OPTIMIZING VOLUME ESTIMATION FOR CONVEX POLYTOPES

Constantin Dragancea, David Buzatu, François Costa, Roxana Stiuca

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

The volume of convex bodies is a useful and important attribute used in many applications. However, calculating the exact volume is challenging due to the high dimensional figures used in practice. While theoretical studies have explored volume estimation methods, practical implementations have been lacking. Our approach was built on top of PolyVest [1], which is an efficient method computing the volume leveraging the Multiphase Monte-Carlo algorithm, employing a hit-and-run method along coordinate directions, and incorporates a sample point reutilization technique to reduce the number of operations. We present our PolyVest implementation as as an asymptotically faster algorithm using various optimizations and SIMD intrinsics that achieves better performance and accurately estimates volumes for convex polytopes in high-dimensional spaces.

1. INTRODUCTION

Motivation. Computing an object's volume is a requirement for many applications, from 3D modelling and computer graphics to various domains, such as medical imaging, construction, linear systems modeling, and statistics. While analytical solutions exist for many known shapes, we have focused on convex polytopes in n-dimensional spaces. These are sufficiently expressive to model a wide range of objects, and their representation allows the utilization of numerous methods already extensively studied in linear algebra.

Contribution. Our work is meant to build upon the theoretical findings for Volume Estimation methods and offer a practical implementation that takes into consideration the limitations of current computers architectures. Namely, we target improving PolyVest [1] implemented in C++. Our contributions include an improved complexity resulting in a significant reduction of the number of floating-point operations required for convergence and achieving a faster runtime by applying optimizations techniques, resulting in speed-ups of $\sim$ 5x relative to our baseline implementation. Finally, using vectorization through Intel intrinsics, we have reached more than $\sim$20x speed-up. More details about the conditions for the speed-ups and input sizes can be found in section 4.

The author thanks Jelena Kovacevic. This paper is a modified version of the template she used in her class.

Related work. The problem of volume computation has been studied for many years. Dyer et. al. [2] have proven that the problem is #P-hard for general polytopes. For convex polytopes, Büeler et. al. [3] have shown a number of algorithms for exact volume computation, but their complexity is exponential in regards to the dimension, making them impractical for high-dimensional spaces. Thus, there has been a shift in focus towards approximation techniques, which converge in polynomial time [4]. PolyVest [1] offers a practical implementation with a complexity of $O^*(n^4)$. We have focused on improving its performance by achieving a lower number of floating-point operations and a higher ratio of operations per cycle.

2. BACKGROUND ON THE ALGORITHM/APPLICATION

In this section we formally define a convex polytope, introduce the multiphase Monte-Carlo algorithm used, and perform a cost analysis of the algorithm.

Convex polytopes. The half-space representation describes a convex polytope as the intersection of closed half-spaces, while the vertex representation represents it as the convex hull of a finite set of vertices. The half-space representation focuses on inequalities, while vertex representation focuses the polytope's corner points.

We define in our algorithm a convex polytope as the intersection of a finite number of half-spaces, which is defined as:

$P = \{Ax \leq b\}$, where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

Volume Estimation Algorithm. Our algorithm consists of 2 steps:

1. preprocessing
2. volume estimation.

In the preprocessing step, we find an affine transformation which centers the object at the origin of the coordinate space. Then, we consider r concentric balls of increasing radius - centered at the origin - such that the smallest is completely contained in the polytope, and the largest contains the en-

tire polytope we want to estimate the volume for. Afterwards, for each ball, the algorithm samples a number random points and counts how many are inside the intersection of the ball with the polytope. The product of all ratios of points sampled and points contained by polytope leads to the value of the true volume. For random sampling, the algorithm proposes using the coordinate directions hit-and-run method [5].

Cost Analysis. We analyzed the cost measure of PolyVest[1] by the rough number of cheap and expensive operations performed. We have arrived at the following cost function: $C(m, n) =:$

- $3n^3mlog^2(n)$ additions
- $3n^3mlog^2(n)$ multiplications
- $3n^2log^2(n)$ square roots
- $4n^2mlog(n)$ divisions

We aimed to replace costly operations with equivalent expressions which take less cycles, reduce the overall flop count, and eliminate dependencies between consecutive operations.

The time complexity of PolyVest[1] is $O^*(mn^3)$ or more precisely $O(mn^3(log(n))^2)$.

3. OUR METHOD

We implemented our approach using C, and the computations were performed on double-precision floating point numbers. Due to the nature of the generated tests, in our cost and complexity analysis, we made the simplifying assumption that m is equal to $2n$ and we will express the input size only through one variable: n.

Baseline. The preprocess step uses linear programming, through the GLPK library[6], in order to remove redundant hyperplanes and to generate the initial ellipsoids. Following these steps, a number of matrix-vector operations are executed to find the affine transformation that shifts the polytope to the center, and finally correctly place the polytope and the ellipsoids using the transformation. The latter has an $O(n^3)$ time complexity. For the Simplex algorithm used by GPLK in the first part, it is known that, while it can reach exponential complexity, in practice it is very fast [7] for small n values. This part is hard to measure exactly, but profiling and experiments presented in a subsequent section have shown that for our use cases, it is indeed fast and not a bottleneck of the implementation.

In the volume estimation step, we iterate over the ellipsoids and generate $step\_size$ samples using a *Walk* function. We've set the value of $step\_size$ to the one proposed in PolyVest[1], namely 1600. The function *Walk* implements a hit-and-run sampling method, has a complexity of $O(n^2)$,

Function / Call Stack	CPU Time ▼
walk_with_rand_functions	2.920s
__ieee754_log_fma	0.104s
estimate_volume_with_rand_functions	0.069s
next	0.021s
next	0.016s
func@0x401130	0.011s
rand_double	0.009s
__log	0.008s
rotl	0.002s
top16	0.002s
rand_int	0.002s

Fig. 1. VTune profiling after Basic Optimizations were applied. As it can be seen, the walk function requires by far the most amount of CPU time, the estimate_volume_with_rand_functions method coming next as it depends on walk's execution.

and it is being invoked $1600(nlog(n))^2$ times, making it the bottleneck of the implementation.

Profiling and finding the bottleneck. When optimizing code for improving the performance, it is important to identify the bottleneck, which refers to the part of the code that is causing most of the slowdown. To locate the bottleneck, we manually examined the code to determine where it spends most of the time. The walk function proved to be the fitting candidate after we additionally validated our assumptions using Intel VTune, a software profiler that helps analyze and measure the performance of the code - see Figure 1.

Initial optimizations. We have started by applying techniques learned in the Advanced Systems Lab, which are meant to facilitate compiler optimizations, improve data locality, and gain a higher performance (measured through floating point operations per cycle).

`Basic optimizations`. Firstly, we inlined all the small functions which are called repeatedly. We also placed all code in header files only to enable the compiler to inline methods more easily, resulting in a binary that causes less context switches. Then, we moved all memory allocations and frees to the initialization step, rather than other code blocks which were getting called repeatedly (such as the *Walk* function). This also allows us to keep track more easily of the memory used and mitigate memory leaks.

Next, we pre-computed constants and replaced common sub-expressions. One such constant is:

```
trunc(p->n*log(m)/(log((double)2)*2))+1.
```

We keep the state of the algorithm in a C `struct` which gets passed as a pointer to necessary functions. Many of its components are accessed frequently inside functions, therefore we applied scalar replacement and modified computations to use local variables instead, updating the struct only at the end of function calls. One example of such a reference

is the value inside `p->n`.

`Optimizing preprocess`. Another optimization we applied was in the *preprocess* function. Relying on other methods, it computes a number of matrix-vector multiplications and other matrix operations. In our baseline, we have used BLAS[1] to support these operations - e.g., $T = c1 * (T - c4 * t * t.t())$ -, but a better runtime was achieved by writing the operations through double-loops. This can be explained by resulting in less operations than using the `_dgemv` function that computes $\alpha * A * B + \beta * C$, rather than $A * B + C$.

In our baseline, we have initially implemented this functionality through one level-2 BLAS function call, followed by 3 passes through the matrix for each scalar multiplication or subtraction. However, our final implementation uses a simple double-loop which needs less cycles.

`Faster random number generation`. Finally, we replaced the C `rand()` functions with the xoshiro256+[8] random number generator. This is a faster way of producing random numbers that makes great use of bit operations while preserving statistical properties of randomness.

Algorithmic optimization. The optimizations in this category are distinct from basic optimizations because they focus on improving the algorithm rather than fine-tuning the code itself. As a result, the overall performance, measured by asymptotic time, is significantly improved. Moreover, these optimizations have the advantage of being applicable regardless of the programming language or the underlying hardware used, making them versatile and adaptable.

After applying the aforementioned basic optimizations, we profiled our binary using VTune. We saw that the walk function was by far the bottleneck, further confirming that the complexity analysis we have performed was indeed accurate. We noticed that on each `walk` function call, a dimension is chosen randomly and the following array is computed ($i := 1 \to m$):

$$bounds_i = \frac{b_i}{A_{i,rand\_dir}} - \frac{\sum_{j=1}^{n} A_{i,j} * sampling\_points_j}{A_{i,rand\_dir}}$$

This computation has an $O(mn)$ complexity, but some of the operations can be done only once and then be reused, rather than doing them every time. Concretely, we can take $A^{-1}_{i,rand\_dir}$ as common factor, and then we can reuse the value

$$precomputed_i = b_i - \sum_{j=1}^{n} A_{i,j} * sampling\_points_j$$

giving us an $O(m)$ complexity. Since each `walk` function call updates the array $sampling\_points$, we have to update our $precomputed$ array as well, but this can be done in $O(m)$. Thus, we are able to reduce the complexity of our `walk` function from $O(mn)$ to $O(m)$.

Additionally, inside the `walk` function, the sum

$$\sum_{i=1, i \neq rand\_dir}^{m} sampling\_points_i^2$$

is computed. This can be optimized by storing the sum

$$\sum_{i=1}^{m} sampling\_points_i^2$$

and subtracting the expected $sampling\_points_{rand\_dir}$, while also updating the sum on each change of the $sampling\_points$ array.

Vectorization. In `walk`, we divide a vector `bounds` by a specific column in matrix `A`, then in another loop update the `bounds` vector element by element. These computations can be done in parallel for multiple elements at a time. Thus, we have unrolled the loop by 4 and used SIMD instructions (AVX256 Intel intrinsics) in both loops. Because we access matrix `A` by column, we keep a copy of it transposed. Moreover, as the values of `A` never change and division is an expensive operation in terms of the cycles used, we additionally keep a copy of the $A\_inv^T_{i,j} = (A^T)^{-1}_{i,j}$ computation computed in the preprocess step and perform multiplication instead.

Outside of the *Walk* function, there are more loops with computations necessary to estimate the volume. While these are not the most time-consuming, any optimization helps, and we therefore used SIMD instructions in the *Estimate Volume* function as well.

`Padding technique`. SIMD instructions allow performing multiple operations simultaneously, typically in groups of four. To make effective use of `SIMD` instructions, a hard constraint is to have a vector size that is divisible by four in double precision arithmetic. However, we have no guarantees that the input size will be divisible by this number in the input.

We explored different approaches to address this problem. One idea was to apply `SIMD` instructions by groups of four and then handle any remaining elements using a scalar loop. However, this approach would increase code complexity and make the implementation harder to read as we would have to update each other vector/matrix that our computation is using to match the expected size. Another approach that we call `padding technique` can overcome this issue. The `padding technique` involves constructing an alternative input that produces the same results as the original input, but ensures that the size of the padded input is always divisible by the `SIMD` vector size. This is a `constructive algorithm` that generates a new input set from the previous one. By applying the padding technique, we can utilize `SIMD` instructions.

[1] https://www.netlib.org/blas/

In the C++ code below, the padding technique is applied to the sum of a vector. In the first example with a vector size of 5, AVX instructions cannot be directly applied because it is not divisible by four. However, in the second example, the vector size is divisible by four, allowing the use of AVX instructions. The result remains the same because adding the neutral element ($a + 0 = a$) always yields the same value. In this example, the new input will give the same results than the previous input by construction.

```
int sum() {
    vector<int> numbers = {5, 9, 2, 7, 1};
    int sum = 0;
    for (int i = 0; i < numbers.size();i++) {
        sum += num;
    }
    return sum;
}
```

Listing 1. C++ code that performs the Sum of a Vector

```
int sum() {
    vector<int> numbers = {5, 9, 2, 7, 1, 0, 0,
    0};
    int sum = 0;
    for (int i = 0; i < numbers.size();i++) {
        sum += num;
    }
    return sum;
}
```

Listing 2. C++ code that performs the Sum of a Vector with the padding technique

Unrolling The compiler can do operations in parallel, on different ports. In order to achieve the best throughput for this, we aimed to reorganize the code. We unrolled the loop in *Walk* by 16; for the remaining iterations outside of the largest multiple of 16, we unrolled by 8, then by 4 and finally looped with steps of 1.

AVX-512. We attempted to utilize AVX-512, which is an extension of SIMD capabilities that supports a larger size of 512 bits vector instead of the usual 256 bits vector. When compared to the SSE and AVX families, AVX-512 offers a greater number of instructions and the ability to parallelize up to 8 operations simultaneously. Transitioning the code from AVX to AVX-512 involved a fairly straightforward process, as most instructions were a one-to-one mapping.

Unsuccessful attempts. We will discuss additional attempts we made but did not yield any benefits. We believe the following findings to be especially useful as further work can be developed around these trials and reduce the amount of overhead for attempting the optimizations we experimented with.

Unrolling. Our experimentation and setup turned out to come with its own set of complexities. We looked into various unrolling factors and step sizes, from 2, all the way up to 16. Furthermore, we implemented various styles for the unrolling, namely looping in increments of the step - e.g., 16 - and then having a single loop for the remaining elements, or looping with consecutive loops of different step sizes - e.g., 16, 8, 4 and 1. We call this consecutive step-decrease loops. From our runs, having consecutive step-decrease loops with different step sizes worked out best, and the step size we used was 16. One other option we looked into was using a padding larger than 4 such that the size of the matrix was a multiple of the step size. However, this was problematic, as multiple data structures had to be updated to also match this padding, which added quite a bit of redundant data and the runs took longer than our consecutive step-decrease loops.

Precomputing masks. We made another attempt to optimize the performance by precomputing two comparisons that were causing a bottleneck within the for loop of the walk function. Since the matrix A remains constant and does not change over time, precomputing these comparisons seemed like a valid optimization technique in this scenario. Following the precomputation of the comparison, we directly loaded the mask and applied it to the vectors. However, this optimization did not yield better results and, in fact, performed worse on some of our machines.

Upon further investigation, we realized that we were not actually eliminating an instruction within the critical loop. Instead, we were replacing a comparison instruction with a load instruction. As a result, the relative running time of the code depended on the specific hardware being used.

Random number generator. We tried generating number in different ways, such as using multiplications instead of modulo operations, or reinterpreting bits using union in C. This did not bring any gain in terms of performance or reduction of executed flops.

Compilers. We experimented with different compilers: GCC, Clang, and ICC - the Intel C compiler. Additionally, we also tried different compiler flags, such as running with O2 and selectively enabling flags that are part of O3 optimizations set. In the end, GCC offered the best results.

AVX 512 After comparing some AVX-512 imlementations with AVX-256 ones, we realized that the implementation using AVX-512 was a little faster or slower compared to the implementation using AVX with 256-bit vectors. The results were not consistent and unstable. Therefore, we cannot affirm that this optimization provided consistent improvements. The optimizations were tested on a real AVX-512 machine.

4. EXPERIMENTAL RESULTS

Experimental setup. We used the following hardware to test our optimizations.

Test machine. Most of the experiments were run on an Asus laptop with an Intel i7-10750H processor. This

processor has a Comet Lake microarchitecture, a 2.60 GHz base frequency, 384 KB of L1 cache, 1.5 MB of L2 cache, and 12MB of L3 cache. We used the GCC compiler, version 13.1.1. We player around with multiple flags, but we found that the following configuration gives the best results:

`-O3 -ffast-math -mfma -march=native`.

`AVX-512 machine`. Some AVX-512 experiments were run on a different machine that had support for it: a Dell laptop with an 11th Gen Intel(R) Core(TM) i9-11900H processor. This processor has a Cypress Cove microarchitecture, a 2.50 GHz base frequency, 384 KB of L1 cache, 10 MB of L2 cache, and 24MB of L3 cache. We used the GCC compiler, version 11.3.0. The flags used were the same as for the Asus laptop.

Input generation. For the inputs, we followed the PolyVest[1] approach of generating n-dimensional cube examples using half-planes defined in a ladder-like manner. For example, a 2-dimensional cube, better known as a rectangle, is generated as:

$$\begin{array}{lll} 4 & 2 & \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & -1 \\ 1 & -1 & 0 \end{array}$$

where the first value represents the number of lines, the second the number of columns, and the 4x2 matrix the values of the cube. We chose this representation because it allows the $m = 2n$ simplification, making the input-size 1-dimensional instead of 2-dimensional. We ran our experiments on sizes from 1 to 49, in steps of 2. For each input, we ran a warm-up round, then 3 actual rounds while counting the number of flops, taking the average as the result. We stopped at 49 because the baseline implementation was taking too long to run.

Basic optimizations vs. baseline. As it can be seen in Figure 2, our basic optimizations - as described in section 3 - provided a significant speedup for lower dimensions up to 7, reaching a peak of ∼2.25 at size 1. After size 7, the speedup steadily decreased almost reaching 1.0 at 49. As the working sizes are insignificant for the achieved speedup, this experiment helped us understand the need for more complex optimizations.

In terms of number of $\frac{flops}{cycle}$, the basic optimization gives at best an increase of 1.6x at the size 17 as it can be seen in Figure 3.

Vectorization vs. algorithmic optimization. In Figure 4 we plot the speedup difference of the vectorization version of our optimizations as compared to the algorithmic optimizations. We do not compare it to the baseline implementation as the complexity and number of flops is completely

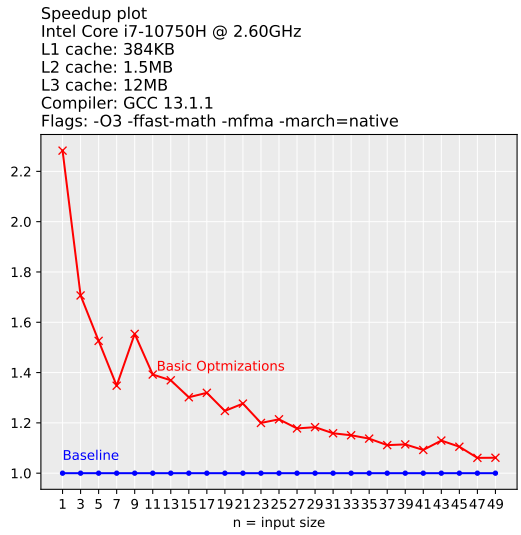

Fig. 2. Speedup plot of the Basic Optimizations, before doing the complexity reduction optimization

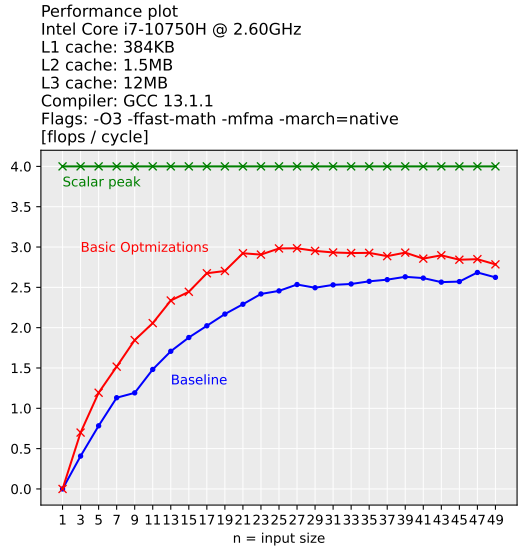

Fig. 3. Performance plot of the Basic Optimizations, before doing the complexity reduction optimization

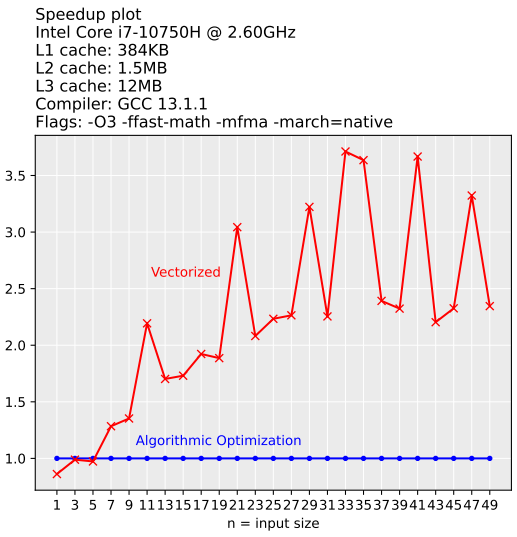

Fig. 4. Speedup plot of the vectorization optimizations relative to the complexity reduction optimization.

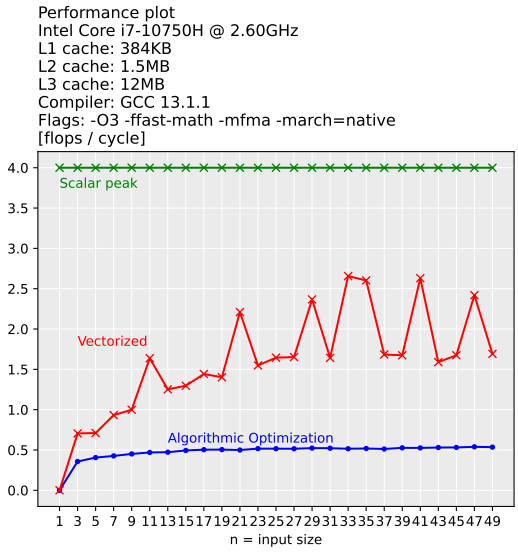

Fig. 5. Performance plot of the vectorization optimizations relative to the complexity reduction optimization.

.

different. As it was expected, vectorization brings up to a 3.5 speedup increase as the dimension increases. Numerous descends and ascends can be seen following the trend of the speedup which we explain by the decision to unroll the loops inside the `walk` function by the aforementioned decremental steps - 16, 8, 4, 1. The same explanation holds for the trend observed in Figure 6, where the $\frac{flops}{cycle}$ shows close to 5.2x increase for the vectorization version.

Roofline plot. Finally, the roofline plot generated by Intel Advisor is:

5. CONCLUSION

To start our optimization process, we began by creating a basic version of the algorithm using the C programming language. During optimizations, we considered the entire abstraction stack by examining the code at various levels of abstraction, from high-level algorithms to low-level implementation details. Even this baseline implementation proved to be faster than the PolyVest C++ implementation.

To achieve better performance, we made changes to the algorithm itself, aiming for improved asymptotic running time. We also employed various optimization techniques, such as scalar replacement and precomputing values, to make the code more efficient.

Subsequently, we delved into the realm of SIMD instructions, which brought other challenges that required code changes. For example, we used the `padding technique`, which involved modifying the critical loop to work with brancheless code, enabling SIMD instructions to be applied seamlessly and we also investigated different unrolling fac-

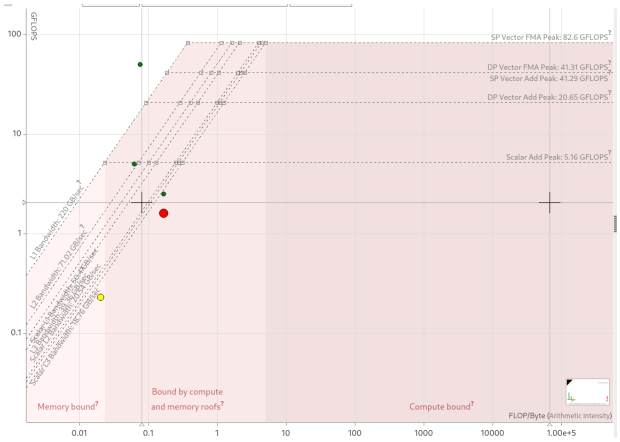

Fig. 6. Roofline plot of the final binary

.

tors to fit our use cases.

Finally, the optimizations we implemented resulted in a notable speedup of 7x for the scalar version of the code compared to our baseline implementation. Additionally, the SIMD version achieved a speedup of 22x compared to the same baseline.

6. CONTRIBUTIONS OF TEAM MEMBERS

This section details each of the team members' contribution to the project. We've all worked on the initial baseline and we've each iteratively added optimizations.

Constantin. Set up the skeleton of the repo, while making sure that the library dependencies can be installed by all members. Did constant profiling of our application using Intel VTune and Intel Advisor. Did some basic optimizations such as reducing the number of memory allocations and deletions, replacing common subexpressions. Played around with the Added the xoshiro random number generator to our code, and tried different ways of generating random floating point numbers from the random bits. Worked on the algorithmic optmization which reduced the complexity of the algorithm and the number of floating point operations as a consequence.

David. Implemented the `Walk` function for the baseline. Tried various math libraries to implement `Armadillo`'s [9] [10] operations efficiently in our C baseline and worked on our math wrappers around `Lapack`[11]. Reduced cycle count by using scalar replacement across data structure accesses using pointers and references; managed to work around rounding of polytope sizes and aligning the data structures in memory to enforce contiguous memory accesses that improved our caching and SIMD utilization (by removing `__mm256_loadu` and `__mm256_storeu` instructions). Worked on unrolling the loops in `Walk` while also experimenting with SIMD instruction reduction, although this turned out to be unsuccessful.

François. Created wrappers for the linear algebra operations. Wrote initial optimizations to improve the speed of the code, such as precomputing values, replacing scalar values with more efficient alternatives, and implementing basic optimizations. My primary focus was on the `SIMD` work, particularly within the walk function, which was the bottleneck. After careful analysis, I devised a method to apply `SIMD` (explained using padding technique). Additionally, I incorporated `AVX-512` and made further optimizations to the `SIMD` implementation, such as replacing division operations with multiplication for better performance.

Roxana. Wrote the `EstimateVolume` function for the baseline. Optimized the code in `Preprocess` by replacing BLAS calls with double-loops and checking runtime for each variant. Statistical results (checking our implementation against PolyVest for correctness). SIMD optimization in `EstimateVolume`. Helped Constantin with running software for flop count. Tried to pre-compute separate the positive and negative elements of array `bounds` in `Walk` so that we don't have to use `blendv` (ultimately, didn't reach a solution with a better performance).

7. REFERENCES

[1] Cunjing Ge, Feifei Ma, and Jian Zhang, "A fast and practical method to estimate volumes of convex polytopes," 2013.

[2] M. E. Dyer and A. M. Frieze, "On the complexity of computing the volume of a polyhedron," *SIAM Journal on Computing*, vol. 17, no. 5, pp. 967–974, 1988.

[3] Benno Büeler, Andreas Enge, and Komei Fukuda, *Exact Volume Computation for Polytopes: A Practical Study*, vol. 29, pp. 131–154, 07 2011.

[4] M. Dyer and A. Frieze, "A random polynomial time algorithm for approximating the volume of convex bodies," in *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, New York, NY, USA, 1989, STOC '89, p. 375–381, Association for Computing Machinery.

[5] Robert L. Smith, "Efficient monte carlo procedures for generating points uniformly distributed over bounded regions," *Oper. Res.*, vol. 32, no. 6, pp. 1296–1308, dec 1984.

[6] A. MAKHORIN, "Glpk (gnu linear programming kit)," *http://www.gnu.org/s/glpk/glpk.html*, 2008.

[7] Donald Goldfarb, *On the Complexity of the Simplex Method*, pp. 25–38, Springer Netherlands, Dordrecht, 1994.

[8] "xoshiro/xoroshiro generators and the prng shootout," https://prng.di.unimi.it/.

[9] Conrad Sanderson and Ryan Curtin, "Armadillo: a template-based c++ library for linear algebra," *Journal of Open Source Software*, vol. 1, no. 2, pp. 26, 2016.

[10] Conrad Sanderson and Ryan Curtin, "A user-friendly hybrid sparse matrix class in c," in *Mathematical Software – ICMS 2018*, pp. 422–430. Springer International Publishing, 2018.

[11] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.