High Performance Kernels for FFT

via Modern C++

Paul Caprioli Robby Jenkins

support@hpkfft.com

https://hpkfft.com

2024 January 23

$\boldsymbol{X} = \mathcal{F}(\boldsymbol{x})$ where $\boldsymbol{x}, \boldsymbol{X} \in \mathbb{C}^N$

$$X_k = \alpha \sum_{n=0}^{N-1} x_n \exp\left(\frac{-2\pi i}{N} kn\right)$$

1. Application Programming Interface
2. Accuracy Results
3. Performance Results

In-place Example

One line FFT computation:

```
1  #include <hpk/fft/makeFactory.hpp>
2
3  int main() {
4      std::vector<std::complex<double>> v =
5              {7.0, 0.0, 0.0, 0.0};
6
7      hpk::fft::makeFactory<double>()
8              ->makeInplace({4})
9              ->forward(v.data());
10 }
```

Design Principles

- $\mathcal{O}(N \log N) \ \ \forall N$
- Avoid globals
- Leverage static typing
- Function overloading
- Functions: free $\prec$ member $\prec$ friend
- Immutable $\implies$ thread safety

Broken Strawman (for a fake API)

```
1  int main() {
2      std::vector<std::complex<double>> x =
3              {7.0, 0.0, 0.0, 0.0};
4      std::vector<std::complex<double>> X(4);
5      auto fft =
6          std::make_unique<
7              Descriptor<Precision::float32,
8                         Domain::complex>>(4);
9      fft->setValue(Parameter::forwardScale,
10                   "ortho");
11     fft->commit();
12     computeForward(*fft, x.data(), X.data());
13 }
```

Out-of-place Example (with variables)

```
1  int main() {
2      std::vector<std::complex<double>> x =
3              {7.0, 0.0, 0.0, 0.0};
4      std::vector<std::complex<double>> X(4);
5      auto factory =
6              hpk::fft::makeFactory<double>();
7      auto fft = factory->makeOoplace({4});
8      fft->forwardCopy(x.data(), X.data());
9
10
11
12
13 }
```

Out-of-place Example (with scaling factor)

```
1  int main() {
2      std::vector<std::complex<double>> x =
3              {7.0, 0.0, 0.0, 0.0};
4      std::vector<std::complex<double>> X(4);
5      auto factory =
6              hpk::fft::makeFactory<double>();
7      auto fft = factory->makeOoplace({4});
8      fft->forwardCopy(x.data(), X.data());
9
10     double alpha = 0.5;
11     fft->scaleForwardCopy(&alpha, x.data(),
12                           X.data());
13 }
```

API: Strides

Consider

```
1          struct Datum {
2              double x_real;
3              double x_imag;
4              int    i;
5          };
6
7          std::vector<Datum> data;
```

Note that

```
sizeof(Datum) == 24;
alignof(Datum) == 8;
```

API: Strides for Out-of-place

So, given

```
1          struct Datum {
2              std::complex<double> x;
3              int                  i;
4          };
5
6          std::vector<Datum>   time;
7          std::vector<double> freq;
```

set

```
hpk::fft::OoplaceDim layout[1] =
                {{std::ssize(time), 3, 2}};
```

Memory Allocation

```
1  int main() {
2      std::vector<Datum> v = {{7.0, 0}, // etc.
3      auto factory =
4              hpk::fft::makeFactory<double>();
5
6      auto a = hpk::AlignedAllocator<double>();
7      auto fft = factory->makeInplace<1>(
8                  {{std::ssize(v), 3}}, 1, a);
9
10     auto scratch = allocateScratch(*fft);
11     fft->forward(v.data(), scratch);
12 }
```

Accuracy and Performance

- Note that `hpk::fft` orthogonally supports:
  - half, single, or double precision
  - in-place or out-of-place
  - unscaled or scaled (per call site)
  - arbitrary dimensions: 1D, 2D, 3D, ...
  - complex or real time domain $\leftrightarrow$ complex freq domain
- FP16 comparison: `ipp`
  - out-of-place, unscaled, 1D, complex $\leftrightarrow$ complex only
  - 58 specific lengths only
- FP32, FP64 comparison: `fftw` and `mkl`
  - 400 lengths chosen

Details

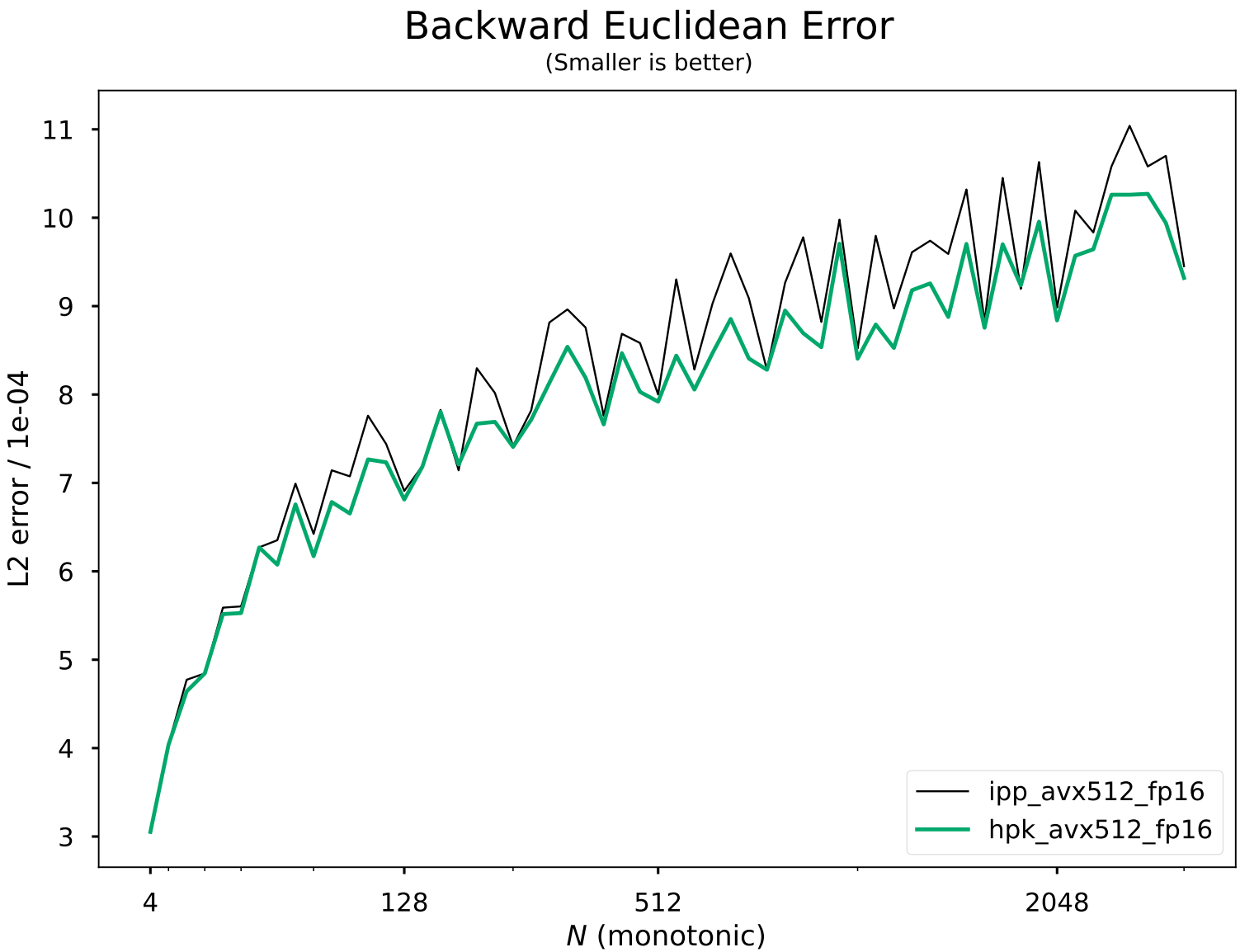
- High Performance Kernels 0.3.0
- `x86_64`
  - Intel Xeon w7-2495X (formerly Sapphire Rapids)
  - Debian 12
  - oneAPI 2024.0.0 (for `mkl`, `ipp`)
- `aarch64`
  - Amazon Graviton 3E processor
  - Amazon Linux 2023
- Environment
  - `OMP_NUM_THREADS=1`

Accuracy Summary

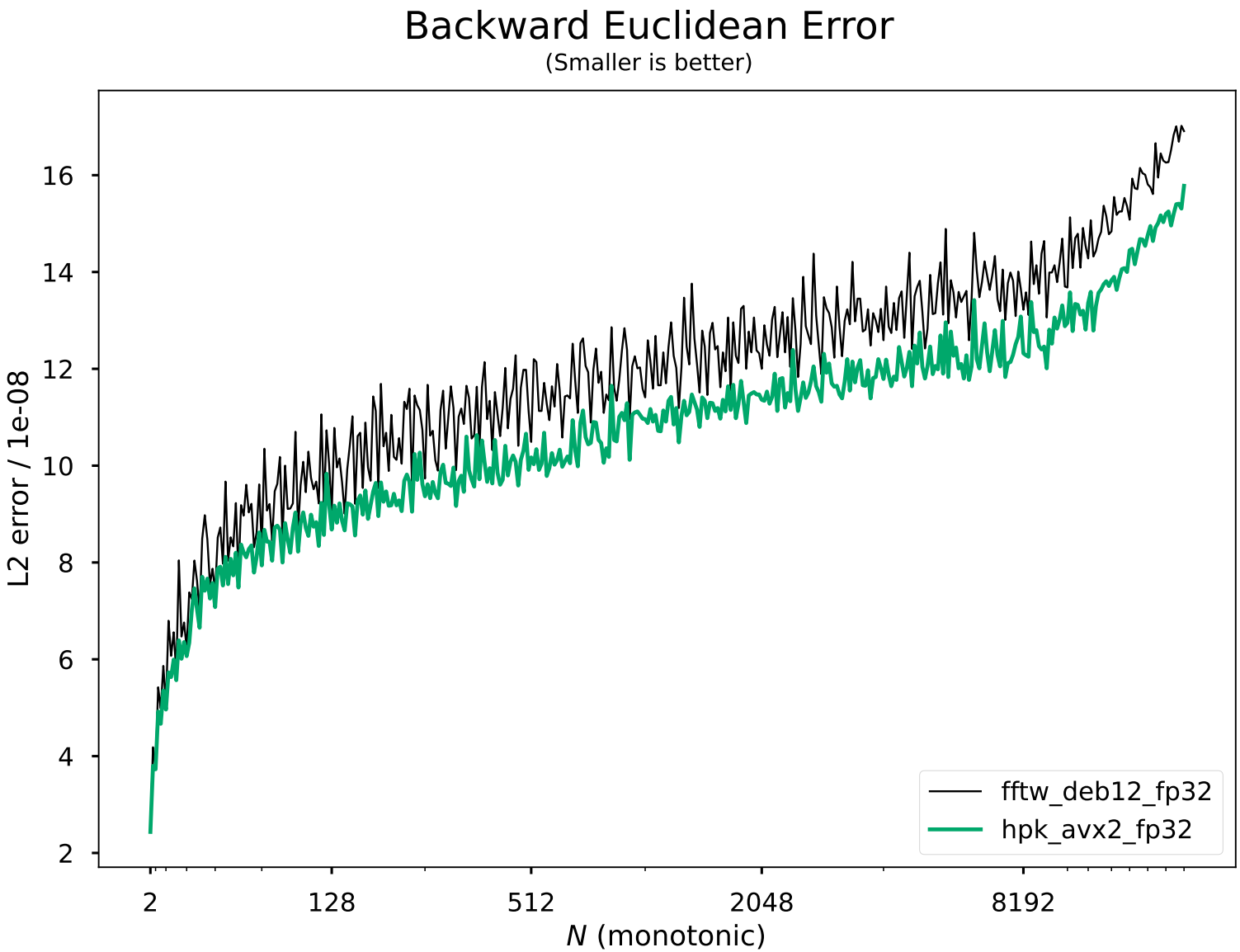
$$\text{error}_{\texttt{other}} \Big/ \text{error}_{\texttt{hpk::fft}}$$

hpk::fft	other	precision	ratio
hpk_avx2	fftw_deb12	fp32	1.105
		fp64	1.117
	mkl_avx2	fp32	1.047
		fp64	1.217
hpk_avx512	ipp_avx512	fp16	1.040
	mkl_avx512	fp32	1.196
		fp64	1.378
hpk_sve256	fftw_aws	fp32	1.023
		fp64	1.054

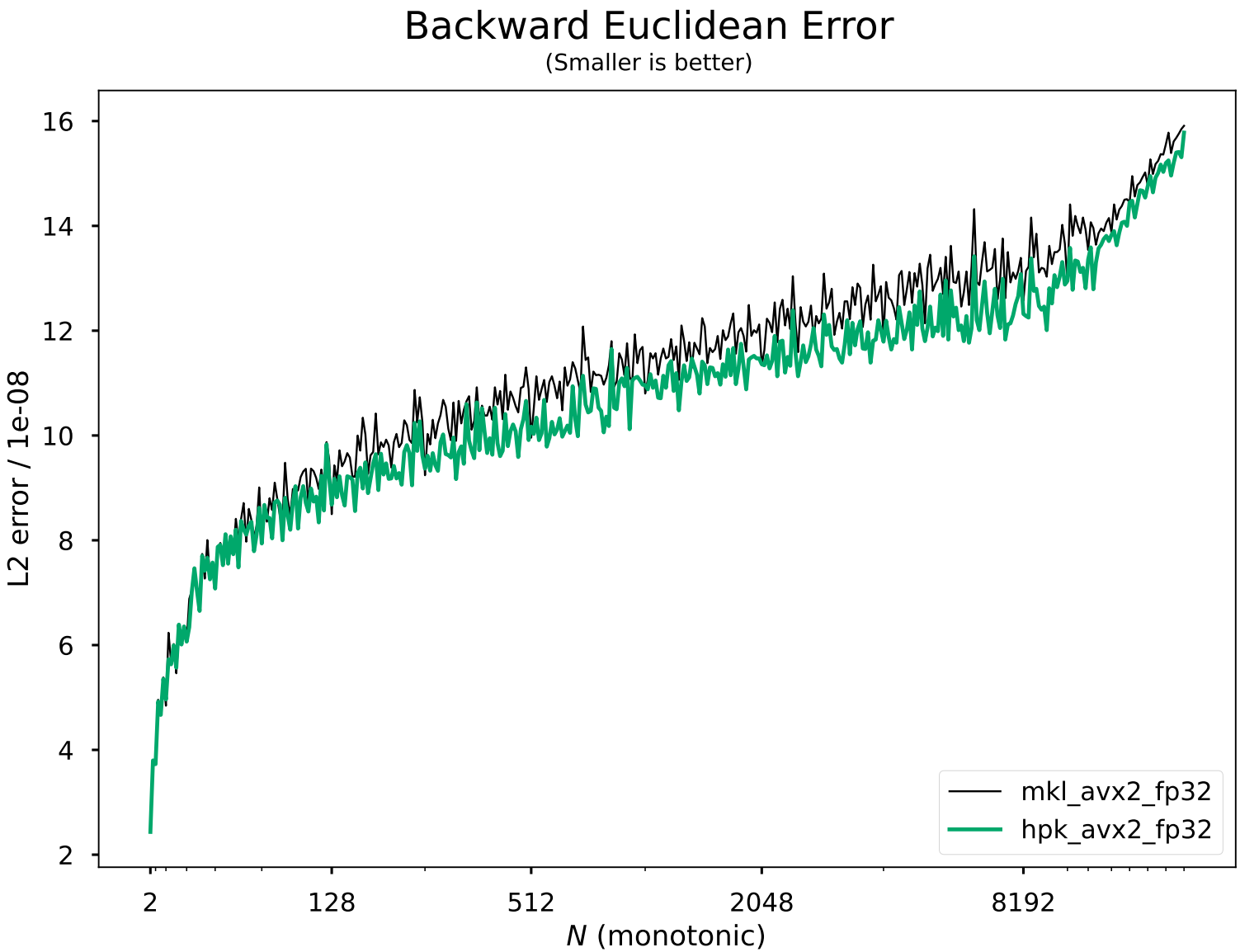
AVX512 Half Precision vs. IPP



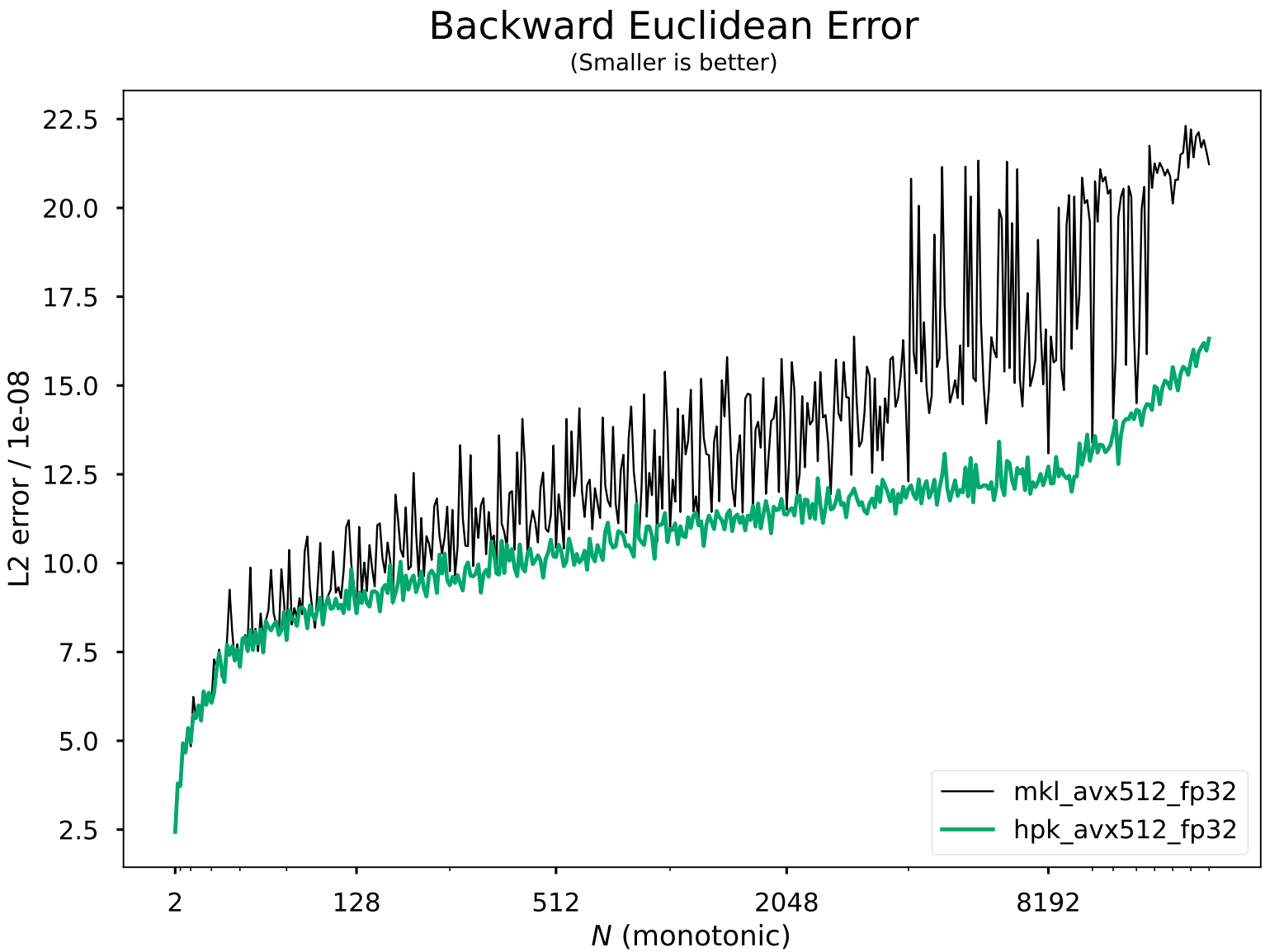
AVX2 Single Precision vs. FFTW



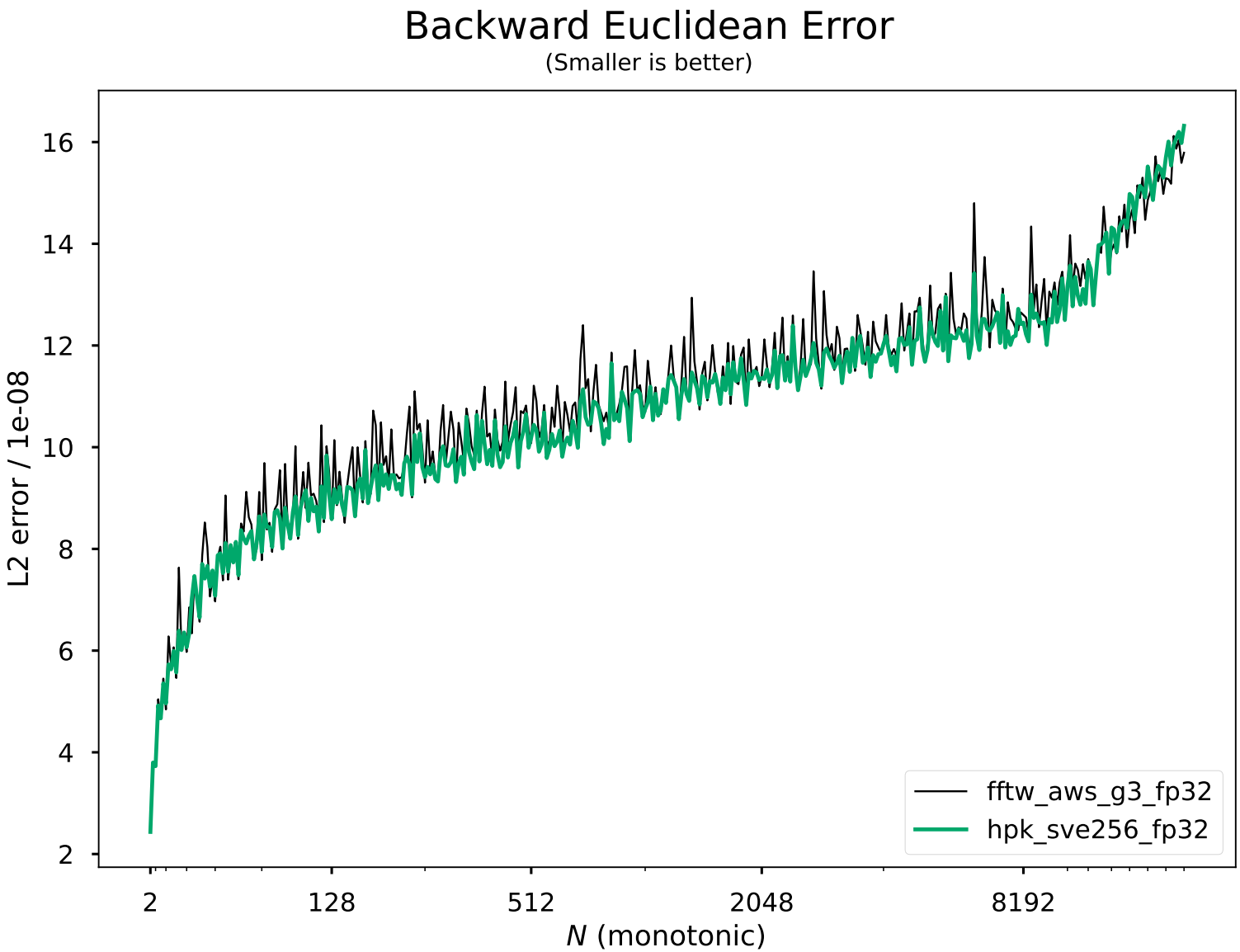
AVX2 Single Precision vs. MKL



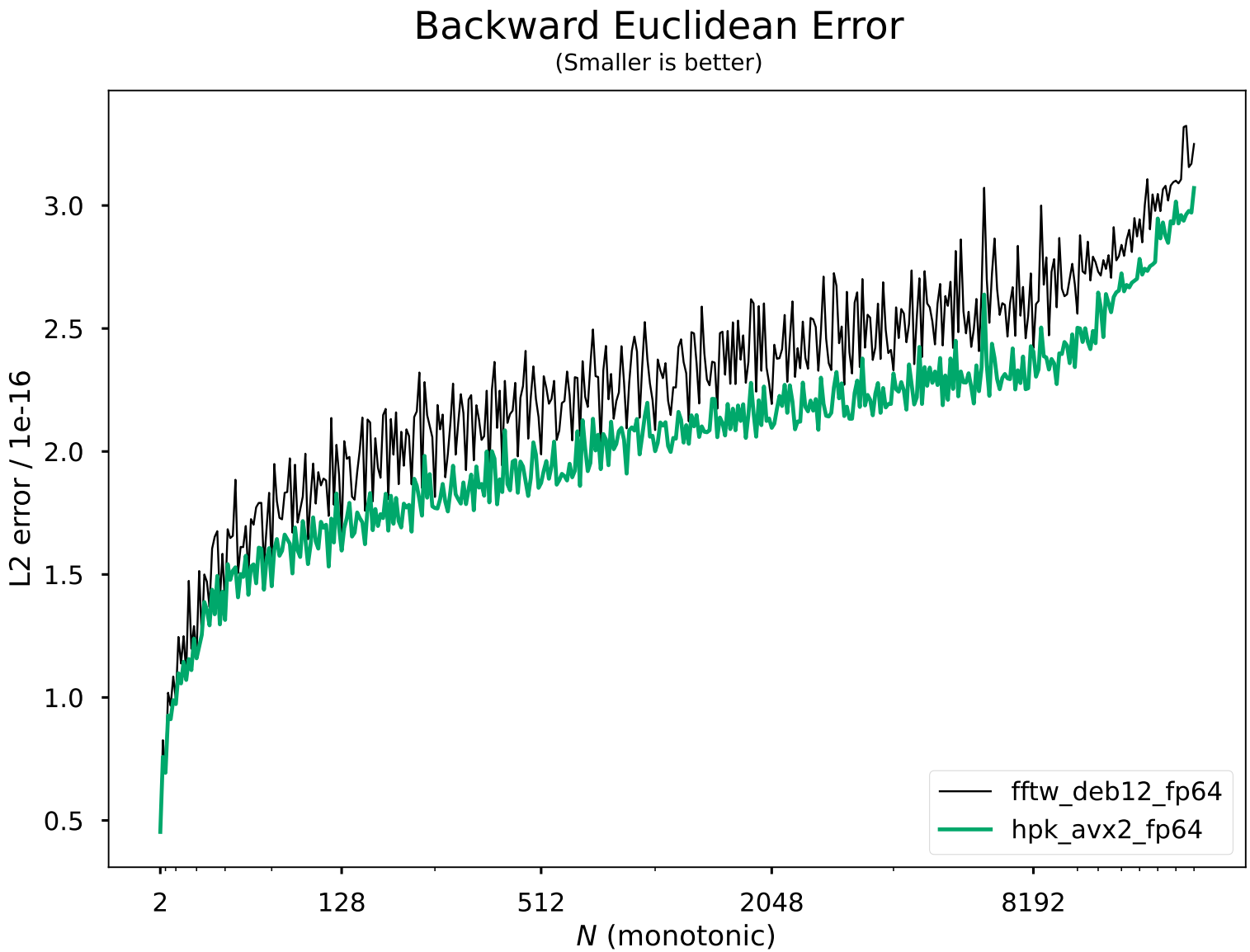
AVX512 Single Precision vs. MKL



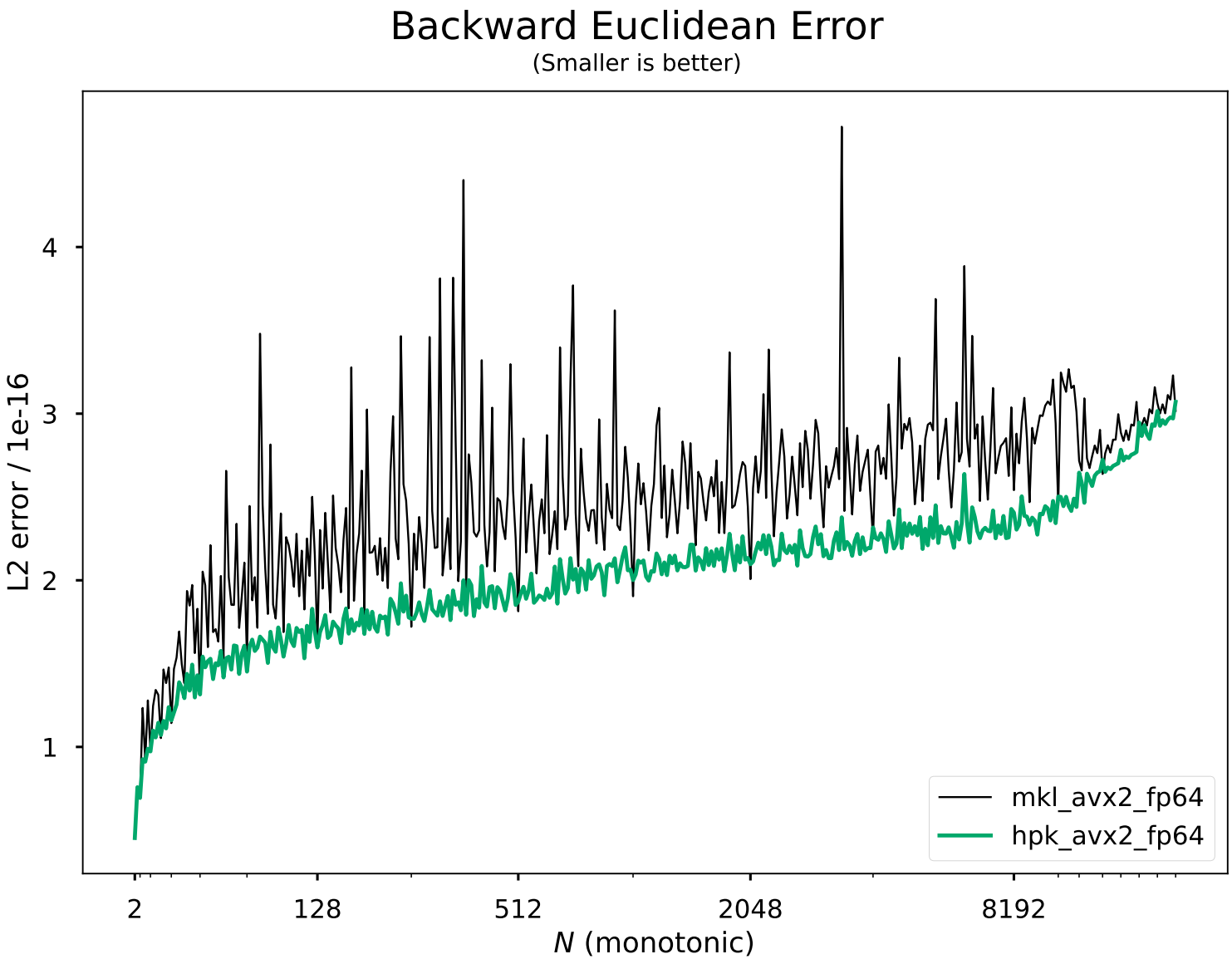
SVE256 Single Precision vs. FFTW



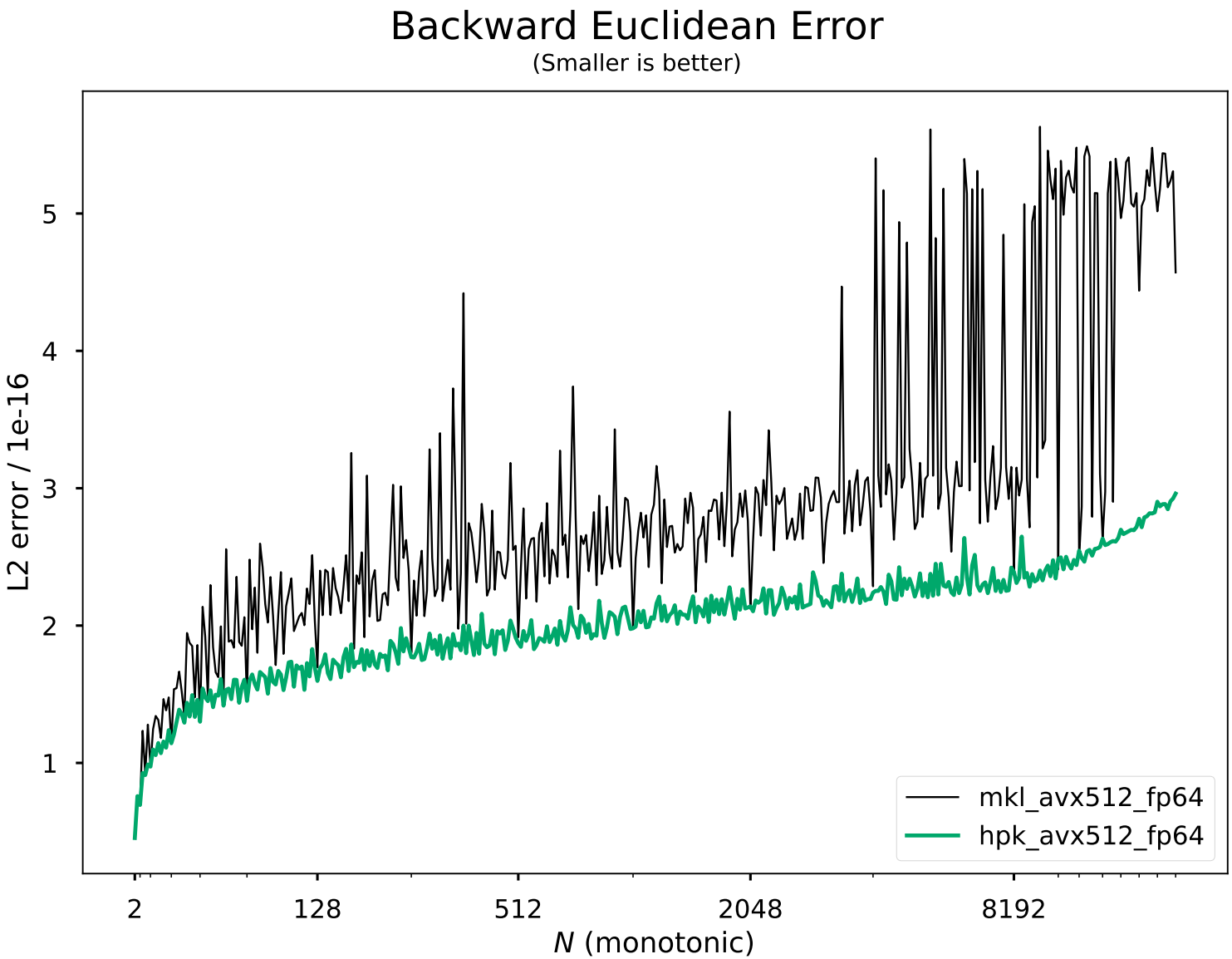
AVX2 Double Precision vs. FFTW



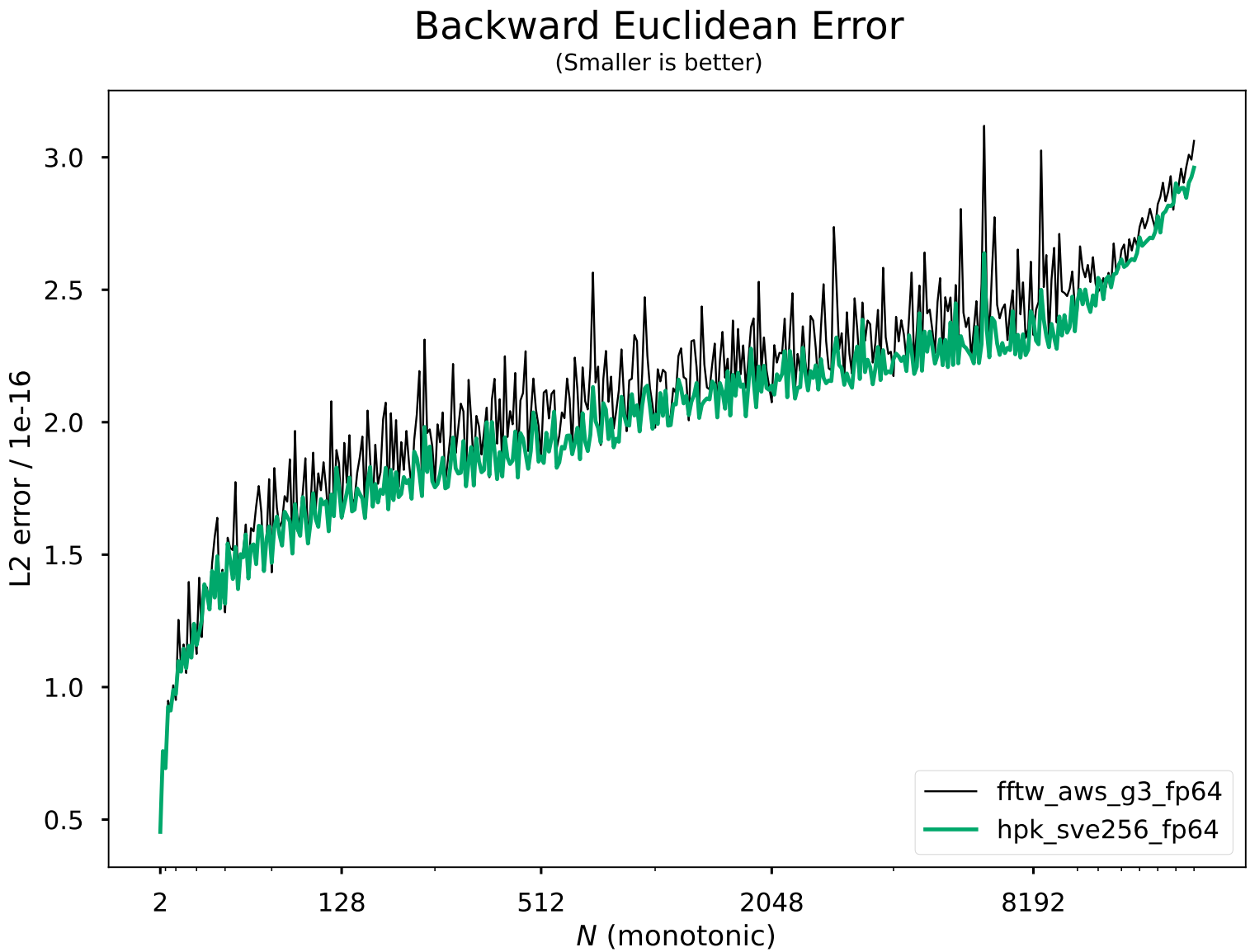
AVX2 Double Precision vs. MKL



AVX512 Double Precision vs. MKL



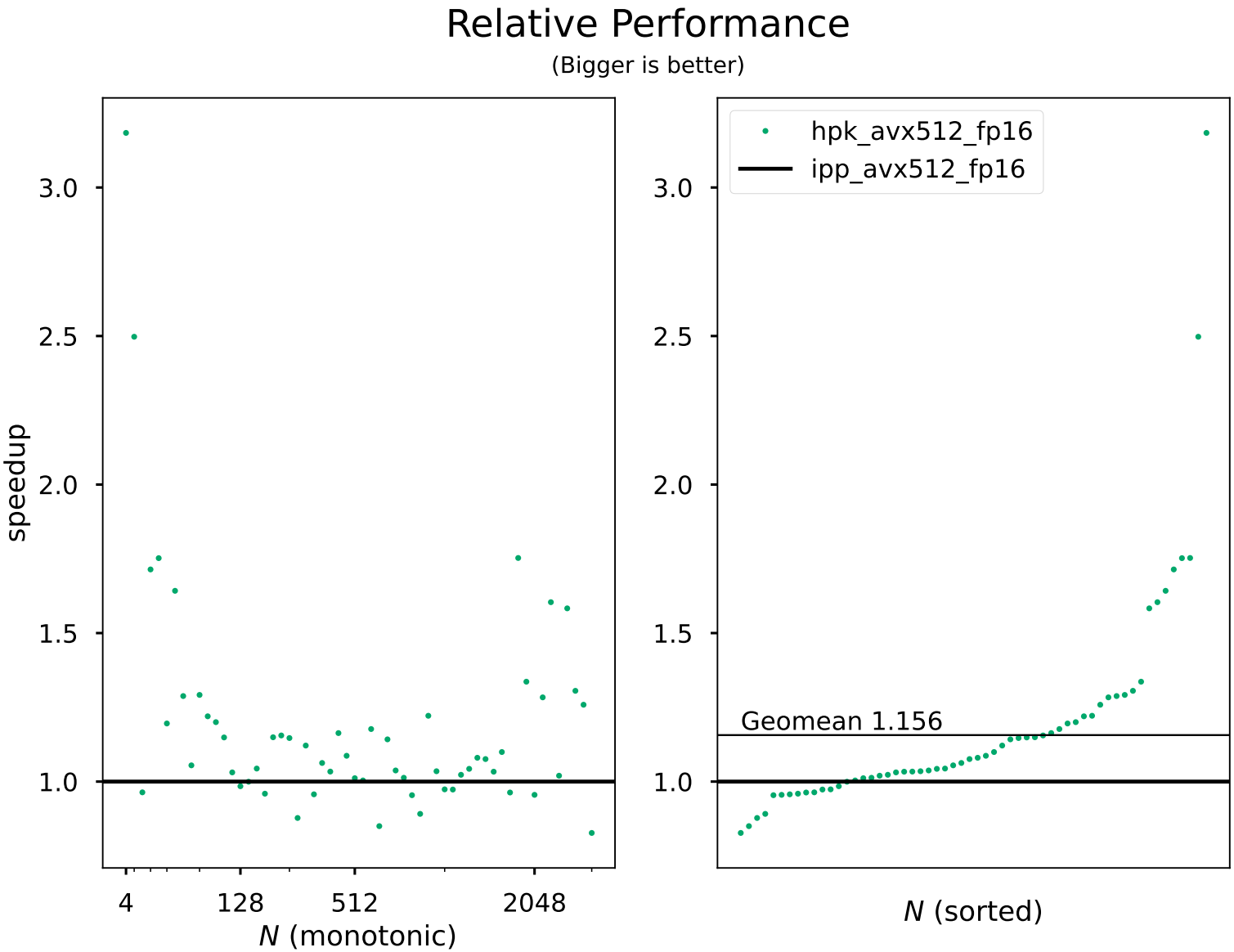
SVE256 Double Precision vs. FFTW



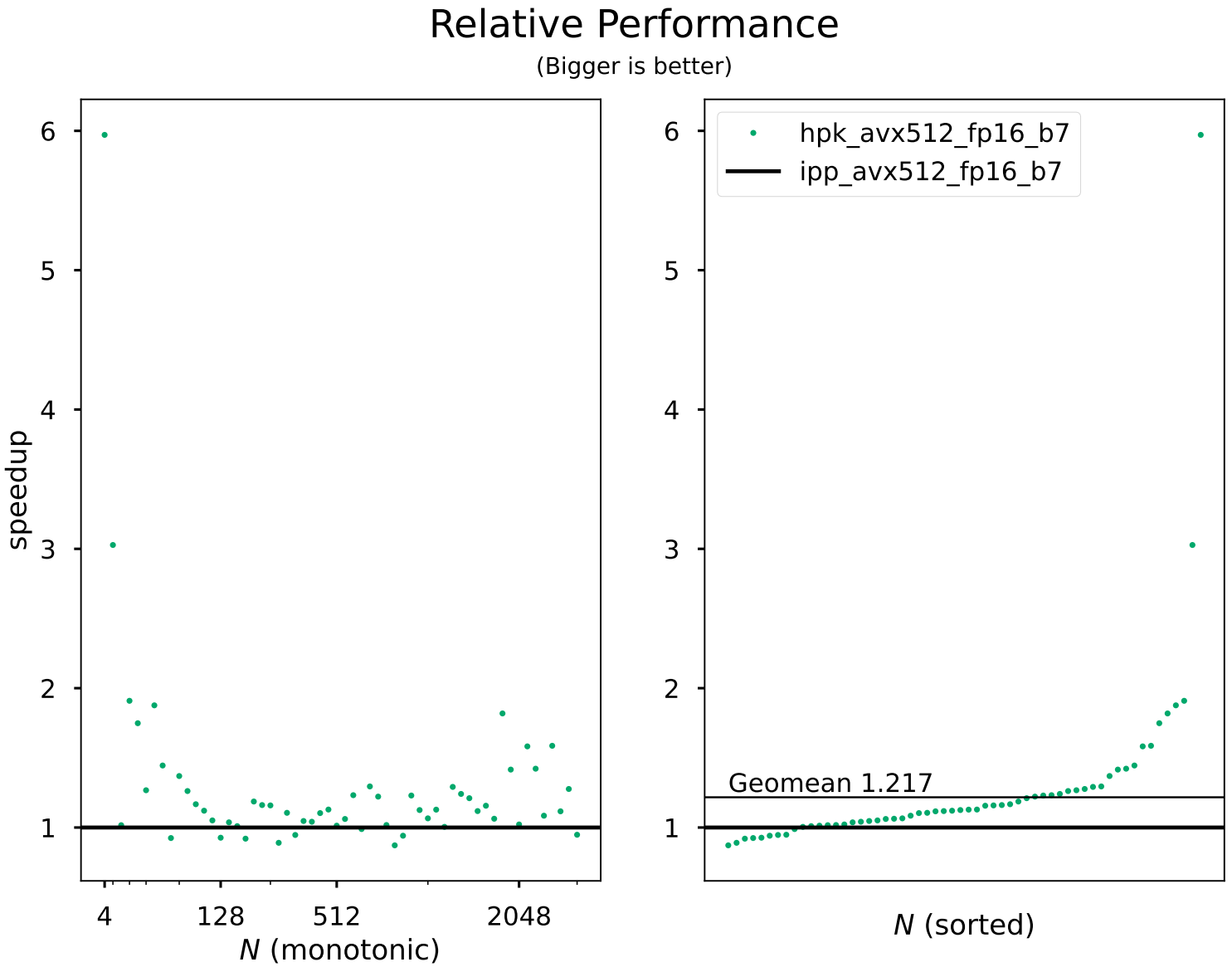
Performance Summary $\text{time}_{\texttt{other}} \,/\, \text{time}_{\texttt{hpk::fft}}$

hpk::fft	other	precision	ratio
hpk_avx2	fftw_deb12	fp32	2.228
		fp64	1.514
	mkl_avx2	fp32	1.617
		fp64	1.412
hpk_avx512	ipp_avx512	fp16	1.156
		fp16_b7	1.217
	mkl_avx512	fp32	1.358
		fp64	1.318
hpk_sve256	fftw_aws	fp32	3.882
		fp64	2.180

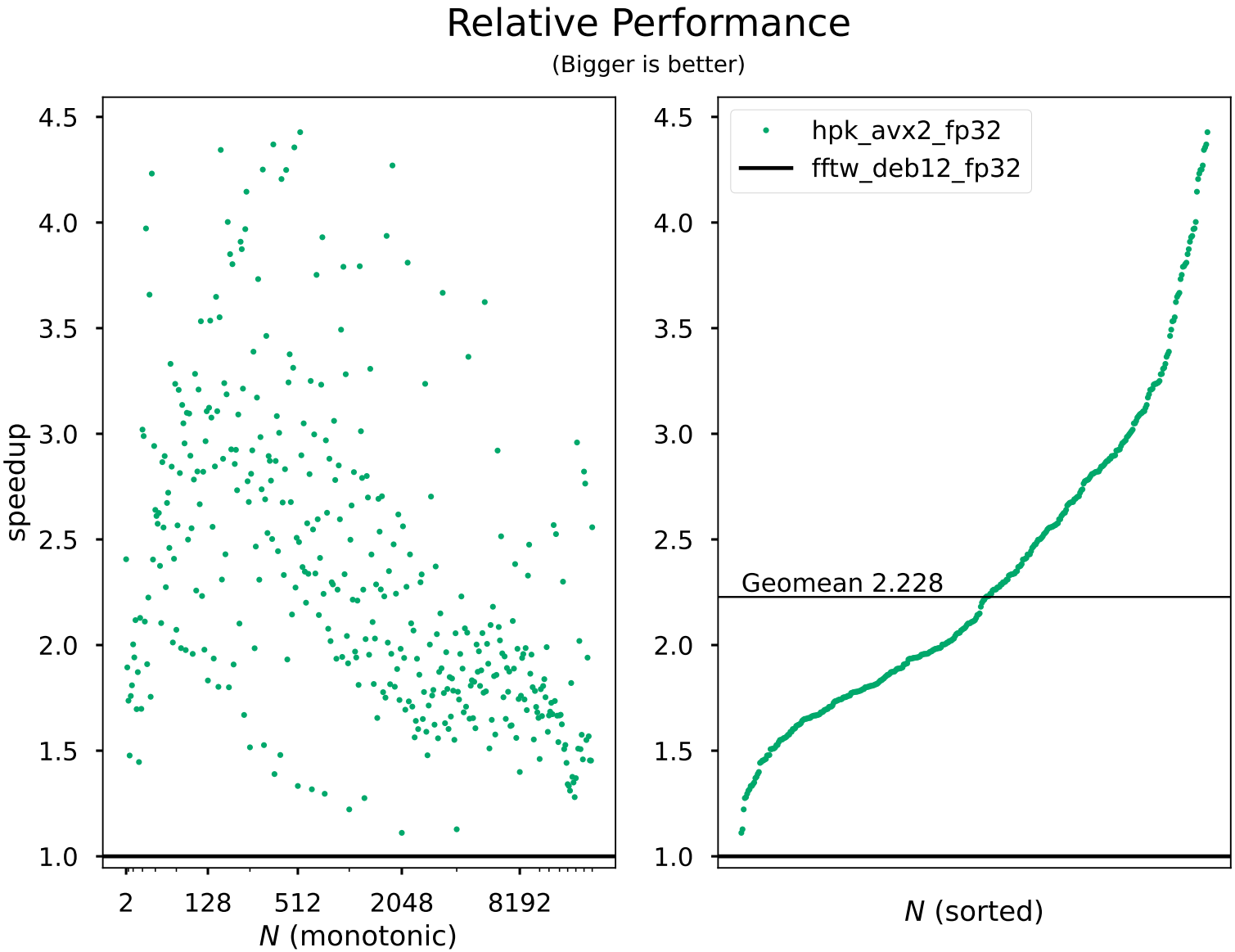
AVX512 Half Precision vs. IPP



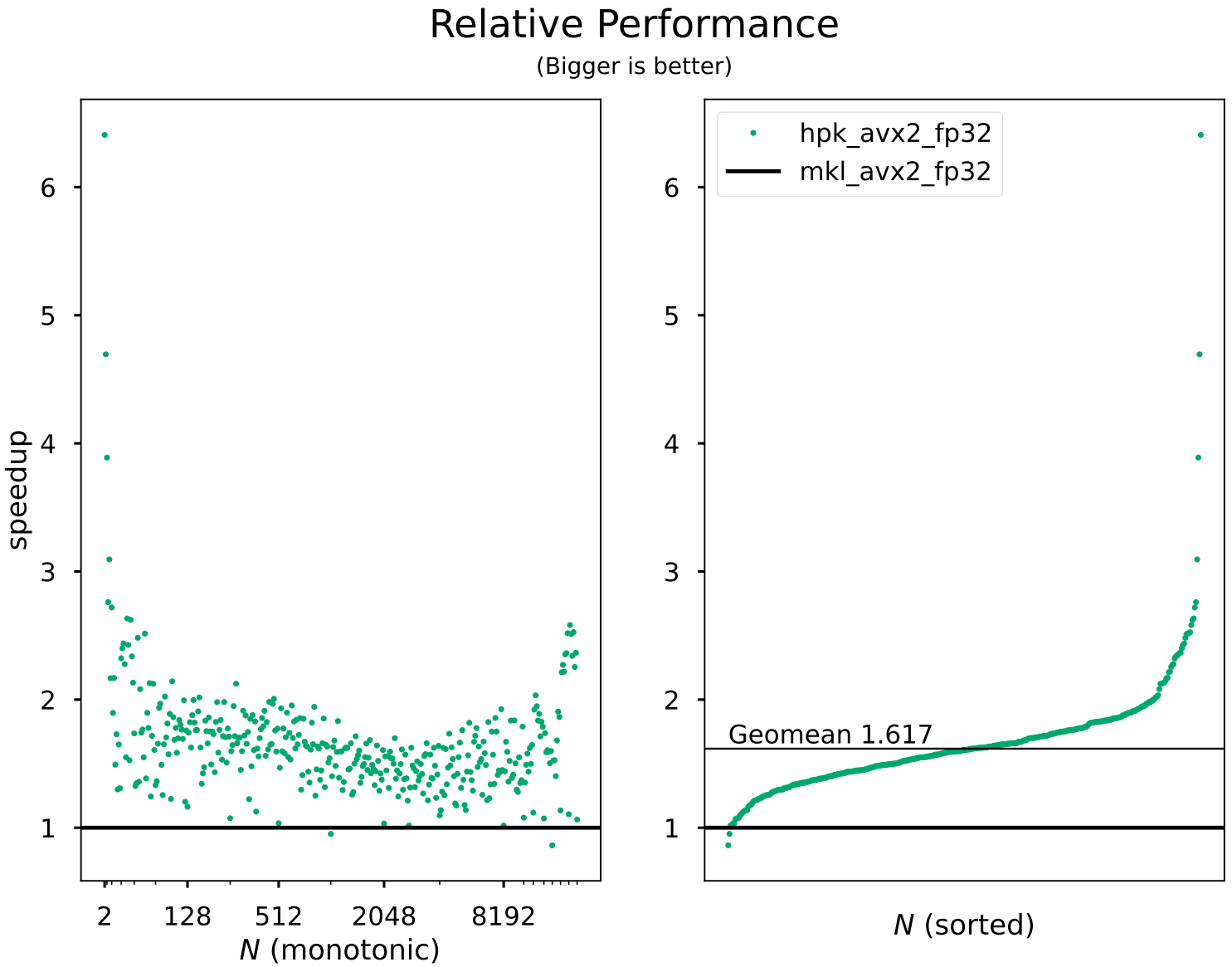
AVX512 Half Precision, Batch 7, vs. IPP



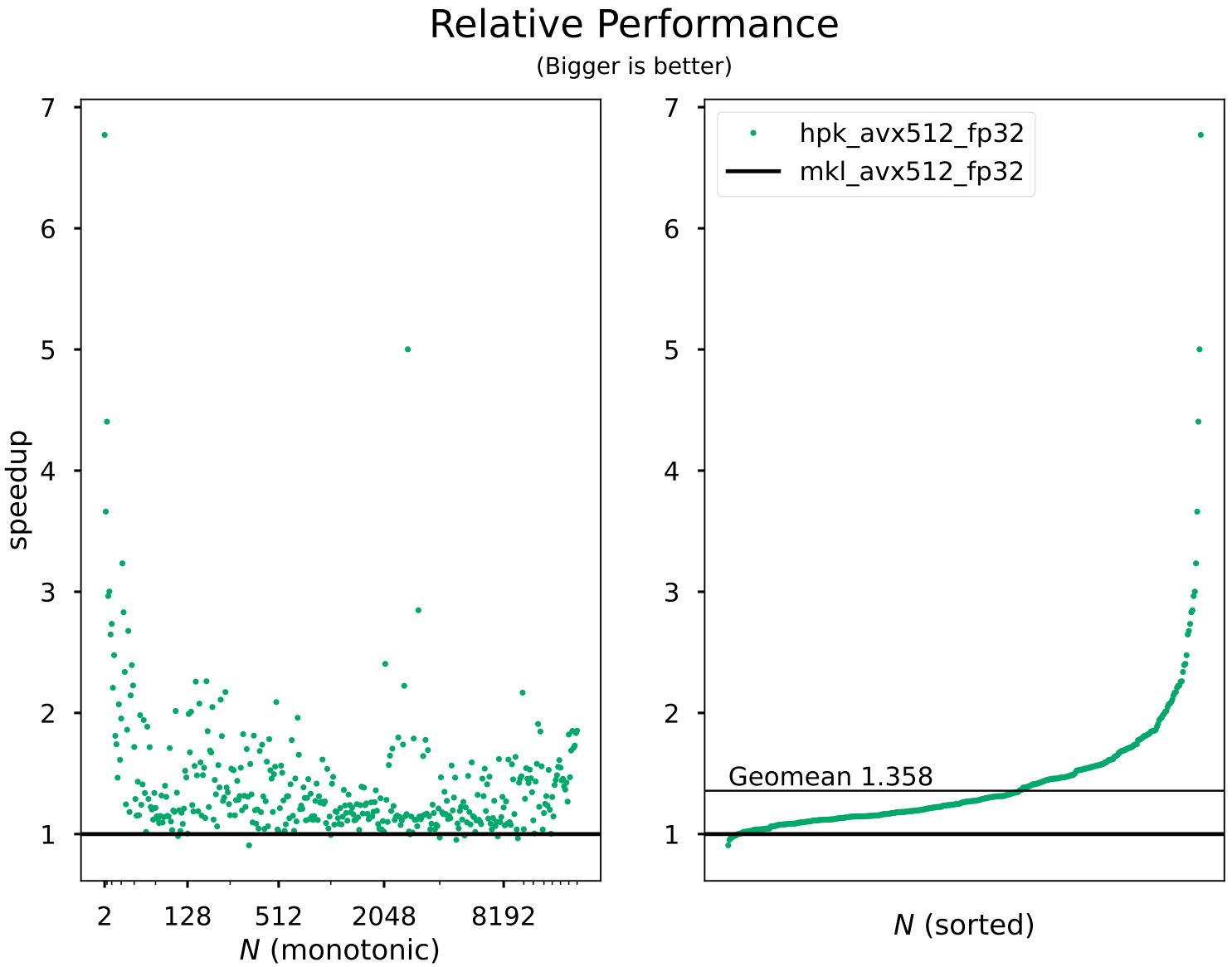
AVX2 Single Precision vs. FFTW



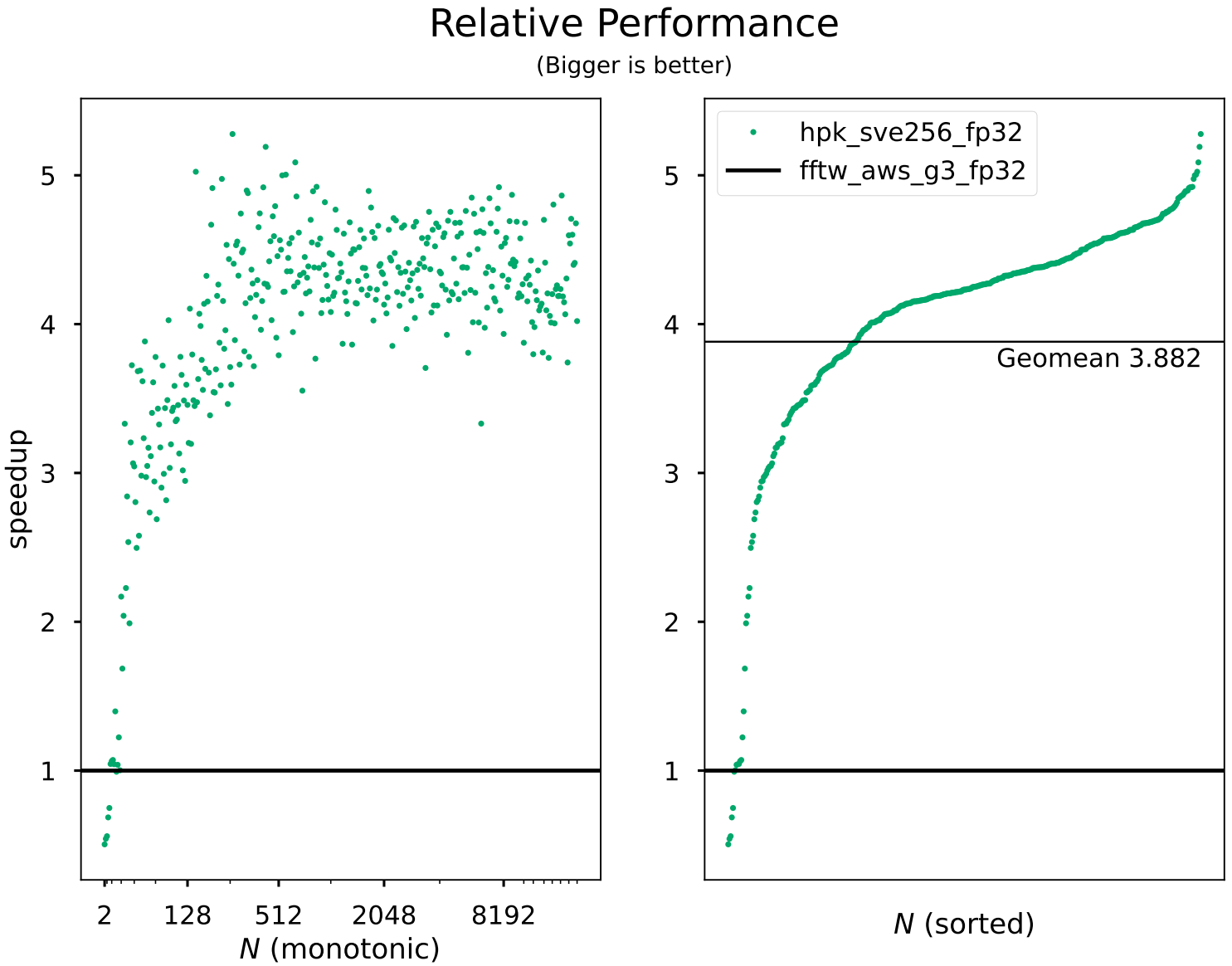
AVX2 Single Precision vs. MKL



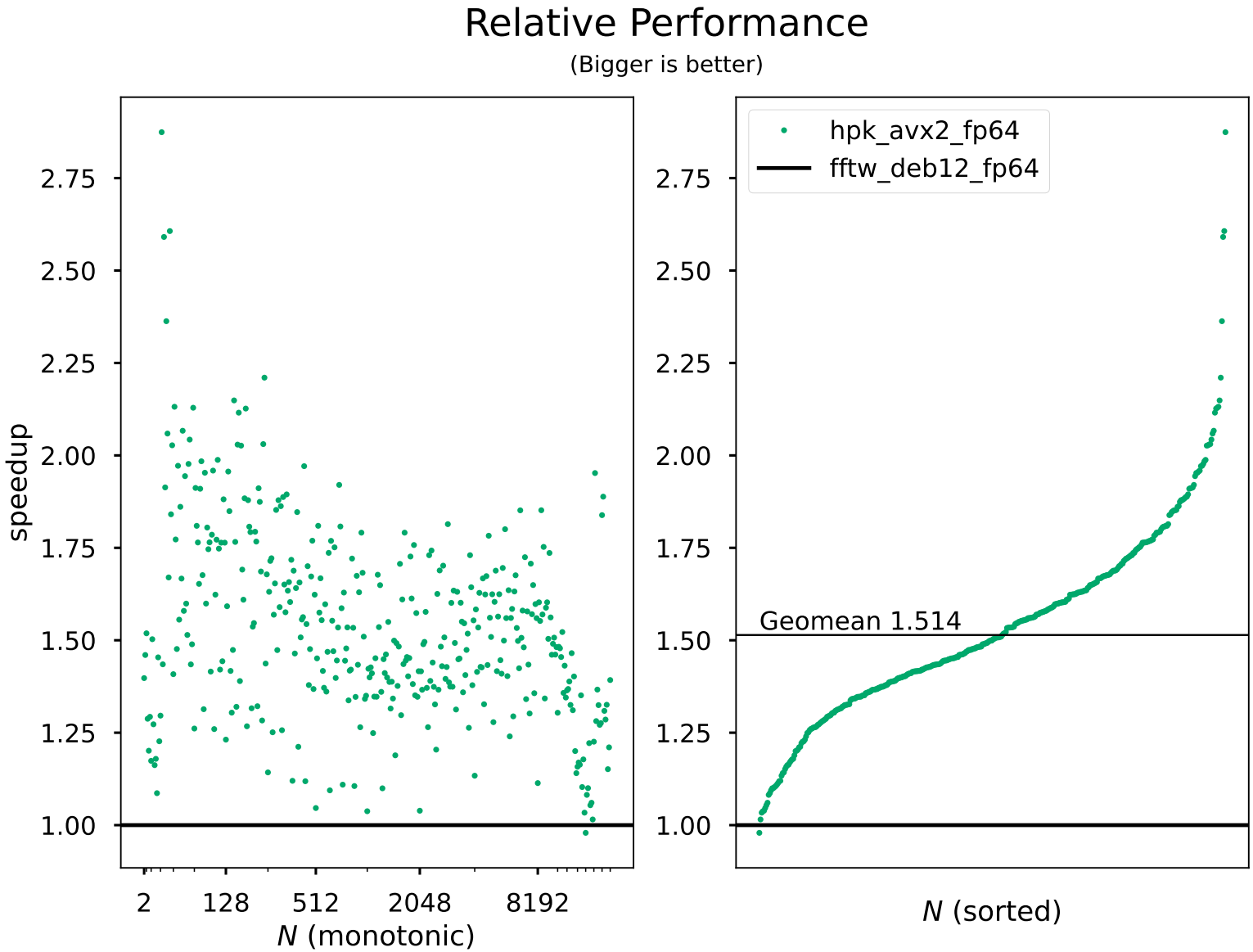
AVX512 Single Precision vs. MKL



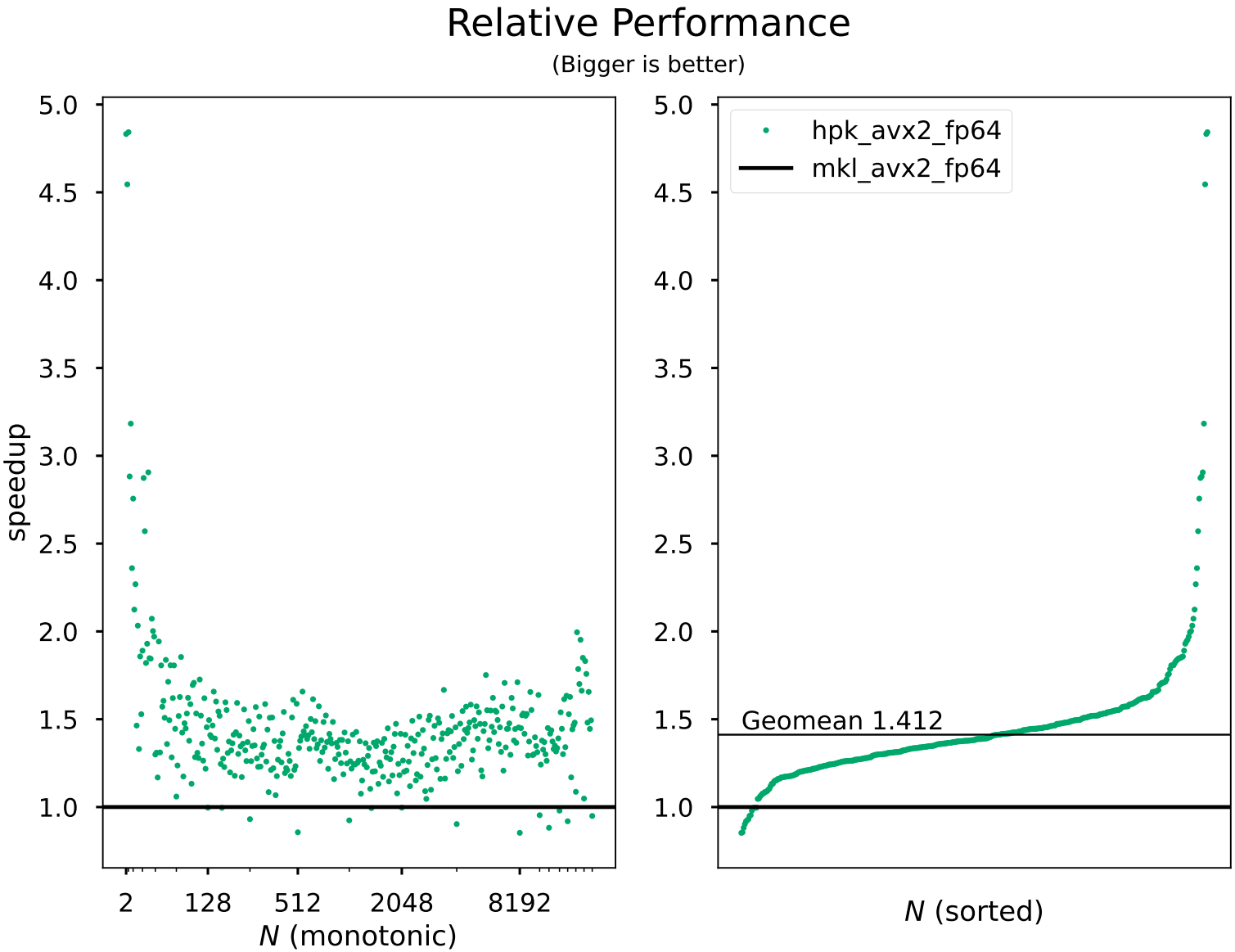
SVE256 Single Precision vs. FFTW



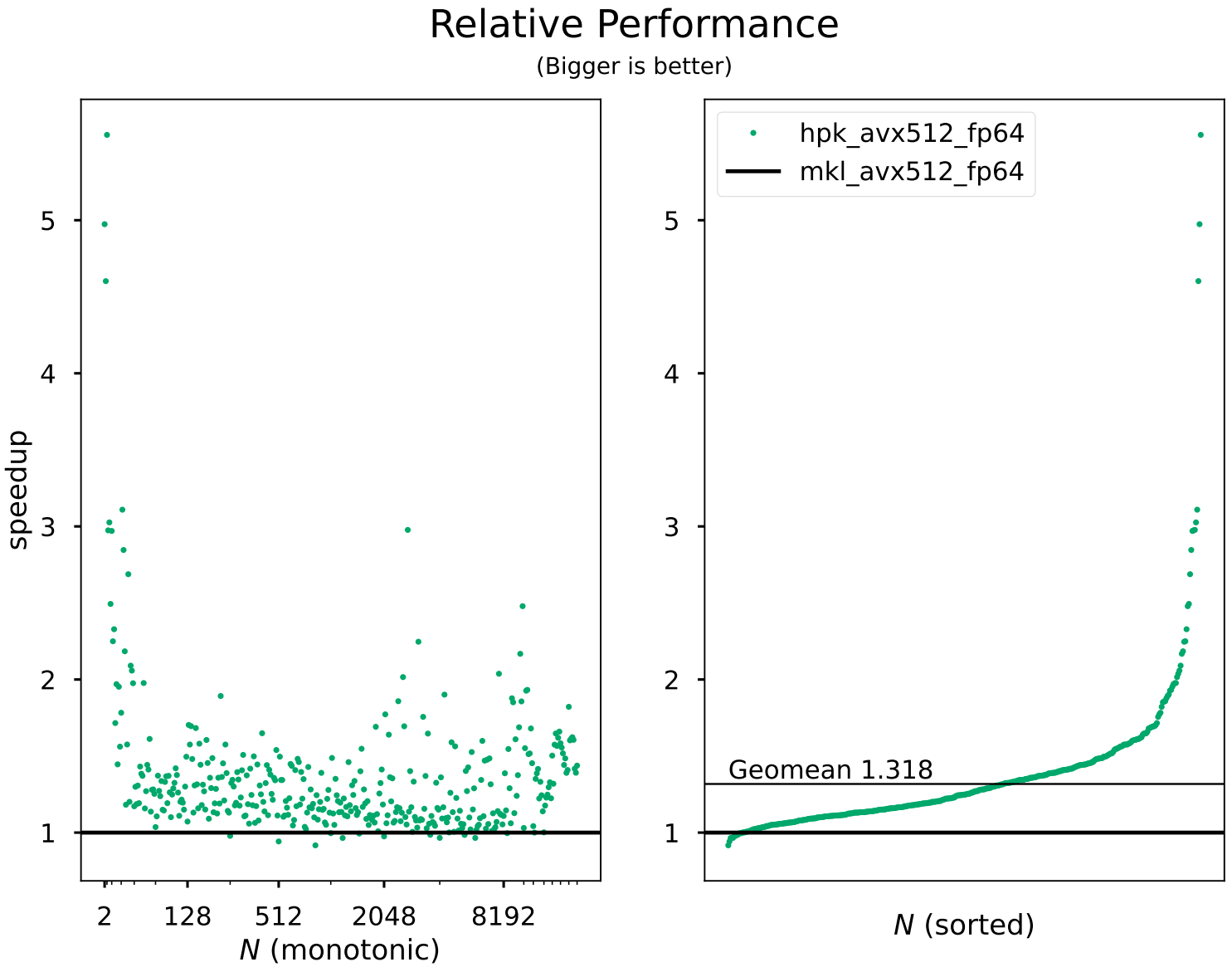
AVX2 Double Precision vs. FFTW



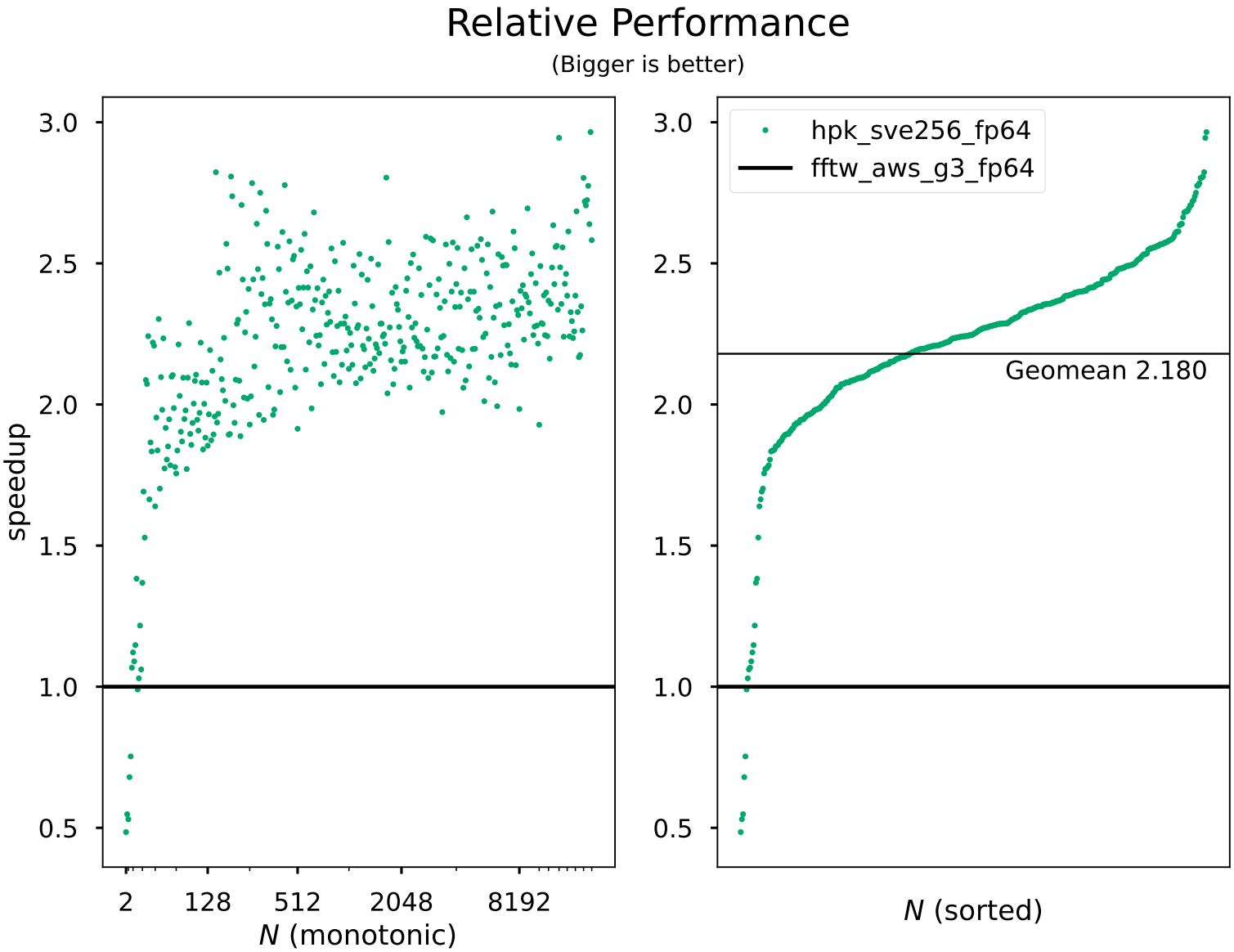
AVX2 Double Precision vs. MKL



AVX512 Double Precision vs. MKL



SVE256 Double Precision vs. FFTW




Слава Україні!
Героям слава!