The Journal of Open Source Software

kifmm-rs: A Kernel-Independent Fast Multipole Framework in Rust

Srinath Kailasa [1]

1 Department of Mathematics, University College London, UK

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

We present `kifmm-rs` a Rust based implementation of the kernel independent Fast Multipole Method (kiFMM) with C bindings, that serves as a implementation framework for kiFMMs (Greengard & Rokhlin, 1987; Ying et al., 2004). The FMM is a key algorithm for scientific computing due to its acceleration of the computation of N-body potential evaluation problems of the form,

$$\phi(x_i) = \sum_{j=1}^{N} K(x_i, y_j) q(y_j) \tag{1}$$

from $O(N^2)$ to $O(N)$ or $O(N \log(N))$, where the potential ϕ is evaluated at a set of target points, $\{x_i\}_{i=1}^{M}$, due to a set of densities, $\{q_j\}_{j=1}^{N}$ and $K(.,.)$ is the interaction kernel. Compatible kernels commonly arise in science and engineering, such as the Laplace kernel which models the electrostatic or gravitational potentials corresponding to a cluster of source points on a cluster of target points,

$$K(x, y) = \begin{cases} \frac{1}{2\pi} \log(\frac{1}{\|x-y\|}), & (2D) \\ \frac{1}{4\pi\|x-y\|}, & (3D) \end{cases} \tag{2}$$

FMMs split (1) for a given cluster of target points into *near* and *far* components, the latter of which are taken to be amenable to approximation,

$$\phi(x_i) = \sum_{y_j \in \text{Near}(x_i)} K(x_i, y_j) q_j + \sum_{y_j \in \text{Far}(x_i)} K(x_i, y_j) q_j \tag{3}$$

The evaluation of the near field component is done directly and referred to as the point to point (P2P) operation. The far-field component is compressed via a *field translation*, referred to as the multipole to local (M2L) operation. This split, in addition to a recursive loop through a hierarchical data structure used to discretise the problem, gives rise to the complexity of the FMM. These two operations dominate runtimes in practical implementations, and commonly the focus of implementation optimisations.

Statement of need

Previous high-performance codes for computing kiFMMs include (Malhotra & Biros, 2015; Wang et al., 2021). Both of these efforts are provided as templated C++ libraries with optimisations specialised for x86 architectures. Notably, neither softwares is well optimised for Arm targets which are becoming more common as both commodity and HPC platforms.

Our principle contributions with `kifmm-rs` are:

- A *highly portable* Rust-based data-oriented software design that allows us to easily test the impact of different algorithmic approaches and computational backends, such as BLAS libraries, for critical algorithmic sub-components as well as deploy to different architectures enabled with Rust's LLVM based compiler. We present the software for shared memory, with plans for distributed memory extension.
- *Competitive* single-node performance, especially in single precision, enabled by the optimisation of BLAS based M2L field translation, based entirely on level 3 operations with high arithmetic intensity that are well suited to modern hardware architectures that prioritise minimal memory movement per flop.
- The ability to *process multiple sets of source densities* corresponding to the same point distribution using (1), a common application in the Boundary Element Method.
- *A C API*, using Rust's C ABI compatibility allowing for the construction of bindings into other languages, with full Python bindings for non-specialist users.

A full API description is available as a part of published documentation. Both Python and Rust examples can be found in the repository.

Software design

Rust traits are contracts between types, and types can implement multiple traits. We are able to compose complex polymorphic behaviour for our data structures, which consist of simple structs of arrays, by writing all interfaces using traits. Enabling us to compose sub-components of our software, such as field translation algorithms, explicit SIMD vectorisation strategies for different architectures (via the Pulp library (Kazdadi, 2024)), tree datastructures, interaction kernels and underlying BLAS or LAPACK implementations (via the RLST library (Betcke, 2024)). This makes our software more akin to a framework for developing kiFMMs, which can take different flavours, and be used to explore the efficacy of different FMM approaches across hardware targets and software backends.

Benchmarks

We benchmark our software against leading implementations on a single node (Malhotra & Biros, 2015; Wang et al., 2021) in Figure (1) for the x86 architecture in Table (1) for achieving relative errors, ϵ, of 1×10^{-11} in double precision and 1×10^{-4} in single precision with respect to the direct evaluation of potential for points contained in a given box for a benchmark problem of computing (1) for the three dimensional Laplace kernel (2) for problem sizes between 100,000 and 1,000,000 uniformly distributed source and target points, which are taken to be the same set. Best parameter settings are described in the Appendix of (Kailasa et al., 2024). We repeat the benchmark for the Arm architecture for `kifmm-rs` in Figure (2), presented without comparison to competing software due to lack of support.

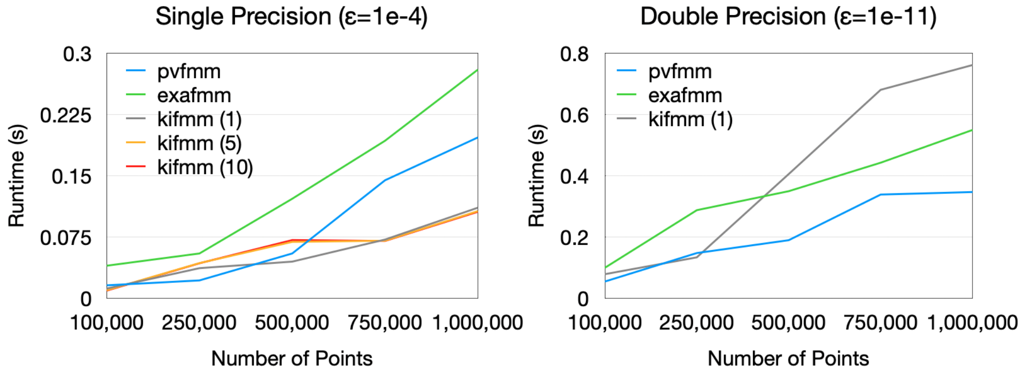

Figure 1: X86 benchmarks against leading kiFMM software for achieving relative error ϵ, for `kifmm-rs` the number of sets of source densities being processed is given in brackets, and runtimes are then reported per FMM call.

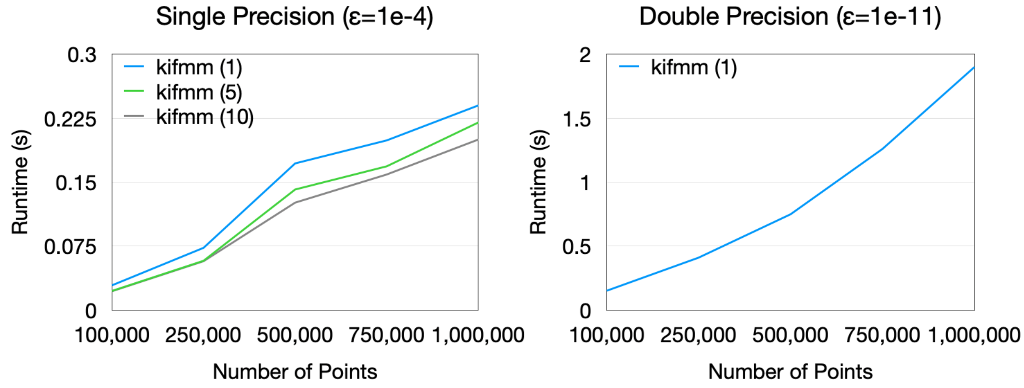

Figure 2: Arm benchmarks for achieving relative error ϵ, for `kifmm-rs` the number of sets of source densities being processed is given in brackets, and runtimes are then reported per FMM call.

Table 1: Hardware and software used in our benchmarks. We report per core cache sizes for L1/L2 and total cache size for L3.

	Apple M1 Pro	**AMD 3790X**
Cache Line Size	128 B	64 B
L1i/L1d	192/128 KB	32/32 KB
L2	12 MB	512 KB
L3	12 MB	134 MB
Memory	16 GB	252 GB
Max Clock Speed	3.2 GhZ	3.7 GhZ
Sockets/Cores/Threads	1/8/8	1/32/64
Architecture	Arm V8.5	x86
BLAS	Apple Accelerate	Open BLAS
LAPACK	Apple Accelerate	Open BLAS
FFT	FFTW	FFTW
Threading	Rayon	Rayon
SIMD Extensions	Neon	SSE, SSE2, AVX, AVX2

Acknowledgements

Srinath Kailasa is supported by EPSRC Studentship 2417009

References

Betcke, T. (2024). *RLST: A linear algebra library in rust*. https://github.com/linalg-rs/rlst; GitHub.

Greengard, L., & Rokhlin, V. (1987). A fast algorithm for particle simulations. *Journal of Computational Physics*, *73*(2), 325–348.

Kailasa, S., Betcke, T., & El-Kazdadi, S. (2024). M2L translation operators for kernel independent fast multipole methods on modern architectures. *Submitted To SIAM Journal on Scientific Computing*.

Kazdadi, S. E. (2024). *Pulp: A safe abstraction over SIMD instructions*. https://github.com/sarah-ek/pulp; GitHub.

Malhotra, D., & Biros, G. (2015). PVFMM: A parallel kernel independent FMM for particle and volume potentials. *Communications in Computational Physics*, *18*(3), 808–830.

Wang, T., Yokota, R., & Barba, L. A. (2021). ExaFMM: A high-performance fast multipole method library with c++ and python interfaces. *Journal of Open Source Software*, *6*(61), 3145.

Ying, L., Biros, G., & Zorin, D. (2004). A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics*, *196*(2), 591–626.