bound domain. In our benchmark, this effect is amplified, because more frequent resampling is neccessary, in order to avoid numerical instabilities. This can happen when data points are very close together. Also, memory accesses on the feature are not continuous, i.e. the order of features determines locality. We partially account for that by indexing new features in temporal order, but this might fail to correctly encode features that are close in world position (e.g. because of occlusions), resulting in bad locality

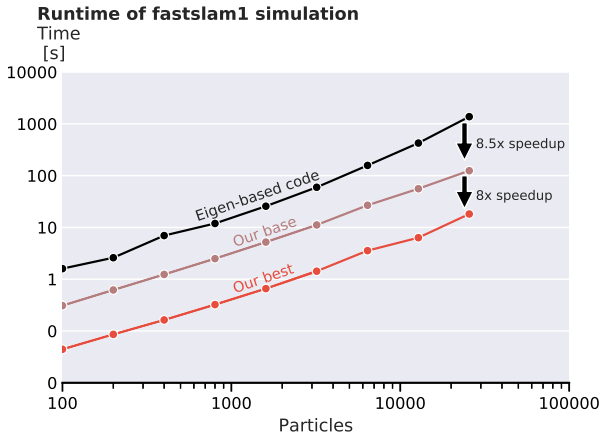

Fig. 6: **Runtime comparison between implementations.** Our improved method against our baseline, as well as a popular open-source FastSLAM implementation. $N_f = 70$.

when we revisit this area.

The colored flat line shows the peak performance based on the final instruction mix of each function. The upper bound was computed by multiplying the number of instruction calls with their theoretical throughputs and considering AVX. This gives a theoretical, but hardly achievable upper bound, since sequential dependencies are not taken into account. Modelling the sequential dependencies in our case is difficult, since they are significantly data-dependent. Considering these inaccuracies in the upper-bound, our methods are reasonably close to the roofline, with `fastslam1_sim_active` being 30% optimal, `observe_update` around 25% and `predict_update` even around 50% optimal.

Finally, in Fig. 6 we present the overall runtime (in seconds) of our optimized FastSLAM implementation, against our baseline, as well as against the previously mentioned open-source C++ implementation based on Eigen. We benchmark all implementations in the same feature map for an increasing number of particles. All implementations show linear scaling with the number of features, our optimizations mainly improving the algorithm by a constant factor. Our proposed method achieved an overall speedup of $8\times$ over our own baseline implementation, which in turn is faster by a factor of $8.5\times$ compared to the Eigen-based implementation.

Validation on the Victoria Park Dataset. Since we made some assumptions about accuracy of certain functions after approximating them, and rely on a pseudo-random number generator, we validated our fastest implementation on the widely-used Victoria Park Dataset. We used the data association published by Shoudong et.al. [9] as an input, with 893 distinct landmarks/features. Our results when using $N_p = 1000$ particles are illustrated in Fig. 7. Our estimation matches the GPS data very closely, arguably sometimes even with higher accuracy and fewer jumps compared to the physical path. This indicates that our assumptions and approximations are valid. Overall, we are able to achieve a $7\times$ speedup over our own baseline on this dataset.

Fig. 7: **FasterSLAM in the Victoria Park Dataset.** Visualization of the estimated trajectory (red) using our fastest implementation. Blue crosses show ground truth obtained from GPS, and black dots are our estimated landmarks. The thin, dashed black line shows trajectory estimation solely based on odometry data.

5. CONCLUSION

In this work we presented how the FastSLAM algorithm can be implemented efficiently by exploiting data parallelism between particles, carefully choosing function approximations for faster evaluation and by implementing optimized linear algebra functions for the 2-dimensional feature setting. We showcased how we used profiling to detect code hotspots and that using manual AVX instructions and loop unrolling techniques can increase the performance of the important functions drastically. Furthermore we showed that trigonometric function approximations and pseudo-random number generation do not impact overall accuracy and convergence in the case of a motion and feature model involving noise. Overall, we were able to increase the operational intensity on the main functions, increasing the theoretical and actual peak performance. Still, for a large number of features, our method remains memory bound by the resampling algorithm. Our benchmarks indicate that the proposed method scales well on large inputs, showing linear scaling with particles and only slight losses in performance in scenes with many landmarks. Finally, we showcased that the implementation is able to achieve good results on a real-life dataset, qualifying it as a promising candidate for performance dependent SLAM challenges.

6. CONTRIBUTIONS OF TEAM MEMBERS

Philipp Lindenberger. Optimized cache layout for Particles. Optimized compute_jacobians using AVX and integrated it into observe_update. Optimized observe_update, unrolling & AVX (subfunctions were cooperations). Optimized predict_update, unrolling & AVX (version before Victoria Park adaption). Angle normalization AVX. Implemented the benchmarking architecture (RDTSC). Scaling benchmarks (features + particles). Fast RNG Benchmark. Part of the initial C++ implementation. Part of the work/memory instrumentation. Worked on making the algorithm run on Victoria Park Dataset. Visualization for debugging/validation.

Nikolaos Tselepidis. Designed and optimized linear algebra kernels using AVX, along with related utilities like register transposition. Implemented and optimized KF_update using AVX. Cooperated with Philipp on observe_update optimization for "smart" interfacing with unrolled KF_-update. Optimized predict_update using AVX (version for Victoria Park dataset). Implemented various benchmarks for several functions. Part of the initial C++ implementation. Part of the work/memory instrumentation. Minor AVX optimizations in other functions like tscheb_sines. Worked on making the algorithm run on Victoria Park Dataset. Profiling using GProf.

Romeo Valentin. Optimized cache layout for Particles. Designed and implemented new resampling algorithm. Implemented Chebyschev sin approximation using AVX. Implemented batch computation for $x^T Ax$ using AVX. (Roofline) plots for presentation. Automatic instruction count utility. Various benchmarks for several functions. Part of the initial C++ implementation. Part of the work/memory instrumentation. Profiling using Valgrind and Callgrind.

Jonathan Lehner. Part of the work/memory instrumentation. Implemented various benchmarks for several functions. Initial compute_jacobians optimizations, with and without SIMD. Refactored compute_jacobians out of compute_weight and feature_update. Compiled list of all optimizations. Worked on missing optimizations in other parts of the code (e.g. better intrinsics instructions, cache optimization, SSA style instead of unnecessary arrays). Smaller part of the initial C++ implementation. Cache analysis. Speed-up/runtime plots. Performance plots.

7. REFERENCES

[1] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit, et al., "Fastslam: A factored solution to the simultaneous localization and mapping problem," *Aaai/iaai*, vol. 593598, 2002.

[2] Randall C Smith and Peter Cheeseman, "On the representation and estimation of spatial uncertainty," *The international journal of Robotics Research*, vol. 5, no. 4, pp. 56–68, 1986.

[3] MWM Gamini Dissanayake, Paul Newman, Steve Clark, Hugh F Durrant-Whyte, and Michael Csorba, "A solution to the simultaneous localization and map building (slam) problem," *IEEE Transactions on robotics and automation*, vol. 17, no. 3, pp. 229–241, 2001.

[4] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit, et al., "Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges," in *IJCAI*, 2003, pp. 1151–1156.

[5] Nicholas Nethercote and Julian Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[6] Susan L Graham, Peter B Kessler, and Marshall K Mckusick, "Gprof: A call graph execution profiler," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[7] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[8] John McCalpin, "STREAM benchmark faq's," 2020.

[9] Shoudong Huang, Zhan Wang, and Gamini Dissanayake, "Iterated slsjf: A sparse local submap joining algorithm with improved consistency," 08 2009.