EFFICIENT VOLUME ESTIMATION OF CONVEX POLYTOPES IN HIGH DIMENSIONS

Samuel Kiegeland, S Deepak Narayanan, Vraj Patel, Raghu Raman Ravi

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Estimating the volume of high-dimensional convex polytopes poses significant challenges, impacting numerous disciplines like computational geometry, machine learning, and optimization theory. Exact methods like VINCI [1, 2] are constrained to polytopes with lower dimensions. In contrast, PolyVest [3], an approximation method, extends usability to larger dimensions using the Multiphase Monte-Carlo algorithm and the coordinate directions hit-and-run random walk. However, PolyVest's effectiveness is hindered due to the computational overhead of the repeated random walk. We propose an approach that boosts the efficiency of high-dimensional volume approximation by tackling the performance issues linked to the random walk process in PolyVest, using strategic precomputation and vectorization. Through extensive experimentation, we demonstrate that our method can be up to $\sim 115\times$ faster than PolyVest, especially for large dimensions. Our method provides a practical and streamlined alternative for estimating the volumes of high-dimensional convex polytopes.

1. INTRODUCTION

The problem of efficiently estimating the volume of convex polytopes in high-dimensions has long been challenging in computational geometry. It is essential to many disciplines, such as optimization, machine learning, and statistical physics. A convex polytope Q can be defined in the hyperplane representation as the set of all points x that satisfy the inequality $Ax \leq b$, where A is a real-valued matrix in $\mathbb{R}^{m\times n}$, and b is a real-valued vector in $\mathbb{R}^m$. In this representation, n represents the dimensionality of the polytope, while m signifies the number of hyperplanes.

Exact methods have been used to compute volumes, but these are limited to polytopes of smaller dimensions (< 20) [2] due to computational constraints. For larger dimensions, approximation methods have been devised. A prominent example is PolyVest [3], which provides a way to estimate the volumes of convex polytopes in higher dimensions. However, as the dimensionality grows, PolyVest becomes increasingly infeasible. This is because its runtime increases with the dimensionality and the number of hyperplanes. One of the culprits behind this inefficiency is the coordinate directions hit-and-run random walk method used in PolyVest for bound computation, which has a time complexity of $\mathcal{O}(m + mn)$.

Related work. Our work builds upon a rich body of prior work. Previous studies [4, 5] have shown that exact volume computation is #P-hard. Since then, various methods have been proposed to approximate the convex polytopic volumes. One notable contribution is the VINCI [1, 2] method, an exact volume computation tool limited to dimensions up to fifteen due to computational limitations. Our work builds on PolyVest [3], which significantly contributes to the field due to its ability to estimate volumes of high-dimensional polytopes. PolyVest uses the Multiphase Monte Carlo algorithm and the coordinate directions hit-and-run random walk. The algorithm improved efficiency by reutilizing sample points. Despite its effectiveness, it suffers from a high runtime, particularly with increasing dimensions.

Contribution. In this work, we propose an efficient method for the volume estimation of convex polytopes in high dimensions, aiming to address the limitations of PolyVest. Our approach mainly targets the random walk, which we find to be the computation bottleneck. Alongside precomputations and vectorization, our method improves the time complexity of the bound computation from $\mathcal{O}(m + mn)$ to $\mathcal{O}(m)$, allowing for a much more efficient approximation of volumes in higher dimensions. This contribution offers a substantial reduction in computational complexity and hence opens the door to feasible and efficient volume estimation of convex polytopes in high dimensions, an area previously hindered by the infeasibility of existing techniques.

2. BACKGROUND ON THE ALGORITHM

As mentioned in Section 1, our primary focus is on Polyvest, presented in [3]. Below, we briefly describe the setup and present the algorithm.

2.1. Algorithm in PolyVest

Recalling the setup, we are given a convex polytope Q in n dimensions, described by all the points $x \in \mathbb{R}^n$ satisfying the linear inequality $Ax \leq b$, where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. Our goal is to estimate the volume of Q. We use $\texttt{vol}(A)$ to denote the volume of a convex polytope A, and $B(u, R)$ to denote the ball centered at $u \in \mathbb{R}^n$ of radius R. The algorithm proceeds in the following steps.

Preprocessing. In this step an affine transformation T is applied to the polytope Q to ensure that $B(0,1) \subseteq P = T(Q) \subseteq B(0,r)$, where P is the transformed polytope, and $r = 2n$. Using the Shallow-β-Cut Ellipsoid Method the transformation T is found in polynomial time [6]. This transformation changes the volume of polytope Q. However, the transformation T allows the computation of the ratio $\gamma := \frac{\texttt{vol}(Q)}{\texttt{vol}(P)}$, directly, without requiring the volumes of the individual polytopes. This results in a simplification of the problem, which boils down to now computing the volume of the transformed polytope P, and then computing $\texttt{vol}(Q) = \gamma \times \texttt{vol}(P)$.

Subdivision. Now consider a series of concentric balls $B_i := \{B(0, r_i)\}_{i=0}^{\ell}$, where $\ell := n \log_2 r$ and $r_i = 2^{i/n}$. Now consider $K_i := B_i \cap P$. Observe that $K_0 = B_0$ and $K_\ell = P$. We then have

$$\texttt{vol}(P) = \texttt{vol}(B_0) \times \prod_{i=0}^{\ell-1} \frac{\texttt{vol}(K_{i+1})}{\texttt{vol}(K_i)}$$

The problem has been further simplifed by subdividing $\texttt{vol(P)}$ into the volume of the unit ball and the ratio of the volumes of the convex polytopes $K_i, K_{i+1}\ \forall i \in \{0\} \cup [\ell - 1]$. Exact computation of the volume of unit ball can be done recursively. The technical challenge is in estimating the volume of the latter. Defining $\alpha_i := \frac{\texttt{vol}(K_{i+1})}{\texttt{vol}(K_i)}$, the Multi-Phase Monte Carlo method is thus used to *estimate* $\alpha_i, \forall i \in \{0\} \cup [\ell - 1]$.

Hit-and-run Random Walk. To approximate α_i, p points are first generated in K_{i+1}. Counting the number of points $c_i \in K_i$, $\alpha_i \approx \frac{p}{c_i}$. The challenge is in generating the p points uniformly in the polytopes K_i. In PolyVest [3] this is addressed using a hit-and-run random walk [7, 8, 9] - in particular the coordinate directions method which is guaranteed to generate a sequence of points whose limiting distribution is the uniform distribution [7]. In classic Multiphase Monte-Carlo methods the ratios α_i's are estimated in the natural order - from α_0 up to $\alpha_{\ell-1}$. In PolyVest [3] however, it is done in the reverse order and is a key novelty - as it allows *reutilization* of previously sampled points, resulting in significant computation reduction.
In summary, the volume of polytope Q is estimated as:

$$\texttt{vol}(Q) = \gamma \times \texttt{vol}(B_0) \times \prod_{i=0}^{\ell-1} \alpha_i$$

where we approximate α_i's using random walk in the reverse order i.e. from $\alpha_{\ell-1}$ to α_0. The pseudocode for the algorithm, following the one given in [3], is described in Algorithm 1 and 2. We generate `stepsize` many samples for estimating the α_i's in Algorithm 1.

Algorithm 1: `estimateVol`

$\mathbf{x} \leftarrow 0$
$t \leftarrow [0, 0, \ldots, 0]$, `count` $\leftarrow 0$
$\ell \leftarrow \lceil n \log_2(2n) \rceil$, `stepsize` $\leftarrow 1600\ell$
for $k \leftarrow \ell - 1, \ldots, 0$ **do**
  for $i \leftarrow$ *count*, *stepsize* **do**
    $\mathbf{x} \leftarrow$ `walk`$(\mathbf{x}, k)$
    if $\boldsymbol{x} \in B_0$ **then**
      $t_0 \leftarrow t_0 + 1$
    end
    else if $\boldsymbol{x} \in B_k$ **then**
      $m \leftarrow \lceil n \log_2 \|\mathbf{x}\| \rceil$
      $t_m \leftarrow t_m + 1$
    end
  end
  `count` $\leftarrow \sum_{i=0}^{k} t_i$
  $\alpha_k \leftarrow$ `stepsize`/`count`
  $\mathbf{x} \leftarrow 2^{-1/n}\mathbf{x}$
end
return $\gamma \times \texttt{vol}(B_0) \times \prod_{i=0}^{\ell-1} \alpha_i$

Algorithm 2: `walk`

$d \leftarrow$ RANDOM$(\{1, 2, 3, \ldots, n\})$
$C \leftarrow$ `norm_sqr(`$\mathbf{x}$`)` - x_d^2
$r \leftarrow \sqrt{R_{k+1}^2 - C}$
MAX $\leftarrow r - x_d$
MIN $\leftarrow -r - x_d$
`bound` $\leftarrow B_d - (A \,/\, (A_{:,d}.\mathbf{1}^T)).x$
  `// Bottleneck - ` $\mathcal{O}(m + mn)$ `time`
for $i \leftarrow 1, 2, \ldots, m$ **do**
  if $A_{i,d} > 0$ ***and*** $bound_i <$ MAX **then**
    MAX = $bound_i$
  end
  else if $A_{i,d} < 0$ ***and*** $bound_i >$ MIN **then**
    MIN = $bound_i$
  end
end
$x_d \leftarrow x_d +$ RANDOM(MIN, MAX)
return x

2.2. Implementation

Our implementation is written in C++. We have Armadillo (`10.8.2`) and GLPK (`GLPK LP/MIP Solver 5.0`) as external dependencies. Our implementation of the algorithm described above consists of three key functions - `preprocess`, `estimateVol` and `walk`. `preprocess` consists of computing the transformation T and subsequently the ratio γ. This step involves the invocation of the Shallow-β-Cut Ellipsoid Method. In the `walk` function we have the coordinates direction hit-and-run random walk method which is invoked many times, and has a linear dependency (cf. Algorithm 1) - the next call of the function depends on the operations done in the previous call. The function `walk` is invoked inside `estimateVol`.

2.3. Cost Analysis

For cost measure, we consider floating point additions, subtractions, multiplications and divisions. As already mentioned, the preprocessing step is polynomial time but the exact runtime computation is a bit involved, and we therefore do not report it [6]. Of particular interest therefore are the `estimateVol` and `walk` functions. In particular, the `walk` function is invoked at most $1600(n\log_2(2n))^2$ times in `estimateVol` which is the bulk of the computation. We do not know the exact number of invocations because of the random walk procedure (see Algorithm 1). For each invocation of the `walk` function there is a total of $(2mn+m+2n+\mathcal{O}(1)) = \mathcal{O}(mn)$ flops. Apart from `walk`, there is at most $\mathcal{O}(n^2\log^2 n)$ flops. In terms of asymptotic complexity, the number of flops of the algorithm (excluding the preprocessing step) is, therefore,

$$\mathcal{O}(mn^3(\log^2 n)) = \tilde{\mathcal{O}}(mn^3)$$

where $\tilde{\mathcal{O}}$ subsumes polylog factors. As we can observe, we have a cubic dependence between the number of flops and the dimension. As we will describe in the sections below, we, in fact, improve this asymptotic flop count by shaving off a factor of n, improving the asymptotic flop count to $\tilde{\mathcal{O}}(mn^2)$.

3. OPTIMIZATIONS

We implement a baseline by following the algorithm in the paper as is, without introducing any optimizations. We then utilize Intel®'s VTune Profiler tool for primary hotspot analysis to analyze the bottlenecks after each optimization step for our program. Based on the initial hotspot analysis, we spend 98.5% of the time in `walk()`, so we focus our optimization on this function. We detail our step-wise optimization process below.

3.1. Optimization - I

Initially, we focus on doing some standard optimizations.

Volume of Unit Ball. In our baseline, we use a recursive function to compute the volume of a unit ball of n dimensions. We observe that it could be made into an iterative algorithm and implement a version without the overhead incurred by recursive function calls.

Loop Unrolling. We unroll loops in `estimatevol()` and `norm_sqr()` by a factor of 4 (adding the appropriate number of accumulators). We expect this to improve the instruction level parallelism of the program and hence improve performance.

Xoshiro Random Number Generator [1]**.** The Xoshiro Random number generator is a custom random number generator that is more efficient than the default `rand()` function in C/C++. We utilize this for generating the random numbers required in `walk()`.

Bound Checks. The Armadillo vectors check the bounds for every access to its elements. We remove these bound checks after ensuring the correctness of the program.

Precomputation of division operations. This is by far the most significant optimization in this phase. There are certain floating point division operations in `walk()`, which we establish is the critical bottleneck of the entire program. Hence, we precompute these operations and store them in an array of matrices, which we then pass as arguments to `walk()`.

The division operation in line 6 of Algorithm 2 can be pre-computed and the results can be stored in an array of matrices, $\tilde{A}$. Thus, we get Algorithm 3 which has the same asymptotic time complexity, but with improved constants.

Algorithm 3: Fast Walk I: After pre-computation

1 $d \leftarrow$ RANDOM($\{1, 2, 3, \ldots, n\}$)
2 $C \leftarrow$ `norm_sqr(`$\mathbf{x}$`)`
3 $\mathcal{O}(1)$ computation on C
4 `bound` $\leftarrow B_d - \tilde{A}_d.x$ `// Bottleneck - `$\mathcal{O}(m + mn)$ `time`
5 $\mathcal{O}(m)$ computation for MIN and MAX
6 $x_d \leftarrow x_d$ + RANDOM(MIN, MAX) `// Update x in the chosen direction`

3.2. Optimization - II

After profiling, we find that our bottleneck is still in `walk()`, and hence we continue to focus our efforts here and perform some significant optimizations in this region.

Asymptotically Faster Algorithm. At this point, we find that we can modify the algorithm to be asymptotically

[1] This was taken from the following GitHub repository: https://github.com/Reputeless/Xoshiro-cpp

faster by maintaining the value of $A.x$ in a variable, in between calls to `walk()`. The key reason why such an optimization is feasible is because the vector x is updated in precisely one coordinate direction, which makes an update to Ax cheap ($\mathcal{O}(m)$ time). This allows us to compute `bound` in $\mathcal{O}(m)$ instead of $\mathcal{O}(m + mn)$ time, making `walk()` asymptotically faster (now $\mathcal{O}(m)$ time as well). This gives us significant speedup especially for large values of n.

Algorithm 4 illustrates the modified walk function where we store the value of $A.x$ in the variable Ax. We maintain this variable appropriately and update it at the end of `walk()`.

Algorithm 4: Fast Walk II: Asymptotically Fast

1 $d \leftarrow$RANDOM($\{1, 2, 3, \ldots, n\}$)
2 $C \leftarrow$ `norm_sqr(`$\mathbf{x}$`)`
3 $\mathcal{O}(1)$ computation on C
4 `bound` $\leftarrow B_d - Ax/A_{:,d}$ `// Now -` $\mathcal{O}(m)$
5 $\mathcal{O}(m)$ computation for MIN and MAX
6 $v \leftarrow$ RANDOM(MIN, MAX)
7 $x_d \leftarrow x_d + v$ `// Update x in chosen direction`
8 $Ax \leftarrow Ax + vA_{:,d}$ `// Update Ax -` $\mathcal{O}(m)$

3.3. Optimization - III

When we look at the performance for Optimization-II (see figure 2), comparing it to the previous optimization and Polyvest, we notice a degradation. Therefore, we now focus our efforts in improving performance in critical regions of the program, primarily through vectorization.

Vectorization. We find many computations that can be vectorized using AVX 256 instructions:

- The only non-trivial vectorization is the computation of the variables max and min in `walk()`. Here, we use vectorized comparisons and then use vectorized `blendv` operations in place of the if-else-construct. Additionally, we use vectorized load, store, min and max operations.
- The calculation of the squared norm of x in `norm_sqr()` is a simple reduction of x and can be done with vectorized loads, stores and FMAs.
- In `estimateVol()`, the updates of the variables x and Ax is straightforward to vectorize and can be done using vectorized loads, stores, multiplications and FMAs.
- The computation of x and the update of Ax in `walk()` is also straightforward to vectorize and can be done using vectorized loads, stores and FMAs.

Precomputation. The aforementioned comparison masks used in `walk()` only depend on the coordinate chosen for that particular execution of `walk()`. Thus, we precompute these masks for every coordinate direction in `estimateVol()`, and pass them as parameters.

3.4. Optimization - IV

We conduct another hotspot analysis using VTune. As shown in table 1, we spend the majority of the time on a function called `__posix_memalign`. This function is called by Armadillo in each call to `walk` for computing *bound* to unnecessarily allocate memory. Due to this significant overhead, we focus our subsequent optimization on removing the Armadillo objects.

Function	Time	% Time
`__posix_memalign`	98.187s	44.2%
`loop in polytope::walk`	28.207s	12.7%
`__memcpy_avx_unaligned_erms`	26.534s	11.9%

Table 1. Hotspot analysis before Optimization - IV. As can be observed, alignment of objects in `__posix_memalign` takes up significant amount of time

Armadillo objects. We completely remove the Armadillo objects from our project except for those present in `preprocess()`, and replace them with arrays represented as simple C pointers. This also lead to the use of simple pointer arithmetic instead of member function calls. We store matrices in column major format because iterating over entries of a column is the dominant access pattern in our code.

Alignment. We align all the arrays to 32 bytes so that we can used aligned vector operations. We also expect better cache access pattern and fewer misses after this modification.

Reduced precision. We reduce the precision of the variables from double precision to single precision (float) in the performance-critical parts of the program like `walk()` and `estimatevol()`. Since we use AVX 256, instead of performing operations on 4 doubles at once, we can do them on 8 floats at once. We can thus expect an almost two-fold improvement in performance.

As shown in Table 2 we remove the significant overhead caused by `__posix_memalign` and re-establish the loops in `walk()` as our main bottleneck. Further, as Table 3 demonstrates, after the alignment stage of this optimization phase, we are able to observe reduced cache misses, when compared post Optimization - III.

Function	Time	% Time
`loop in polytope::walk`	40.680s	34.8%
`__mm256_blendv_pd`	12.787s	10.9%
`__mm256_store_pd`	7.89s	6.8%

Table 2. Hotspot analysis after Optimization - IV. We observe that our optimizations have gotten rid of the computational overhead caused by `__posix_memalign`.

Test	Cache Accesses	Cache Misses	% Misses
After Optimization - III			
cube 40	69'824'799	2'650'001	2.973
cube 50	2'184'968'670	8'213'628	0.536
cube 60	6'000'767'929	13'920'658	0.250
cube 70	16'420'375'155	43'699'435	0.300
cube 80	29'597'701'068	29'467'155	0.097
After alignment in Optimization - IV			
cube 40	167'561'229	3'301'000	1.890
cube 50	1'212'823'147	4'757'832	0.258
cube 60	5'046'025'601	4'744'181	0.094
cube 70	13'852'327'880	8'514'621	0.060
cube 80	29'191'158'802	14'418'078	0.049

Table 3. Cache accesses and misses after Optimization - III and after alignment stage of Optimization - IV. We observe that our optimizations have reduced the % of cache misses.

3.5. Optimization - V

We wrap up with some additional optimizations.

Norm computation. We observe that there is no need to compute the squared norm of x for every single iteration of `walk()`. This is again due to the fact that x changes only in one coordinate. Let us assume we change x in the coordinate i and the new value is x' respectively. Then,

$$\texttt{norm\_sqr(x')} = \texttt{norm\_sqr(x)} - x_i^2 + x_i'^2$$

We can now simply maintain the value of `norm_sqr(x)` in a variable and update it during every call of `walk()`. We also use $\log_2 \|\mathbf{x}\|$ in `estimateVol()`, which can just be re-written as $0.5 \cdot \log_2 \|\mathbf{x}\|^2$ and now computed using the maintained value.

Merging vectorized loops. We observe that we can merge the loops that compute `bound` and MIN/MAX in `walk()`, resulting in the elimination of loads and stores to the `bound` array.

Hint input sizes to compiler. We refactor our program so that it can now be compiled to work only for a specific value of m and n (using macro definitions) so that the compiler can perform more informed unrolling and code elimination.

3.6. Other Optimizations

Below we discuss additional optimizations that we explored. However, these did not produce any conclusive improvement in terms of performance or speedups.

Single compilation unit. We merge all the separate `.cpp` files into a single file and then compile it. We observe that while this led to reduced compile times, there was no noticeable effect with regard to performance and speedups.

Profile guided optimizations. We utilize the profile-guided optimization functionality in the compiler which produces a new executable by profiling the performance of the original executable on specified input instances. This method also did not produce any statistically significant improvement in terms of speedups, and performance.

Clang. We compile the project with `Clang 14.0.0` instead of `GCC 11.3.0`, using the same compiler flags. The executable was worse than the original executable compiled using GCC in terms of both performance and speedups.

4. EXPERIMENTAL RESULTS

In this section, we describe the range of experiments we conduct. We begin by offering a detailed overview of our experimental framework, succeeding by examining our implementations' validity and accuracy. Next, we test the efficiency of our enhancements on varying-sized cubes and compare these results to those of PolyVest [3]. We perform a roofline analysis on our final optimization and delve into evaluating the peak performance percentage for our starting point and ultimate enhancements. We quantify the speedup ratio of all our optimization steps relative to our initial setup and PolyVest. In the end, we compare our final optimization with PolyVest's results in the context of crosses, simplices, and random hyperplanes, ensuring the versatility and robustness of our enhancements when applied to a variety of convex polytopes.

Experimental Setup. We run all of our experiments on an Intel®Core™ i7-10610U processor using the Comet Lake architecture, with a base frequency of 1.80 GHz. The cache sizes for L1, L2 and L3, are 256KB, 1MB and 8MB respectively. We disable Intel® Turbo Boost. We compile our code with `GCC 11.3.0` and use the following optimization flags: `-O3`, `-march=native`, `-ffast-math`.

Datasets. We test and benchmark our program on several classes of polytopes of varying sizes. `cube_n` denotes the axis aligned hypercube in n dimensions centered at origin and having a side length of 2. Thus, its volume is given by 2^n. `cuboid_n` is obtained from `cube_n` by stretching

it along a random coordinate axis by a factor of 50. The volume of such a polytope is $50 \cdot 2^n$. `simplex_n` denotes a simplex in n dimensions that is defined by the origin and the n unit basis vectors in the positive direction of all the coordinate axes. The volume of such a simplex is given by $\frac{1}{n!}$. `cross_n` denotes a cross polytope centered at origin with vertices given by all the unit basis vectors. Its volume is given by $\frac{2^n}{n!}$. `random_n_m` denotes a random hyperplane in n dimensions given by m uniformly random hyperplanes tangent to the unit hypersphere.

Correctness. Before analyzing the performance of our optimizations, we ensure the correctness of our implementation. To do this, we estimate the volumes of different polytopes 100 times using both PolyVest and our final optimization (Opt-V). To report our results, we opt for a similar analysis as [3] and report the average volume $\hat{v}$, standard deviation σ, the 95% confidence interval, and the error $\epsilon = \frac{q-p}{\hat{v}}$. Table 4 shows that our final optimization leads to comparable estimates across all polytopes. Due to the similarity of our estimations and PolyVest, we refer to the comparison with ground truth volume computations in [3], who compute the exact volume for polytopes with small dimension using VINCI [1, 2].

Test	Avg. vol $\hat{v}$	Std Dev σ	95% CI $\mathcal{I} = [p, q]$	Error $\epsilon = \frac{q-p}{\hat{v}}$
		PolyVest		
cube_10	1016.68	46.04	[926.44, 1106.92]	0.18
cube_20	1.05e+06	47980.94	[9.55e+05, 1.14e+06]	0.18
cube_30	1.07e+09	4.98e+07	[9.69e+08, 1.16e+09]	0.18
cube_40	1.10e+12	5.40e+10	[9.93e+11, 1.20e+12]	0.19
cuboid_10	51051.27	1384.83	[48337.05, 53765.49]	0.11
cuboid_20	5.24e+07	1.75e+06	[4.89e+07, 5.58e+07]	0.13
cuboid_30	5.36e+10	1.81e+09	[5.00e+10, 5.71e+10]	0.13
cross_5	0.27	4.54e-03	[0.26, 0.28]	0.07
cross_8	6.33e-03	1.60e-04	[6.01e-03, 6.64e-03]	0.10
cross_10	2.82e-04	6.35e-06	[2.70e-04, 2.95e-04]	0.09
simplex_10	2.76e-07	9.62e-09	[2.57e-07, 2.95e-07]	0.14
simplex_20	4.09e-19	1.73e-20	[3.75e-19, 4.43e-19]	0.17
simplex_30	3.79e-33	1.48e-34	[3.50e-33, 4.08e-33]	0.15
		Opt-V		
cube_10	1031.33	40.81	[951.34, 1111.31]	0.16
cube_20	1.05e+06	46366.24	[9.56e+05, 1.14e+06]	0.17
cube_30	1.07e+09	5.08e+07	[9.71e+08, 1.17e+09]	0.19
cube_40	1.10e+12	5.59e+10	[9.93e+11, 1.21e+12]	0.20
cuboid_10	51155.79	1530.94	[48155.20, 54156.39]	0.12
cuboid_15	1.64e+06	49012.89	[1.54e+06, 1.73e+06]	0.12
cuboid_20	5.23e+07	2.07e+06	[4.82e+07, 5.63e+07]	0.16
cuboid_30	5.35e+10	2.03e+09	[4.96e+10, 5.75e+10]	0.15
cross_5	0.27	5.73e-03	[0.26, 0.28]	0.08
cross_8	6.41e-03	1.63e-04	[6.09e-03, 6.73e-03]	0.10
cross_10	2.82e-04	7.95e-06	[2.66e-04, 2.97e-04]	0.11
simplex_10	2.76e-07	1.09e-08	[2.55e-07, 2.98e-07]	0.15
simplex_20	4.10e-19	1.68e-20	[3.77e-19, 4.43e-19]	0.16
simplex_30	3.77e-33	1.67e-34	[3.44e-33, 4.09e-33]	0.17

Table 4. Correctness and accuracy for PolyVest and our final optimization (Opt-V). We achieve comparable values for all estimations across different polytopes.

Performance. We measure the performance of our implementations using Linux perf. For each test case, we run five repetitions and report the average flop count per cycle using the command `perf stat -M FLOPc -r 5 ./polyvol tests/cube_80`. Figure 1 shows the performance of our baseline, PolyVest, and Optimization - I from Section 3. Although the initial performance of our base model falls short when juxtaposed with PolyVest, the first round of optimizations already brings our performance up to the level of PolyVest. Since Optimization - II changed

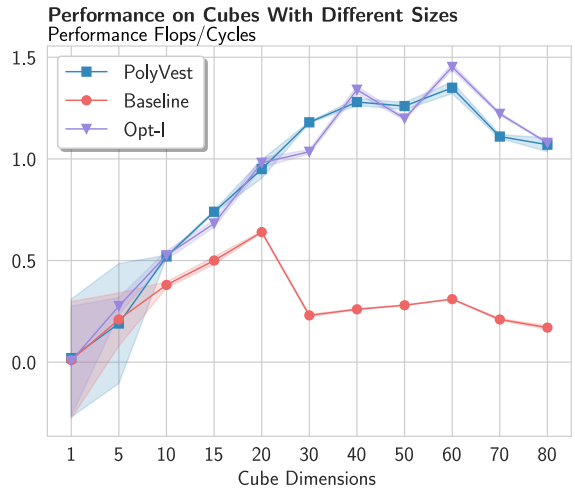

Fig. 1. Performance of our Baseline, PolyVest, and initial Optimization (Opt-I) measured with Linux perf. While our baseline performs increasingly worse with growing dimension, Opt-I achieves a performance comparable to PolyVest

our main algorithm, we compare all subsequent optimizations separately. Figure 2 shows the performance of all following optimization steps. We can see that particularly Optimization - IV and Optimization - V result in a substantial performance increase, leading to a final performance of $\sim$ 3.5-4 Flops per cycle for higher-dimensional polytopes.

Performance on Higher Dimensions. Due to the strong performance of Opt-V, we investigate its performance on cubes of higher dimensions up to 200. The result in Figure 3 shows that the performance drops with increasing cube dimensions which we investigate in the following section. However, we retain a performance greater than 3.5 flops per cycle for the larger dimensions. To illustrate the extent of speedup, we note that Opt-V requires $\sim$ 700 seconds to estimate the volume on a cube of dimension 200, which is more than $4\times$ faster than the time taken by PolyVest on cube 80 ($\sim$ 3000 seconds).

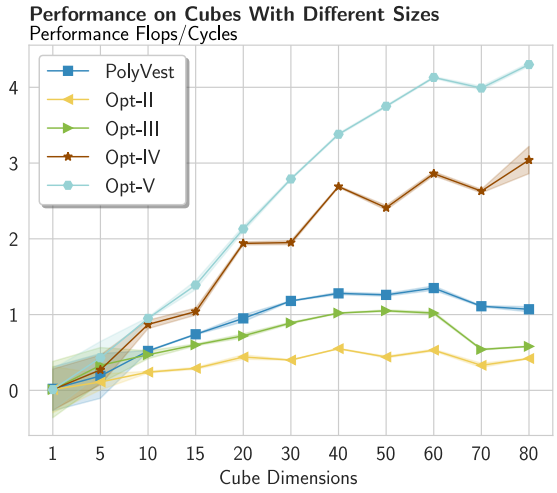

Fig. 2. Performance of PolyVest, and all major optimization from Opt-II measured with Linux perf. Opt-V outperforms PolyVest and all previous optimizations achieving a performance of more than 4 Flops per cycle.

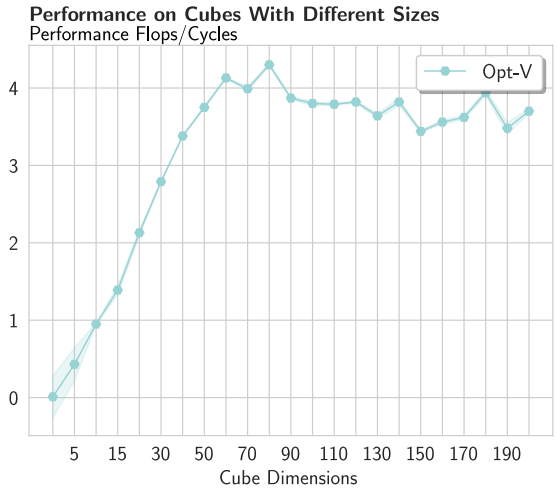

Fig. 3. Performance of Opt-V with cube dimensions up until 200. While the performance drops with increasing cube dimensions, we retain a performance greater than 3.5 flops per cycle.

Roofline. Following our results from Figure 3, we investigate why the performance of Opt-V drops for cubes of dimensions 90 and larger. We conduct a roofline analysis using the Intel® Advisor and visualize the hotspot of our implementation, which again is `walk()`. As shown in Figure 4, cube 90 is precisely the point at which our program becomes memory bound with respect to all cache levels. We verify that we remain memory-bound by computing the roofline for cube 150.

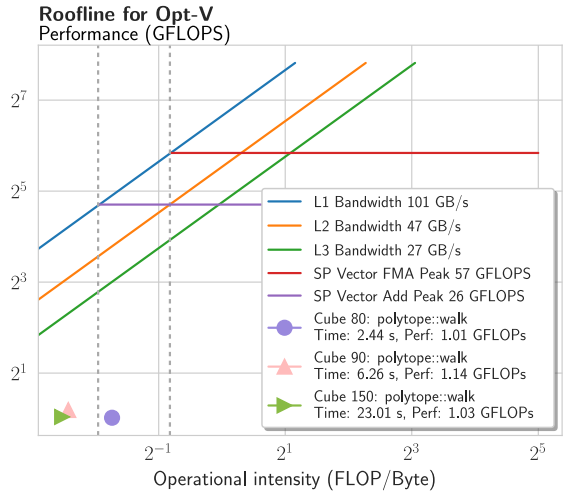

Fig. 4. Roofline for Opt-V measured on cube 80, cube 90 and cube 150. With cube 90 our implementation becomes memory bound, explaining the drop in performance we observe in Figure 2.

Peak Performance. We analyze the peak performance achieved by our baseline and final optimization using Intel® Advisor. Since the baseline uses double-precision floating-point numbers, we compare it against the double-precision roofline, while we compare our final optimization (Opt-V) against the single-precision roofline. The results in Table 5 show that our optimizations led us from 0.62% peak performance (baseline) to 11.73% peak performance (Opt-V), resulting in over 18× improvement in peak performance.

Method	Precision	GFLOPS	% Peak
Baseline	DP	0.175	0.62
Opt-V	SP	6.703	11.73

Table 5. Peak Performance for our baseline and Opt-V. Our optimizations lead to over 18× improvement in peak performance of Opt-V (11.73% Peak) over our baseline (0.62% Peak)

Speedup. We compare the runtime of all our optimizations against our baseline and PolyVest. The respective speedups over our baseline are shown in Figure 5. Our experiments show that our final method, marked as "Opt-V", achieves remarkable speedups. Compared to our baseline implementation, it registers a speedup of up to 700 times, with the speedups becoming increasingly pronounced for larger dimensions.

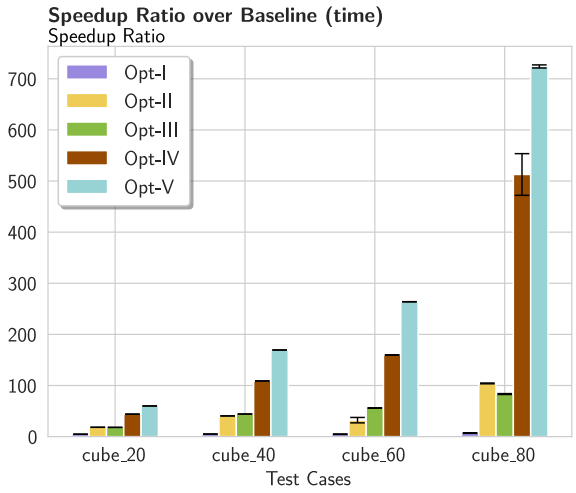

Fig. 5. Speedup of all major optimization steps over our initial baseline. Our final optimization (Opt-V) achieves a speedup of more than 700 over the baseline.

The speedups of our optimization with respect to PolyVest are depicted in Figure 6. We can see that all optimizations register a speedup, increasing with the number of cube dimensions. Our final optimization "Opt-V" achieves a speedup of up to 115 times when tested on a cube with 80 dimensions.

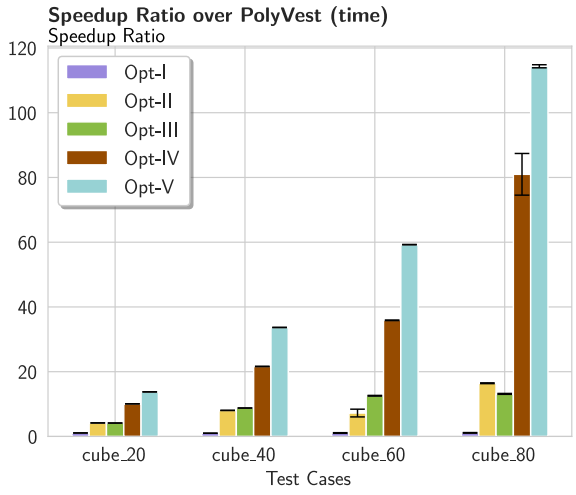

Fig. 6. Speedup of all major optimizations over PolyVest [3]. Our final optimization (Opt-V) achieves a speedup of more than 115 over the PolyVest.

Other Polytopes. To ensure that our optimizations perform well across different types of convex polytopes, we test on other polytopes, such as cuboids, simplices, crosses, and random hyperplanes. As shown in Table 6, Opt-V achieves higher performance and faster runtime across all polytopes, with increasing performance and speedup for polytopes with higher dimensions.

	Opt-V		PolyVest		Opt-V
	FLOPc	Time	FLOPc	Time	Speedup
cuboid_5	0.17	0.01	0.05	0.03	3.00
cuboid_10	1.20	0.04	0.40	0.28	7.00
cuboid_20	1.67	0.23	1.05	2.37	10.30
cuboid_30	2.50	0.72	1.19	13.39	18.60
cuboid_40	3.00	1.68	1.33	44.71	26.61
simplex_5	0.26	0.01	0.12	0.04	4.00
simplex_10	0.56	0.05	0.45	0.23	4.60
simplex_20	1.09	0.31	0.89	2.32	7.48
simplex_30	1.35	1.04	1.07	11.21	10.78
simplex_40	2.06	1.89	1.27	32.65	17.28
simplex_50	2.14	4.03	1.30	85.45	21.20
simplex_60	2.26	7.31	1.38	177.79	24.32
simplex_70	2.23	13.01	1.38	350.08	26.91
simplex_80	2.33	20.97	1.43	608.85	29.03
cross_5	0.15	0.01	0.25	0.06	6.00
cross_6	2.27	0.02	0.43	0.14	7.00
cross_7	4.23	0.03	0.43	0.46	15.33
cross_8	3.66	0.06	0.88	0.70	11.67
cross_9	3.58	0.13	0.98	1.78	13.69
cross_10	4.63	0.30	1.01	4.93	16.43
cross_13	3.30	6.01	1.07	93.22	15.51
rh_5_12	0.39	0.02	0.11	0.03	1.50
rh_15_50	1.61	0.18	0.90	1.72	9.56
rh_20_50	1.79	0.39	1.11	3.89	9.97
rh_20_100	2.32	0.49	1.28	5.73	11.69
rh_30_100	2.20	1.61	1.15	28.37	17.62
rh_40_100	2.16	3.58	1.19	77.75	21.72

Table 6. Performance (FLOPc) and time (s) of PolyVest and Opt-V and other types of Polytopes. Opt-V achieves higher performance and faster runtime across all polytopes, increasing with the dimension of the polytope.

5. CONCLUSIONS

In this paper, we present a collection of improvements for the efficient volume computation of convex polytopes. We apply conventional performance-boosting measures, including precomputation and vectorization techniques. We identify critical mathematical attributes of the baseline coordinate directions method, enabling the creation and deployment of a considerably faster algorithm. We empirically test each step of our enhancements and notice steady advancements in performance and runtime. We validate these enhancements across diverse polytopes, from cubes to random hyperplanes. Our methodology consistently outperforms the benchmark PolyVest implementation, often by a significant margin (up to 115 times), especially for high-dimensional cases. Importantly, our significantly faster algorithm has broad applicability and can be incorporated into any problem that employs the coordinate directions method. We also delve into other enhancement strategies, such as PGO, though these did not lead to statistically meaningful performance gains. Future investigations could probe into why these enhancements failed to boost performance, to tweak them for potential advantages.

6. CONTRIBUTIONS OF TEAM MEMBERS

Here we highlight the salient contribution of each team member to the project.

Samuel. Contributed to the foundational design and preprocessing for the initial implementation. Conducted an early hotspot analysis using VTune, which informed the first phase of optimizations (Opt-I). Employed Linux perf for preliminary profiling and performance assessment. Benchmarked the initial implementations and PolyVest. Executed hotspot analyses for all subsequent optimizations with VTune, including Opt-III and Opt-IV, which drove the decision to eliminate the Armadillo objects. Conducted cache and memory analysis for Opt-III and Opt-IV. Conducted roofline analyses using Intel® Advisor. Wrote bash and python scripts to automate roofline, performance, and hotspot analyses across all branches. Conducted peak performance analysis. Carried out all benchmarks. Wrote scripts to create all plots for performance, speedup, and roofline in the experimental section. Conducted statistical evaluations to ensure the accuracy of results.

Deepak. Worked on implementation of `walk()` and `estimateVol()` with Vraj for the baseline. Implemented the first stage of Optimization. Mathematically formalized, and implemented Optimization - II. Debugged the implementation of Optimization - II with Vraj and Raghu. Implemented most of the vectorization outside `walk()` and introduced additional precomputation leading to more vectorization, as part of Optimization - III. Removed Armadillo objects and ensured aligned objects with Vraj, taking up the first half of Optimization - IV. Implemented the optimized norm computation, resulting in improved speedups, as part of Optimization - V.

Vraj. Worked on implementation of `walk()` and `estimateVol()` for the baseline with Deepak. Did major round of debugging of the baseline with Raghu. Added Xoshiro random number generation. Contributed the idea for Optimization II. Debugged the implementation of Optimization II with Deepak and Raghu. Aligned vector allocation and removal of Armadillo objects for Optimization IV with Deepak. Did code refactoring and re-implementation of some helper functions for Optimization-IV for reducing the precision for vectorized operation. Implemented merging of vectorized loops and code refactoring which enabled hardcoding input size in source (for Optimization V). Also wrote Python script and Makefile which enabled Profile Guided Optimization, merging code into a single file and the use of Clang as compiler.

Raghu. Implemented the skeleton of the baseline project and also worked on the implementation of `Preprocess()`, `EstimateVol()` and `Walk()`. Perfomed two major debugging session of the `Preprocess()`, with Vraj, and `EsimateVol()`, with Deepak and Vraj. Implemented the validation infrastructure which include scripts for generating various polytopes and testing the output of the executable on those test cases. Helped debug the working of Xoroshiro and the implementation of Optimization II with Vraj and Deepak. Implemented the vectorization of the `Walk()` function, and the optimized the same by precomputing the comparison masks in `EstimateVol()`.

7. REFERENCES

[1] Benno Büeler, Andreas Enge, and Komei Fukuda, *Exact Volume Computation for Polytopes: A Practical Study*, pp. 131–154, Birkhäuser Basel, Basel, 2000.

[2] Benno Büeler and Andreas Enge, "Vinci: Polytope volume computation," Nov. 2020.

[3] Cunjing Ge, Feifei Ma, and Jian Zhang, "A fast and practical method to estimate volumes of convex polytopes," *CoRR*, vol. abs/1401.0120, 2014.

[4] 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.

[5] L G Khachiyan, "The problem of calculating the volume of a polyhedron is enumerably hard," *Russian Mathematical Surveys*, vol. 44, no. 3, pp. 199, jun 1989.

[6] Martin Grötschel, László Lovász, and Alexander Schrijver, *Geometric algorithms and combinatorial optimization*, vol. 2, Springer Science & Business Media, 2012.

[7] HCP Berbee, CGE Boender, AHG Rinnooy Ran, CL Scheffer, Robert L Smith, and Jan Telgen, "Hit-and-run algorithms for the identification of nonredundant linear inequalities," *Mathematical Programming*, vol. 37, pp. 184–207, 1987.

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

[9] Claude JP Bélisle, H Edwin Romeijn, and Robert L Smith, "Hit-and-run algorithms for generating multivariate distributions," *Mathematics of Operations Research*, vol. 18, no. 2, pp. 255–266, 1993.