FMMTL: FMM Template Library
A Generalized Framework for Kernel Matrices

Cris Cecka[1], Simon Layton[2]

[1] Harvard University, Cambridge MA, ccecka@seas.harvard.edu
[2] Boston University, Boston MA, slayton@bu.edu

Abstract. In response to two decades of development in structured dense matrix algorithms and a vast number of research codes, we present designs and progress towards a codebase that is abstracted over the primary domains of research. In the domain of mathematics, this includes the development of interaction kernels and their low-rank expansions. In the domain of high performance computing, this includes the optimized construction, traversal, and scheduling algorithms for the appropriate operations. We present a versatile system that can encompass the design decisions made over a decade of research while providing an abstracted, intuitive, and usable front-end that can integrated into existing linear algebra libraries.

1 Introduction

Structured dense matrices arise in a broad range of engineering applications including the discretization of Fredholm integral equations, boundary elements methods, N-body problems, signal processing, statistics, and machine learning. This research concerns kernel matrix equations of the form

$$r_i = \sum_j K(t_i, s_j)\, c_j \tag{1}$$

where we refer to K as the *kernel* generating elements of the dense matrix, the s_j as *sources* and c_j as a source's associated *charge*, and the t_i as *targets* and r_i as a target's associated *result*. Of course, a direct computation of a kernel matrix-vector product requires $\mathcal{O}(N^2)$ computations, where N is the cardinality of the source set and target sets. Fast multipole methods (FMMs) and tree-codes allow for an approximate evaluation of this matrix-vector product with only $\mathcal{O}(N \log^\alpha N)$ complexity, where $\alpha = 0, 1$, or 2 depending on specifics regarding the kernel, the expansions, the traversal algorithms, and the distribution of sources and targets. Common kernels in physics and statistics and their domains and ranges are listed in Table 1.

Name/Equation	$K(x, y)$	Domain	Range
Laplace, Poisson	$1/\lvert\mathbf{x}-\mathbf{y}\rvert$	$\mathbb{R}^3 \times \mathbb{R}^3$	$\mathbb{R}$
Yukawa, Helmholtz	$e^{k\lvert\mathbf{x}-\mathbf{y}\rvert}/\lvert\mathbf{x}-\mathbf{y}\rvert$	$\mathbb{R}^3 \times \mathbb{R}^3$	$\mathbb{C}$
Stokes	$\frac{1}{\lvert\mathbf{x}-\mathbf{y}\rvert}\left(\mathbf{I} + \frac{(\mathbf{x}-\mathbf{y})(\mathbf{x}-\mathbf{y})^T}{\lvert\mathbf{x}-\mathbf{y}\rvert^2}\right)$	$\mathbb{R}^3 \times \mathbb{R}^3$	$\mathbb{R}^{3,3}$
Gaussian	$e^{-\varepsilon\lvert\mathbf{x}-\mathbf{y}\rvert^2}$	$\mathbb{R}^n \times \mathbb{R}^n$	$\mathbb{R}$
Multiquadric	$(1 + \lvert\mathbf{x}-\mathbf{y}\rvert^2)^{\pm 1/2}$	$\mathbb{R} \times \mathbb{R}$	$\mathbb{R}$

Table 1. A table of common kernels emphasizing the varied domain and ranges of the operators.

FMMs require multiple, carefully optimized steps and numerical analysis in order to achieve the improved asymptotic performance and required accuracy. These research areas span tree construction, tree traversal, numerical and functional analysis, and complex heterogeneous parallel computing strategies for each stage. Unfortunately, many FMM codes are written with a particular application (an interaction kernel and/or compute environment) in mind [4,10,2]. It is often difficult to extract out advances from one research area and apply them to another code or application. Indeed, Yokota et. al. [13] discuss recent developments and comparisons to note the disappointing lack of fair benchmarking comparisons between kernel expansions, data structures, traversal algorithms, and parallelization strategies.

In this paper, we review recent development of a parallel, generalized framework and repository for kernel matrices of the form (1). This overarching goal of this library, called FMMTL, is to separate academic concerns in research and development of fast structured dense matrix algorithms. Using advanced C++ techniques and design, we are able to develop the code at a high level, isolate development hurdles and choices, and collect a repository of kernels and their associated expansions for rapid application deployment in any of the above domain areas. This is accomplished by defining generalized interfaces for kernels independent of algorithmic concerns with tree construction and traversal and presenting a coherent front-end for working with kernel matrices as abstract data types. Problems of the form 1 can be constructed and manipulated in an intuitive way and should be able to take advantage of existing solvers or provide their own. The library has already seen use in simple Poisson problems, more advanced boundary element solvers, and the use of FMM as a preconditioner.

2 Background

Fast methods for kernel matrices define or compute a low-rank approximation to the kernel valid for some set of the sources, S, and targets, T:

$$K(T, S) \approx U(T)\, \widetilde{K}\, V^T(S).$$

These approximations can be computed analytically with series expansion or interpolations of the known kernel function [4,5,7] or algebraically by rank-reducing operations on samples of the kernel function [6,12,11]. This allows off-diagonal blocks of the kernel matrix to be approximated and computed quickly and defines the class of Hierarchically Off-Diagonally Low-Rank (HODLR) matrices. The fundamental operations used in working with HODLR matrices are:

S2M: $M = V^T(S) * C$ M2L: $L = \widetilde{K} * M$ L2T: $R \approx U(T) * \widetilde{K}$

where R are the results associated with the targets T, C are the charges associated with sources S, and we call M a *multipole expansion* and L a *local expansion*.

Hierarchically SemiSeparable matrices (HSS) allow the multipoles of sets of sources to be computed from the multipoles of subsets to form a hierarchy

of low-rank approximations. The operations involved are extended to include:

$$\text{M2M: } M' = \widetilde{V}^T * M \qquad \text{L2L: } L' = \widetilde{U} * L$$

Convenient operators to add to this pool are found in most often in tree-codes and can be written as:

$$\text{S2L: } L = \widetilde{K} * V^T(S) * c \qquad \text{M2T: } R \approx U * \widetilde{K} * M$$

Finally, the sets of sources and targets whose block in the kernel matrix is approximated in this way are chosen with a rule called the multipole acceptance criteria (MAC). Whether this rule accepts "nearby clusters" of sources and targets differentiates $\mathcal{H}$ matrices from HOLDR matrices and $\mathcal{H}^2$ matrices from HSS matrices.

In practice, these operations are often built into research implementations and can be difficult to extract, understand, and modify. It is these operators that we wish to classify and fully abstract in order to develop a library that can be used, without modification, for any kernel matrix and any definition of the above operators. Additional goals of FMMTL are to isolate algorithmic features such as tree construction and traversal that are also too often entangled with problem-specific data or algorithms.

3 Design Considerations

The design of FMMTL attempts to make kernels and their expansions independent citizens in that their implementation should not depend on or know about trees, clusters of sources or targets, traversals, or parallelism. Furthermore, it should be portable and easy to use and install. For this reason, the only dependency is a modern C++ compiler with C++11 support and the renowned C++ Boost library (headers only). Additionally, if CUDA is installed and available, the library will attempt to use GPU acceleration.

In this section, we offer a brief overview of the features and design considerations in FMMTL.

3.1 Kernels

Kernels are simply function objects used to generate elements of the matrix, but should also define the domain of the problem. The fundamental types required are the domain of the kernel (the `source_type` and the `target_type`) and the range of the kernel (the `kernel_value_type`).

In Listing 1, `Vec` is a statically sized abstract vector type designed to work on multiple architectures. In addition, note the transpose method labeled optional. In many cases, the kernel satisfies a symmetry property

$$K(s,t) = \mathcal{T} \circ K(t,s)$$

that can be computed much more efficiently than evaluation of the kernel and may be used to accelerate the computation in the case that the source and target sets are the same. The library uses advanced SFINAE – Substitution Failure Is Not An Error – compiler techniques to statically detect whether this optional method is defined at compile time and will use it if appropriate.

```
struct MyKernel : public fmmtl::Kernel<MyKernel> {
  typedef Vec<3,double> source_type;
  typedef Vec<3,double> target_type;
  typedef double kernel_value_type;

  FMMTL_INLINE kernel_value_type
  operator()(const target_type& t, const source_type& s) const {
    return norm(s-t);
  }
  /** Optional transpose operation for optimization **/
  FMMTL_INLINE kernel_value_type
  transpose(const kernel_value_type& kts) const {
    return kts;
  }
};
```

Listing 1. Example kernel typdefs and members

3.2 Expansions

An expansion is the low-rank approximation of a kernel that can be used to accelerate the kernel matrix operations. The primary role of expansions is to declare the type of the multipole and local objects and provide methods for transforming between the `source_type`, `target_type`, `multipole_type`, and `local_type`. All possible conversions are shown in Figure 1.

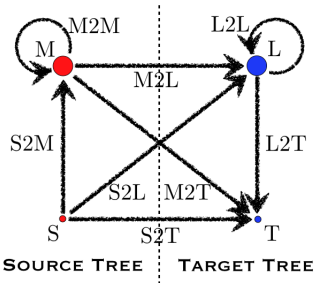

Fig. 1. The "algebraic system" of hierarchical methods is composed of sources S, multipole expansions M, local expansions L, and targets T. The S2T operation is defined by the kernel and all other operators may be defined by the expansion.

Nearly all of the functions in an expansion are optional and their availability is statically detected. This information can be used to determine computational pathways and potentially choose the most efficient. Additionally, this feature is attractive when expansions cannot define a certain operator or some traversal algorithms do not consider some operators. For example, the two primary types of tree-codes both use subsets of the operations shown in Figure 1. The particle-cluster tree-codes [3] use the S2M, M2M, and M2T operations while the cluster-particle tree-codes use the S2L, L2L, and L2T operations. Similarly, many fast multipole methods neglect the S2L and M2T operators while others are beginning to consider a larger set of operations to dynamically determine the cheapest computational pathway [14,8,1].

The expansion also defines a `point_type` which is used as the spacial embedding of the sources and targets for clustering and hierarchical constructions. Because `source_type` and `target_type` need only be convertible to this `point_type`, they are free to be much more complicated objects. For example, in boundary element methods, the source and target types are naturally triangles, patches, or basis functions. These may have a spacial center

that can be used for the construction of the tree, but should remain independent entities for simplifying the definition of the kernel function.

```
struct MyExpansion : public fmmtl::Expansion<MyKernel, MyExpansion> {
    /** Spacial type to provide an interpretation for clustering.
     * @note source_type and target_type must be either
     *   (1) convertible to point_type, or
     *   (2) S2P and/or T2P shall be defined to provide a conversion. */
    typedef Vec<3,double> point_type;

    typedef std::vector<double> multipole_type;
    typedef std::vector<double> local_type;

    /** Optional S2M Operator **/
    void S2M(const source_type& s, const charge_type& c,
             const point_type& center, multipole_type& M) const {
        // Compute M += V^T(s) * c
    }
    ...
};
```

Listing 2. Example expansion typdefs and members

3.3 Tree and traversals

The lightweight tree data structure is constructed on any `point_type`. Depending on the dimension D of the `point_type`, another compile-time constant, a D-dimensional binary tree ($D = 2$ is a quadtree, $D = 3$ is an octree, etc) is constructed. This is accomplished via partially sorting the points on a space-filling curve. However, this implementation detail is hidden behind an interface that any reasonable tree structure should provide, so alternate tree types and representations, such as a k-d tree, can be swapped in at will. A brief interface for a tree is given in Listing 3.

```
struct Tree {
  struct Box {
    unsigned index() const;
    body_iterator body_begin() const;
    body_iterator body_end() const;
    box_iterator child_begin() const;
    box_iterator child_end() const;
    Box parent() const;
  };
  struct Body {
    unsigned number() const;   // Original index this body was added
    unsigned index() const;    // Index within the tree
  };

  body_iterator body_begin() const;
  body_iterator body_end() const;
  box_iterator box_begin(int level) const;
  box_iterator box_end(int level) const;
};
```

Listing 3. A truncated interface for a general tree data structure

The tree need not store the points or the expansions as other implementations of hierarchical algorithms appear to. Instead, each box and point in the tree have an immutable identification index that can be used to manage arbitrary data outside of the data structure. This allows the tree structure

to be lightweight and allows a more context-aware data structure to manage source, target, and expansion data independent of the tree.

The traversals are implemented as a dual tree traversal and are templated on the `Box` type, requiring a `Box` to implement a small number of reasonable methods such as those in Listing 3. The dual tree traversal is often used in tree-codes, but not fast multipole methods. However, Yokota et. al. [13] take advantage of its versatility for hierarchical problems to generalize their codes. We would like to note that with a sufficiently generalized MAC, the dual tree traversal can produce the same interaction lists as the classic FMM, the adaptive UVWX schemes [12], and modern tree-codes. We find that there are two types of MAC: static MACs which depend only the the position and size of the boxes, and dynamic MACs which depend on the expansions or are otherwise dependent on the source and target distributions [9,13].

3.4 Optimizations

When an operator is determined to be required, the operation may be dispatched immediately or scheduled for later use and reuse. In general, making these choices has been an algorithmic option in the development of hierarchical methods. Recent studies have shown that event-driven parallel runtime systems provide success in dynamically resolving the dependencies within fast multipole methods and tree-codes. Ltaief & Yokota [8] use QUARK to schedule threads dynamically to accommodate the the data flow of exaFMM's dual tree traversal. Agullo et al. [1] apply a similar approach with StarPU to their black-box FMM using parallelism on both the CPU and the GPU. These approaches appear promising and could result in an efficient and easier to apply parallelism strategy than a statically implemented distributed algorithm.

FMMTL provides a generic way to implement kernels so that if a CUDA compiler is installed and a GPU is available, the library will use the GPU to accelerate the costly S2T computation. This requires an accumulation of interacting source and target sets and a compression of the interaction list to a form that is suitable for the GPU. Dynamic parallelization studies have found that the methods benefited the most from assigning the S2T operations exclusively to the GPU [1] due to the structured nature and high flop-to-byte ratio of S2T operations. The first major FMMTL parallelization step of executing a generalized S2T on the GPU is motivated by these results.

4 Usage

```
1  MyExpansion K(...);                 // Expansion order, error eps, etc
2  ..
3  std::vector<source_type> s = ...    // Define sources
4  std::vector<charge_type> c = ...    // Define charges
5  std::vector<target_type> t = ...    // Define targets
6
7  fmmtl::kernel_matrix<MyExpansion> M = K(t,s);            // Construct
8  fmmtl::set_options(M, opts);                             // Set options
9  ...
10 std::vector<result_type> r_apprx = M * c;                // FMM/Treecode
11 std::vector<result_type> r_exact = fmmtl::direct(M * c); // O(N^2)
```

Listing 4. A use case of an kernel matrix abstract data type

Providing a kernel matrix data type allows the abstraction level of our code to remain high while retaining generality and efficiency. Integration into existing efficient linear algebra libraries such as MKL, Eigen, and/or ViennaCL for use of solvers and preconditioners is an attractive option. Additionally, higher level linear algebra and computer science concepts such as submatrix blocking, lazy evaluation, and template expressions become a possibility.

4.1 Preliminary Benchmark

While FMMTL remains relatively new and the primary focus has been on design rather than performance, we provide preliminary results in this section to show that the abstractions and design decisions do not significantly impact raw performance of the algorithm.

In Figure 2, the performance of FMMTL for the benchmark case of the Laplace kernel (potential+force) with expansion order $p = 8$ and varying number of sources. The hardware used was an Intel Xeon W3670 3.2GHz CPU and an Nvidia GTX580 GPU. This closely follows the benchmark presented by Yokota et al. in [14].

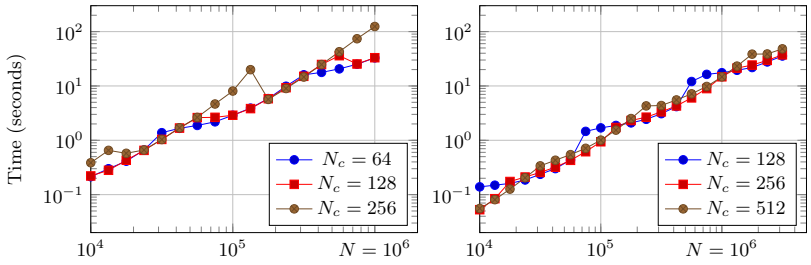

Fig. 2. Timings for the Laplace kernel (potential+force) showing the performance of an FMM with expansion order $p = 8$ and N particles uniformly distributed in a cube. These show (left) CPU only and (right) with GPU acceleration in S2T.

These results show that the performance of FMMTL for this benchmark is on par with that of a more well-tuned [14]. To alleviate the dependence of the performance on the value of N_c, the maximum number of bodies per leaf box, similar auto-tuning procedures may be performed as suggested in [14,1,13], which is made easier and more general due to the encapsulation of operators in FMMTL. In the current state of the library, much more static information about the kernel and expansion may be taken advantage of in the tree traversal and operator evaluation which will be discussed in a forthcoming optimization, performance analysis, and application study publication.

5 Conclusion

Reviewing the literature and codes produced for hierarchical matrix algorithms reveals a large number of difficult to use and modify research codes. In the FMMTL library, we attempt to separate out the needs of a kernel from the needs of an expansion and isolate the tree construction and tree

traversal. By doing so, continuing research into optimal kernel expansions, tree data structures, tree traversals, cluster interaction, and parallel computing strategies can continue independently of one another. At the very least, by growing a repository of kernels and expansions in a uniform format allows research to conduct fair comparisons – a requirement that is needed in the short-term to determine where and why to allocate research resources.

For the time being, careful design has been critical to the development of FMMTL to ensure that dependencies between components is low. Despite this, the serial performance is on par with hand-tuned lower-level research codes and higher performance already possible with OpenMP and GPU acceleration.

References

1. Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Pipelining the Fast Multipole Method over a Runtime System. Research Report RR-7981, INRIA, May 2012.
2. Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational n-body code that runs entirely on the GPU processor. *Journal of Computational Physics*, 231(7):2825–2839, April 2012.
3. Henry A. Boateng and Robert Krasny. Comparison of treecodes for computing electrostatic potentials in charged particle systems with disjoint targets and sources. *Journal of Computational Chemistry*, 34(25):2159–2167, 2013.
4. Cris Cecka and Eric Darve. Fourier-based fast multipole method for the helmholtz equation. *SIAM Journal on Scientific Computing*, 35(1):A79–A103, 2013.
5. H. Cheng, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm in three dimensions. *Journal of Computational Physics*, 155(2):468 – 498, 1999.
6. Will Fong and Eric Darve. The black-box fast multipole method. *Journal of Computational Physics*, 228(23):8712–8725, 2009.
7. J. Kurzak and B. M. Pettitt. Fast multipole methods for particle dynamics. *Molecular Simulation*, 32(10-11):775–790, 2006.
8. Hatem Ltaief and Rio Yokota. Data-driven execution of fast multipole methods. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2013.
9. John K. Salmon and Michael S. Warren. Skeletons from the treecode closet. *Journal of Computational Physics*, 111(1):136 – 155, 1994.
10. Toru Takahashi, Cris Cecka, William Fong, and Eric Darve. Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units. *International Journal for Numerical Methods in Engineering*.
11. Lexing Ying. A kernel-independent fast multpole algorithm for radial basis functions. *Journal of Computational Physics*, 213:451–457, 2006.
12. Lexing Ying, George Biros, and Denis Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics*, 196(2):591 – 626, 2004.
13. Rio Yokota. An fmm based on dual tree traversal for many-core architectures. *Journal of Algorithms & Computational Technology*, 7:301–324, 2013.
14. Rio Yokota and L. A. Barba. Hierarchical n-body simulations with autotuning for heterogeneous systems. *Computing in Science Engineering*, 14(3):30–39, 2012.