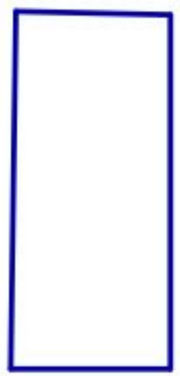
Efficient Volume Estimation of Convex Polytopes in High Dimensions

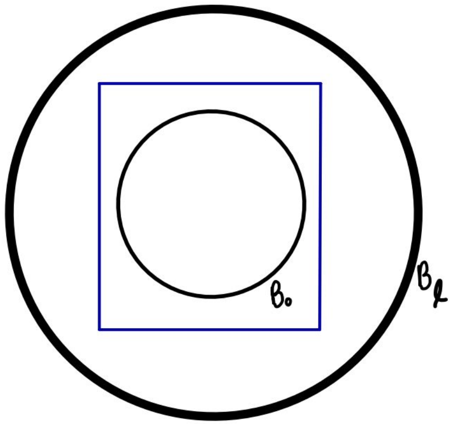
Deepak, Raghu, Samuel, Vraj

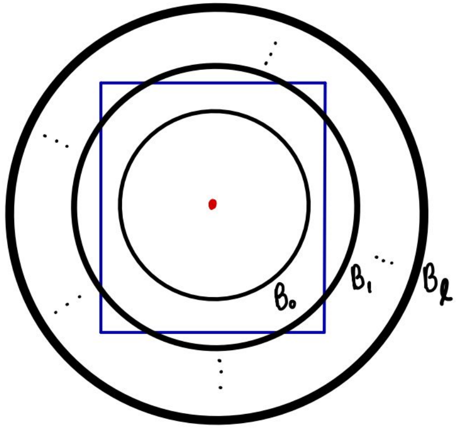
Advisor: Yann Girsberger

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Algorithm







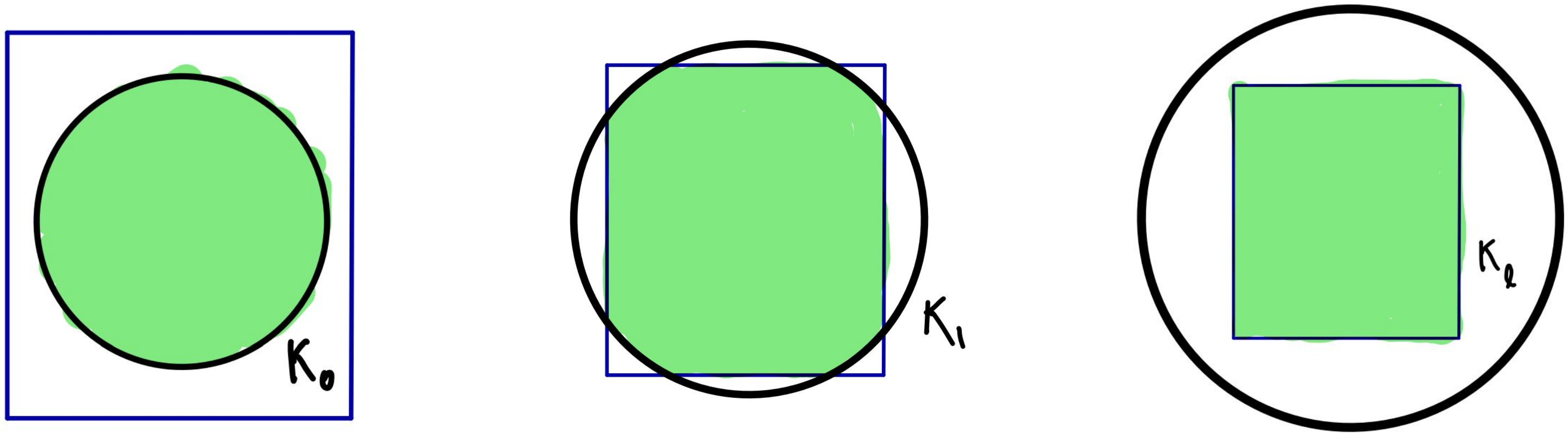
Given Polytope description: $Q = \{x : Ax \leq b\},\ A \in \mathbb{R}^{m \times n},\ b \in \mathbb{R}^m$

Task: Find volume of above polytope

Approach:

1. `Preprocess`: Affine transform T s.t. $B(0,1) \subseteq P = T(Q) \subseteq B(0,r)$
2. `EstimateVol`: Concentric balls $\{B_i\} = \left\{B\left(0, 2^{i/n}\right)\right\} \forall i \in [\ell] \cup \{0\}$

Algorithm



$$\ell = \lceil n \log_2 r \rceil,\ K_i = B_i \cap P$$

$$vol(P) = vol(B(0,1))\, \Pi_{i=0}^{\ell-1}(vol(K_{i+1})/vol(K_i))$$

$$\alpha_i = vol(K_{i+1})/vol(K_i)$$

Estimate α_i using random walk (Monte Carlo Method)

Cost Analysis

- Cost measure: adds, subs, mults, divs
- `Preprocess`: Ellipsoid method for rounding (polynomial time)
- `Estimatevol`: Calls `walk()` $\leq 1600\left(n \log_2(2n)\right)^2$ times
- `Walk`: $(2mn + m + 2n + \mathcal{O}(1))$ flops per call.
- Overall complexity of `Estimatevol` is $\tilde{\mathcal{O}}\left(mn^3\right)$
- As n increases, we expect the bottleneck to be `Walk`

Baseline

All benchmarks were performed on:

Intel Comet Lake i7-10610U @ 1.80GHz

Compiler: GCC 11.3.0

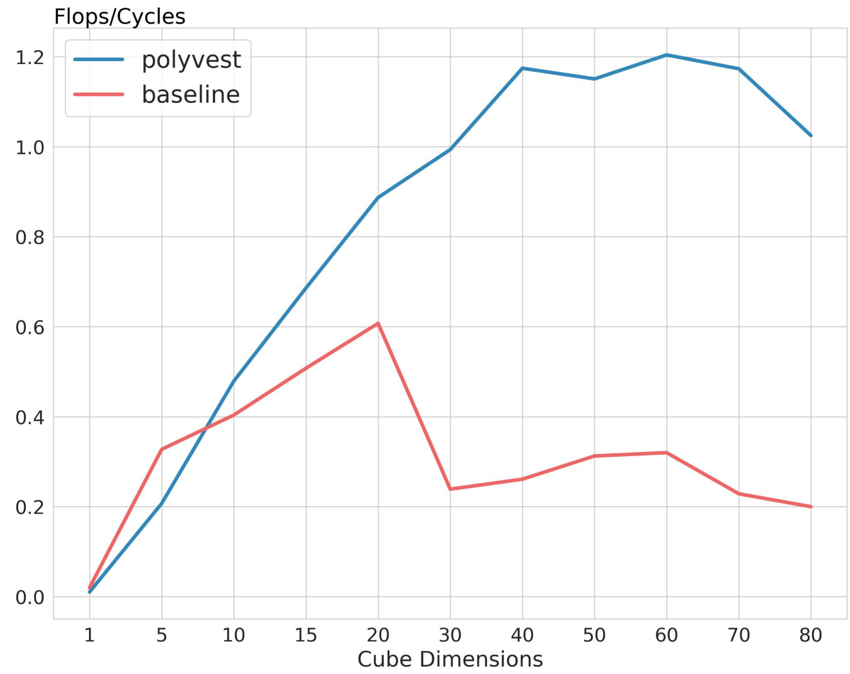
-O3 -march=native -ffast-math

Baseline:

- Focus on algorithm in Ge et al. 2013
- No additional optimization
- Benchmark against PolyVest

Benchmark:

- Linux perf
- 5 repeats
- E.g.: `perf stat -M FLOPc -r 5 ./polyvol tests/cube_80`



Performance on cubes of varying dimensions

Hotspot Analysis

- Basic Hotspot analysis (Intel VTune)
- Focus on polytope::walk

Function Stack	CPU Time: Total	CPU Time: Self
Total	100.0%	0s
_start	100.0%	0s
__libc_start_main_impl	100.0%	0s
main	100.0%	0s
polytope::preprocess	0.0%	0s
polytope::estimateVol	100.0%	7.817s
polytope::walk	98.6%	369.062s
func@0x2754	0.0%	0.020s
func@0x2594	0.0%	0.008s
__pow	0.3%	2.273s
__log2	0.1%	0.978s
__ceil_sse41	0.0%	0.072s
[Unknown stack frame(s)]	0.0%	0s

Hotspots for baseline on cube_40

Optimization - I

- Used DP for more efficient computation of volume of a unit ball.
- Unroll loops in norm and estimatevol for ILP.
- Used more efficient random number generator (Xoshiro).
- Removed Bound checks from array access.
- Precomputed division operations in `walk`.

Optimization - I

$A \in \mathbb{R}^{m \times n}, B$ is an array of vectors.

Algorithm 1: Walk (Before pre-computation)

$d \leftarrow \textsc{random}(\{1, 2, 3, \ldots, n\})$

Constant Time computation

$bound \leftarrow B[d] - (A \;/\; (A[:, d].\mathbf{1}^T)).x$ // Bottleneck - $\mathcal{O}(m + mn)$ time

$\mathcal{O}$(m) computation for MIN and MAX

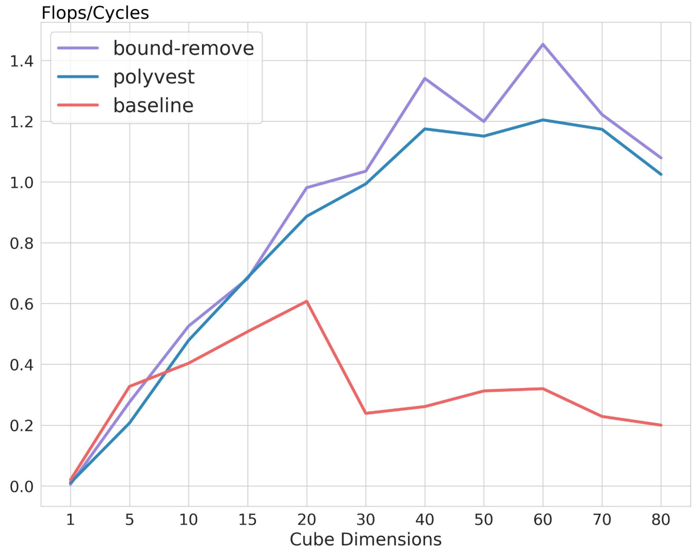
$x_d \leftarrow x_d + \textsc{Random}(\textsc{min}, \textsc{max})$ // Update x in the chosen direction

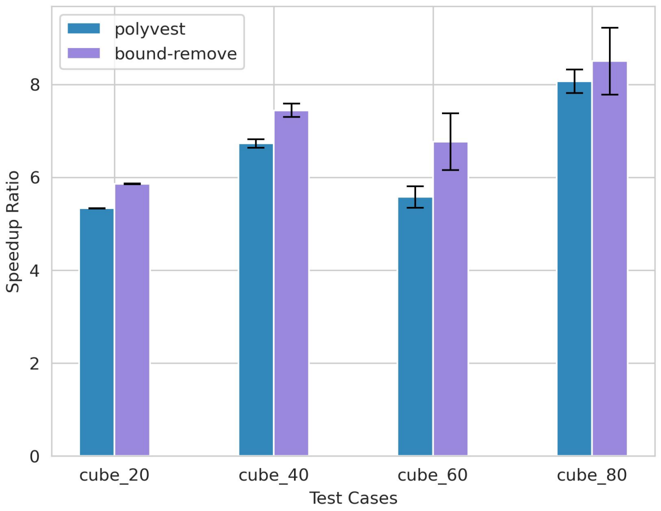
Optimization - I

$$\forall j \in [n]\ \tilde{A}[j] := A\ /\ (A[:, j].\mathbf{1}^T)$$

Algorithm 2: Fast Walk I: After pre-computation

1 $d \leftarrow$ RANDOM($\{1, 2, 3, \ldots, n\}$)
2 Constant Time computation
3 $bound \leftarrow B[d] - \tilde{A}[d].x$ // Bottleneck - $\mathcal{O}(m + mn)$ time
4 $\mathcal{O}$(m) computation for MIN and MAX
5 $x_d \leftarrow x_d +$ RANDOM(MIN, MAX) // Update x in the chosen direction





Speedup Ratio over "baseline" (time)

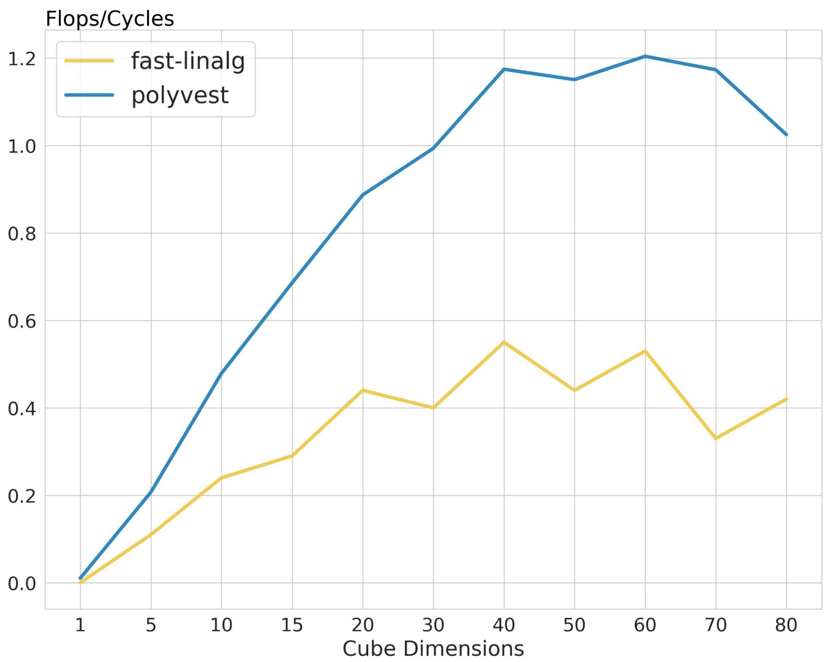
Optimization - II

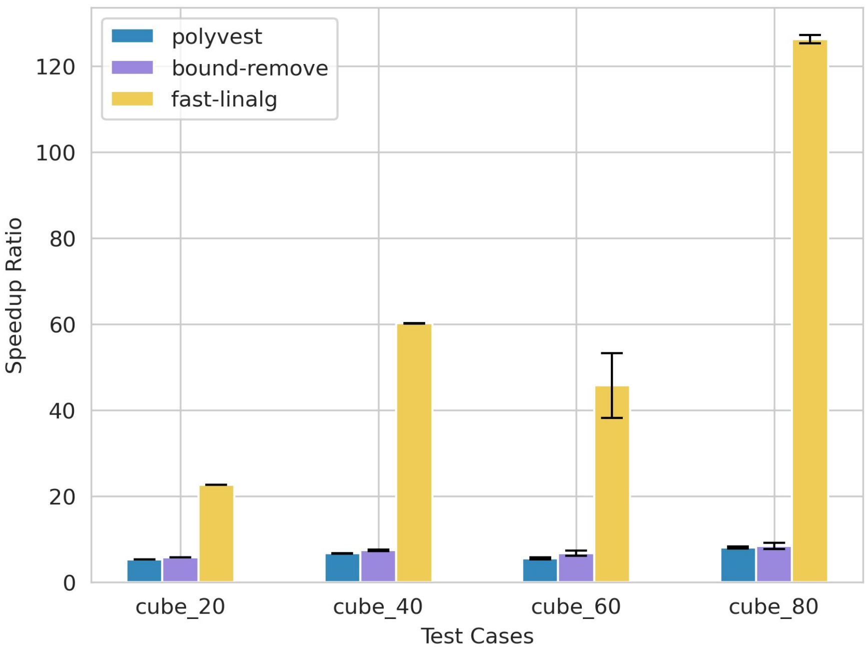
Algorithm 3: Fast Walk II: Asymptotically Fast

$d \leftarrow$ RANDOM($\{1, 2, 3, \ldots, n\}$)
Constant Time computation
$bound \leftarrow B[d] - A.x/A[:, d]$ // Now - $\mathcal{O}(m)$ time
$\mathcal{O}$(m) computation for MIN and MAX
$v \leftarrow$ RANDOM(MIN, MAX)
$x_d \leftarrow x_d + v$ // Update x in chosen direction
$A.x \leftarrow A.x + vA[:, d]$ // x changed, so update A.x - $\mathcal{O}(m)$ time

Maintain the vector $A.x$

Optimization - II

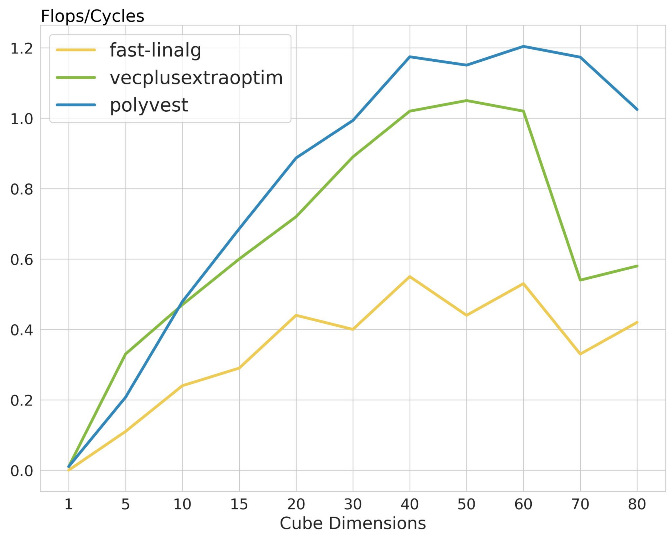


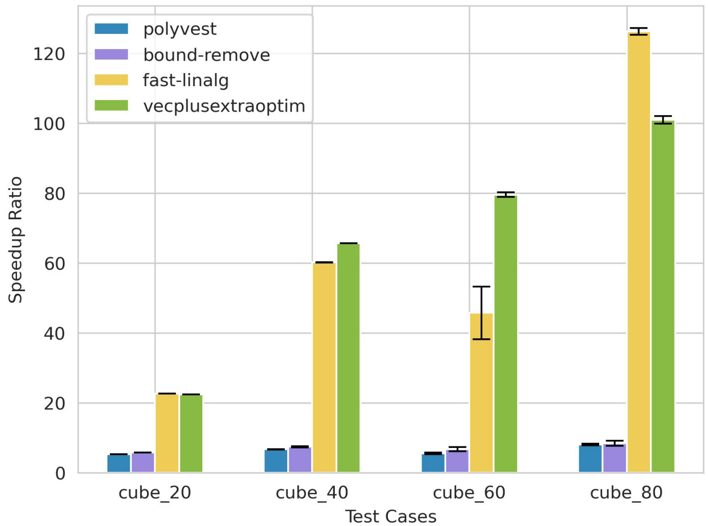


Speedup Ratio over "baseline" (time)

Optimization - III

- Vectorized norm computation and loops in `walk`
- Vectorized calculations in `estimateVol`.
- Precomputed some vector data types used in `walk`





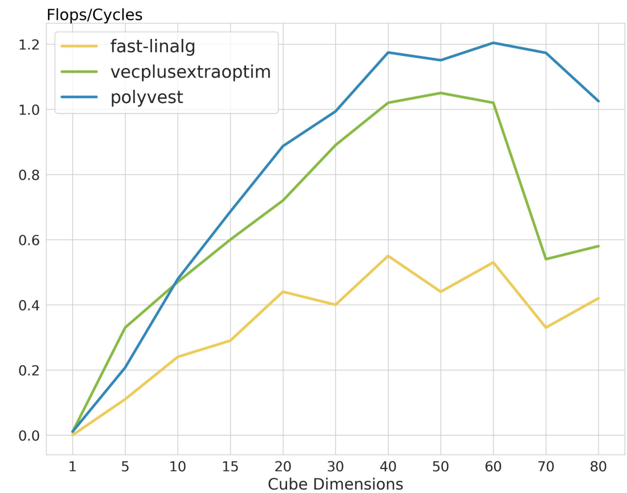
Speedup Ratio over "baseline" (time)

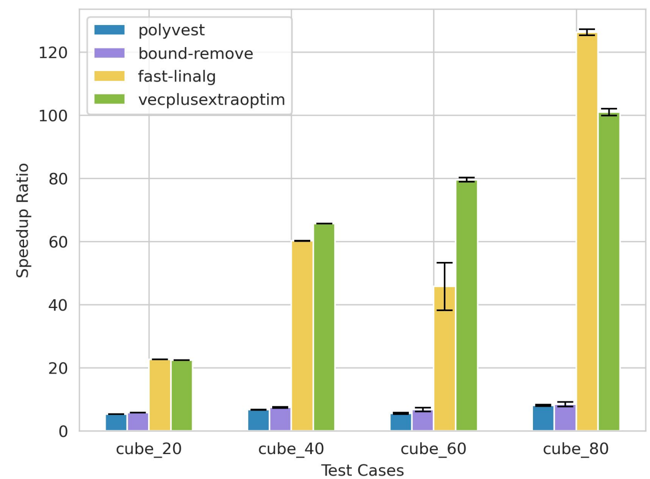
Optimization - III

Why no speedup for larger dimensions?

Function	Module	CPU Time	% of CPU Time
__posix_memalign	libc.so.6	98.187s	44.2%
_ZNK8polytope4walkEPdS0_PKdRKN4arma3MatIdEEPKDv4_dSA_dRN10XoshiroCpp18Xoshiro128PlusPlusE	polyvol	28.207s	12.7%
__memcpy_avx_unaligned_erms	libc.so.6	26.534s	11.9%
__GI_	libc.so.6	8.564s	3.9%
_Z15_mm256_store_pdPdDv4_d	polyvol	6.422s	2.9%
[Others]	N/A*	54.255s	24.4%

```
vec A_dir = A.col(dir), A_negrecp_dir = A_negrecp.col(dir);
```





Speedup Ratio over "baseline" (time)

Optimization - IV

- Replaced most Armadillo objects with (aligned) double*
  - Simpler Pointer Arithmetic
  - Further Vectorization possible
  - Column Major Format
- Reduced precision - From doubles to floats

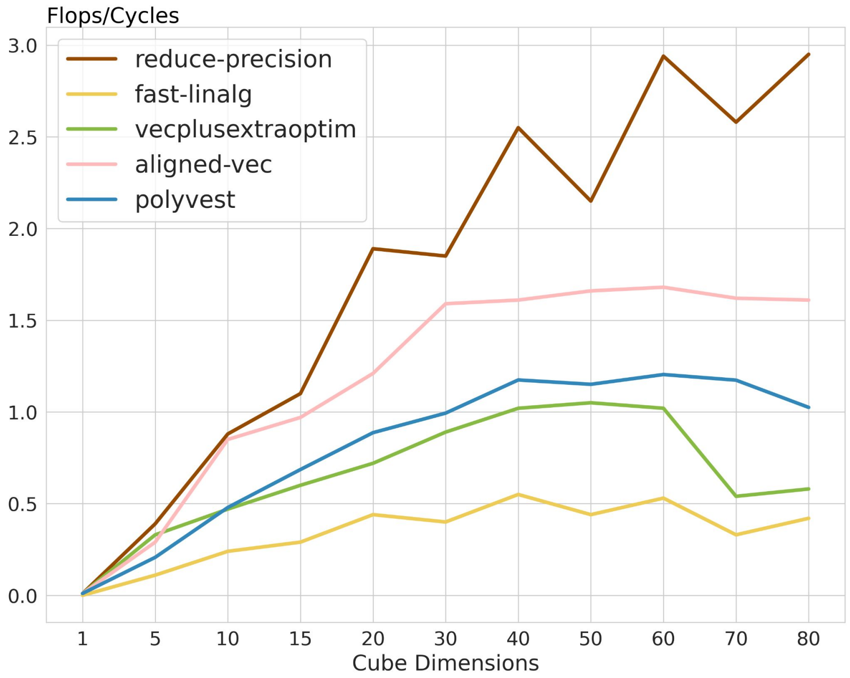
Function	Module	CPU Time	% of CPU Time
_ZNK8polytope4walkEPdS0_PKdS2_PKDv4_dS5_dRN10XoshiroCpp18Xoshiro128PlusPlusE	polyvol	40.680s	34.8%
_mm256_blendv_pd	polyvol	12.787s	10.9%
_Z15_mm256_store_pdPdDv4_d	polyvol	7.890s	6.8%
_mm256_max_pd	polyvol	7.676s	6.6%
norm_2	polyvol	7.452s	6.4%
[Others]	N/A*	40.355s	34.5%

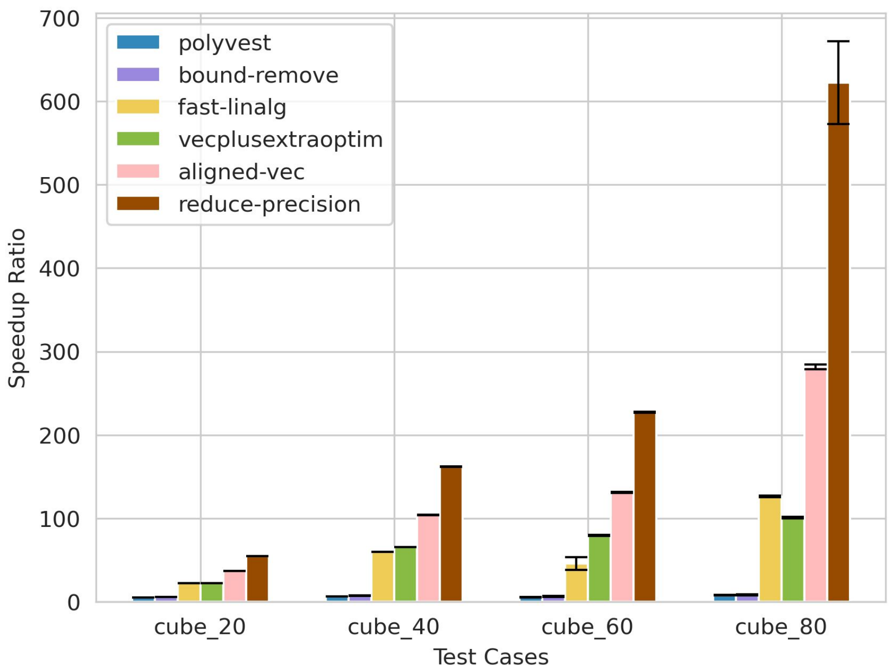
Allocations now outside `Walk`

```
-   vec A_dir = A.col(dir), A_negrecp_dir = A_negrecp.col(dir);
+   double* A_dir = A_arr + m*dir;
```

```
+   double *A_negrecp = (double *) aligned_alloc(32, align_pad(m*n));
```

Optimization - IV - Results



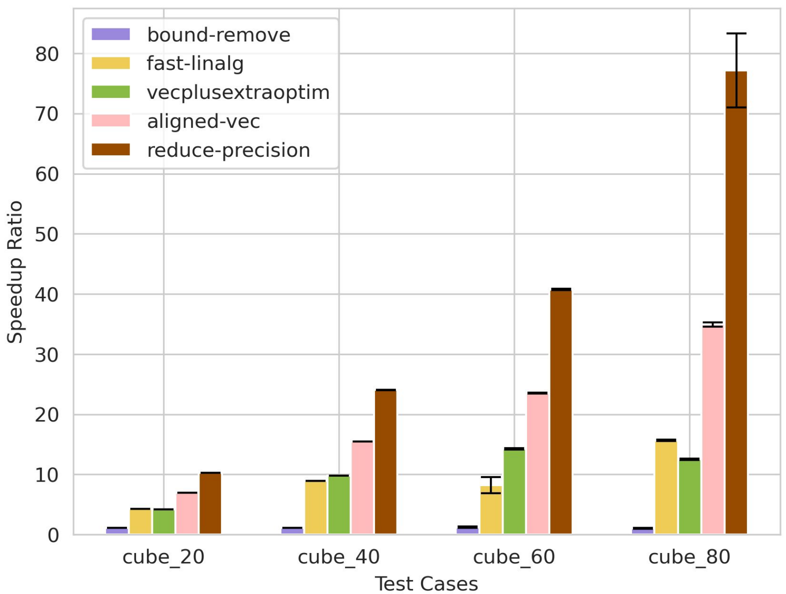


Speedup Ratio over "baseline" (time)

`aligned-vec` : Optimizations removing memory allocations, introducing aligned objects resulting in simpler pointer arithmetic and further vectorization

`reduce-precision` : Precision reduction from doubles to floats on top of `aligned-vec`

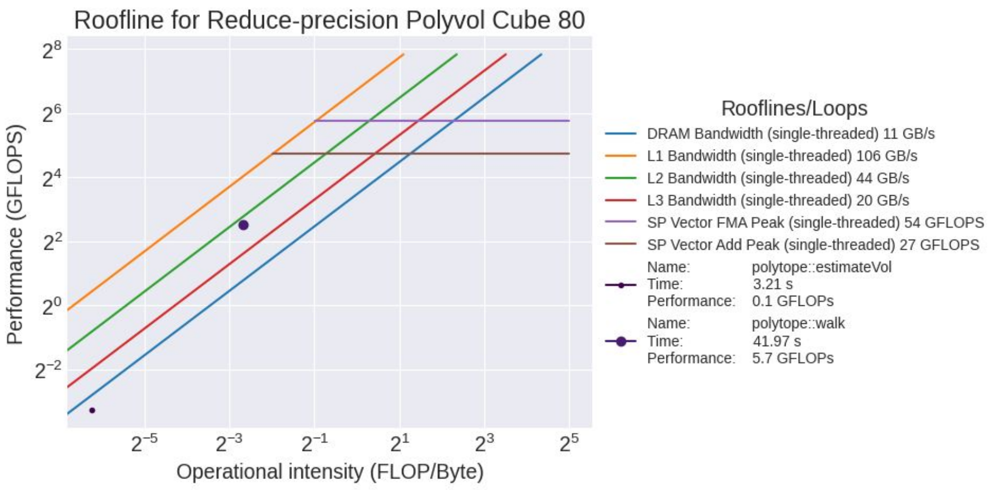
Final Results



Speedup Ratio over "polyvest" (time)

Method	Precision	GFLOPS	% Peak
Baseline	DP	0.175	0.62
Reduce-Precision	SP	4.742	8.28

Percentage Peak Performance



Misc Optimizations

- **Profile-Guided Optimization for cube_80**

Branch	PGO Time (s)	No PGO Time (s)
Vecplusextraoptim	228.65s ± 3.33	231.88s ± 1.29
Reduce-precision	45.23s ± 0.331	38.83s ± 5.46

- **Onefile**

  No Improvement when including everything in one file.

- **Different Compiler (CLANG)**