




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](https://doi.org/10.xxxxxx/draft)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Open Journals](#) 

Reviewers:

- [@openjournals](#)

Submitted: 01 January 1970

Published: unpublished

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

Summary

We present kifmm-rs a Rust based implementation of the kernel independent Fast Multipole Method (kiFMM), with Python bindings, that serves as a implementation framework for implementing the kiFMMs ([Greengard & Rokhlin, 1987](#); [Ying et al., 2004](#)). The FMM is a core algorithm for scientific computing, commonly cited as one of the top algorithmic advances of the twentieth century ([Cipra, 2000](#)) due to its acceleration of the computation of N -body potential evaluation problems of the form,

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

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 $K(\cdot, \cdot)$ is the interaction kernel. Compatible kernels commonly arise from second-order elliptic partial differential equations, such as the Laplace kernel which models the electrostatic or gravitational potentials corresponding to a set of source points on a set of target points,

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

The FMM also finds a major application in the acceleration of Boundary Element Methods (BEM) for elliptic boundary value problems ([Steinbach, 2007](#)), which can be used to model a wide range of natural phenomena. Kernel independent variants of the FMM (kiFMMs) replace the analytical series approximations used to compress far-field interactions between clusters of source and target points, with approximation schemes that are based on kernel evaluations and extensions of the algorithm can also handle oscillatory problems specified by the Helmholtz kernel ([Engquist & Ying, 2010](#); [Ying et al., 2004](#)), with a common underlying algorithmic and software machinery that can be optimised in a kernel-independent manner.

The FMM splits (1) for a given target cluster into *near* and *far* components, the latter of which are taken to be *admissible*, i.e. amenable to approximation via an expansion scheme or alternative interpolation method such as ([Fong & Darve, 2009](#); [Ying et al., 2004](#)),

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

the near component evaluated directly using the kernel function $K(\cdot, \cdot)$, and the far-field compressed via a *field translation*, referred to as the multipole to local (M2L) translation. This split, in addition to a recursive loop through a hierarchical data structure, commonly an octree

for three dimensional problems, gives rise to the linear/log-linear complexity of the FMM, as the number of far-field interactions which are admissible is limited by a constant depending on the problem dimension. The evaluation of the near field component commonly referred to as the particle to particle (P2P) operation. These two operations conspire to dominate runtimes in practical implementations. An approximate rule of thumb being that the P2P is compute bound, and the M2L is memory bound, acceleration attempts for FMM softwares often focus on reformulations that ensure the M2L has a high arithmetic intensity.

Statement of need

Previous high-performance codes for computing kiFMMs include (Malhotra & Biros, 2015; Wang et al., 2021). However, both of these efforts are heavily templated C++ libraries, with brittle optimisations for the M2L and P2P operations that make it complex for users or new developers to exchange or experiment with new algorithmic or implementation ideas that improve runtime performance. It is not possible to readily deploy these softwares on new hardware platforms due to reliance on hand written CMake based builds. Notably, neither softwares support building to Arm targets which are becoming more common as both commodity and HPC platforms, with no performance promised for systems which don't support AVX instruction sets.

Novice users are provided with Python bindings in (Wang et al., 2021), however the state of the art distributed code is only accessible via C++ (Malhotra & Biros, 2015). In both softwares, sub-components such as the octree data structures and kernel implementations are not readily re-usable for related algorithmic work by downstream users, and underlying software used in compute kernels such as libraries for BLAS, LAPACK, or the FFT are not readily exchangeable for experimentation with performance differences across hardwares.

Our principle contributions with kifmm-rs that extend beyond current state of the art implementations 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 such as the M2L and P2P operations.
- *State of the art* single-node performance 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 current and future hardware architectures that prioritise minimal memory movement per flop.
- The ability to *process multiple right hand sides* corresponding to the same particle distribution using (1), a common application in BEM.
- *Simple API*, with full Python bindings for non-specialist users. For basic usage all users need to specify are source and target coordinates, and associated source densities, with no temporary files.

kifmm-rs is a core dependency for the BEM library bempp-rs (Betcke & Scroggs, 2024), and we present a detailed exposition behind the algorithmic and implementation approach in (Kailasa et al., 2024). Currently limited to shared memory systems, distributed memory extensions are an area of active development.

Software design

Rust

As a platform for scientific computing, Rust's principal benefits are its build system Cargo enabling builds with as little as one terminal command from a user's perspective with dependencies specified in a modern TOML style, and its single centrally supported LLVM based

78 compiler `rustc` that ensures consistent cross-platform performance. Compiled Rust code is
 79 compatible with the C application binary interface (ABI), which enables linking the extensive
 80 existing scientific ecosystem, and is indeed supported through Cargo's `build.rs` files. This
 81 also makes it easy to create language bindings for Rust projects from Python, C++, Fortran
 82 and other languages.

83 Data oriented design with traits

84 Rust's 'trait' system effectively allows us to write our code towards the 'data oriented design'
 85 paradigm. Which places optimal memory movement through cache hierarchies at the centre of
 86 a software's design. Resultantly, 'structs of arrays' are preferred to 'arrays of structs' -
 87 i.e. simple objects that wrap contiguously stored buffers in comparison to complex objects
 88 storing complex objects.

89 Traits are contracts between types, and types can implement multiple traits. Therefore we
 90 are able to compose complex polymorphic behaviour for our data structures, which consist
 91 of simple structs of arrays, by writing all interfaces using traits. In this way, we are able to
 92 easily compose sub-components of our software, such as field translation algorithms, explicit
 93 SIMD vectorisation strategies for different architectures (via the Pulp library (Kazdadi, 2024)),
 94 single node and MPI distributed octrees, interaction kernels and underlying BLAS or LAPACK
 95 implementations (via the RLST library (Bettcke, 2024)). This makes our software more akin to
 96 a framework for developing kiFMMs, which can take different flavours, and be used to explore
 97 the efficacy of different FMM approaches across hardware targets and software backends.

98 API

99 Our Rust APIs are simple in comparison to other leading codes, with the requirement for
 100 no temporary metadata files (Wang et al., 2021), or setup of ancillary data structures such
 101 as hierarchical trees (Malhotra & Biros, 2015), required by the user. FMMs are simply
 102 parameterised using the builder pattern, with operator chaining to modulate the runtime
 103 object. At its simplest, a user only specifies buffers associated with source and target particle
 104 coordinates, and associated source densities. Trait interfaces implemented for FMM objects
 105 allows users to access the associated objects such as PDE kernels and data such as multipole
 106 expansions.

```
use rand::{thread_rng, Rng};
use green_kernels::{laplace_3d::Laplace3dKernel, types::EvalType};
use kifmm::{BlasFieldTranslationSaRcmp, SingleNodeBuilder};
use kifmm::traits::tree::{FmmTree, Tree};
use kifmm::traits::fmm::Fmm;

fn main() {
    // Generate some random source/target/charge data
    let dim = 3;
    let nsources = 1000000;
    let ntargets = 2000000;

    // The number of right hand sides, FMM is configured from data
    let nrhs = 1;
    let mut rng = thread_rng();
    let mut sources = vec![0f32; nsources * dim * nrhs];
    let mut targets = vec![0f32; ntargets * dim * nrhs];
    let mut charges = vec![0f32; nsources * nrhs];

    sources.iter_mut().for_each(|s| *s = rng.gen());
    targets.iter_mut().for_each(|t| *t = rng.gen());
```

```

charges.iter_mut().for_each(|c| *c = rng.gen());

// Set tree parameters
// Library refines tree till fewer than 'n_crit' particles per leaf box
let n_crit = Some(150);
// Alternatively, users can specify the tree depth they require
let depth = None;
// Choose to remove empty leaves
let prune_empty = true;

// Set FMM Parameters
// Can either set globally for whole tree, or level-by-level
let expansion_order = &[6];
// Parameters which control speed and accuracy of BLAS based field translation
let singular_value_threshold = Some(1e-5);
let check_surface_diff = Some(2);

// Create an FMM
let fmm = SingleNodeBuilder::new()
    .tree(&sources, &targets, n_crit, depth, prune_empty) // Create tree
    .unwrap()
    .parameters(
        &charges,
        expansion_order, // Set expansion order, by tree level or globally
        Laplace3dKernel::new(), // Choose kernel,
        EvalType::Value, // Choose potential or potential + deriv evaluation
        BlasFieldTranslationSaRcmp::new(
            singular_value_threshold,
            check_surface_diff
        ), // Choose field translation
    )
    .unwrap()
    .build()
    .unwrap();

// Run FMM
let times = fmm.evaluate(true); // Optionally time the operators

// Lookup potentials by leaf from target leaf boxes
let leaf_idx = 0;
let leaf = fmm.tree().target_tree().all_leaves().unwrap()[leaf_idx];
let leaf_potential = fmm.potential(&leaf);
}

```

107 Indeed, the full API is more extensive, including features that enable for variable expansion
 108 orders by level useful for high-frequency problems, accelerated pre-computations for the BLAS
 109 based field translations based on randomised SVDs, alternative field translation implementations,
 110 data visualisation with MayaVi and methods for file IO. Both [Python](#) and [Rust](#) examples can
 111 be found in the repository.

112 Benchmarks

113 We benchmark our software against other leading implementations on a single node ([Malhotra](#)
 114 [& Biro, 2015](#); [Wang et al., 2021](#)) for the target architectures in Table (1) for achieving a
 115 given precision for a common benchmark problem of computing (1) for the three dimensional

Laplace kernel (2) for one million uniformly distributed source and target points. Optimal parameters were calculated for this setting using a grid search, the results of which can be found in Appendix A of (Kailasa et al., 2024). We illustrate our software performance using two common acceleration schemes for the field translation, FFT and BLAS level 3 operations, only the former of which are supported by current state of the art implementations.

[Space for Plot]

Table 1: Hardware and software used in our benchmarks, for the Apple M1 Pro we report only the specifications of its 'performance' CPU cores. 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	BLIS
LAPACK	Apple Accelerate	Netlib
FFT	FFTW	FFTW
Threading	Rayon	Rayon

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/rst>; GitHub.

Betcke, T., & Scroggs, M. (2024). *Bempp-rs: Boundary element methods in rust*. <https://github.com/bempp/bempp-rs>

Cipra, B. A. (2000). The best of the 20th century: Editors name top 10 algorithms. *SIAM News*, 33(4), 1–2.

Engquist, B., & Ying, L. (2010). Fast directional algorithms for the helmholtz kernel. *Journal of Computational and Applied Mathematics*, 234(6), 1851–1859.

Fong, W., & Darve, E. (2009). The black-box fast multipole method. *Journal of Computational Physics*, 228(23), 8712–8725. <https://doi.org/10.1016/j.jcp.2009.08.031>

Greengard, L., & Rokhlin, V. (1987). A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2), 325–348. [https://doi.org/10.1016/0021-9991\(87\)90140-9](https://doi.org/10.1016/0021-9991(87)90140-9)

Kailasa, S., Betcke, T., & El-Kazdadi, S. (2024). Designing kernel independent fast multipole methods for 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.

- 141 Malhotra, D., & Biros, G. (2015). PVFMM: A Parallel Kernel Independent FMM for Particle
142 and Volume Potentials. *Commun. Comput. Phys.*, 18(3), 808–830. <https://doi.org/10.4208/cicp.020215.150515sw>
143
- 144 Steinbach, O. (2007). *Numerical approximation methods for elliptic boundary value problems: Finite and boundary elements*. Springer Science & Business Media.
145
- 146 Wang, T., Yokota, R., & Barba, L. A. (2021). ExaFMM: A high-performance fast multipole
147 method library with c++ and python interfaces. *Journal of Open Source Software*, 6(61),
148 3145.
- 149 Ying, L., Biros, G., & Zorin, D. (2004). A kernel-independent adaptive fast multipole
150 algorithm in two and three dimensions. *Journal of Computational Physics*, 196(2), 591–626.
151 <https://doi.org/10.1016/j.jcp.2003.11.021>

DRAFT