Published in final edited form as:

Int J High Perform Comput Appl. 2008 January 1; 22(2): 219–230. doi:10.1177/1094342008090915.

AUTOMATIC GENERATION OF FFT FOR TRANSLATIONS OF MULTIPOLE EXPANSIONS IN SPHERICAL HARMONICS

Jakub Kurzak¹, Dragan Mirkovic², B. Montgomery Pettitt^{2,3}, and S. Lennart Johnsson²

¹ DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, UNIVERSITY OF TENNESSEE, KNOXVILLE, TENNESSEE 37996 (KURZAK@EECS.UTK.EDU)

² DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF HOUSTON, HOUSTON, TEXAS 77204

³ DEPARTMENT OF CHEMISTRY, UNIVERSITY OF HOUSTON, HOUSTON, TEXAS 77204

Abstract

The fast multipole method (FMM) is an efficient algorithm for calculating electrostatic interactions in molecular simulations and a promising alternative to Ewald summation methods. Translation of multipole expansion in spherical harmonics is the most important operation of the fast multipole method and the fast Fourier transform (FFT) acceleration of this operation is among the fastest methods of improving its performance. The technique relies on highly optimized implementation of fast Fourier transform routines for the desired expansion sizes, which need to incorporate the knowledge of symmetries and zero elements in the input arrays. Here a method is presented for automatic generation of such, highly optimized, routines.

Keywords

fast multipole method; particle dynamics; spherical harmonics; fast Fourier transform; automatic code generation

1 Introduction

The fast multipole method (FMM) in two dimensions was introduced by Greengard (1987) and Greengard and Rokhlin (1987) and its adaptive formulation by Carrier, Greengard, and Rokhlin (1988). Later on it was extended to three dimensions in the work by Greengard and Rokhlin (1988), where notably the fast Fourier transform acceleration was proposed for the first time. Being the most computationally expensive part of the FMM, the translation of multipole expansions in spherical harmonics has been the subject of the most extensive optimization efforts. Greengard and Rokhlin (1988) proposed the fast Fourier transform acceleration technique, also pointing out problems with numerical stability of their method. Elliot and Board (1996) proposed a solution to these problems by using the fast Fourier transforms with block decomposition of the input arrays. At the same time White and Head-Gordon (1996) proposed an algorithm utilizing rotations of spherical harmonics. The most recent acceleration mechanism using the plane wave representation was introduced in two dimensions by Hrycak and Rokhlin (1998), and in three dimensions by Greengard and Rokhlin (1997).

Translation implemented as a convolution, in the original approach, has quadratic complexity, the method proposed by White and Head-Gordon yields cubic complexity and the methods based on the Fourier transforms and the method based on the plane wave representation are of square complexity.

Much effort has been invested in improving numerical stability of the original FFT acceleration scheme. However, numerical properties of the original solution turn out to be satisfactory for classic molecular dynamics simulations. At the same time, the performance advantages of the Fourier transform make the method attractive for implementation on high performance computers.

This work focuses on the fast Fourier transform acceleration of translation. There exist a number of techniques that accelerate the fast multipole algorithm in other ways than by speeding up the translation operation, which will not be discussed here. An extensive overview of these techniques is provided by Kurzak and Pettitt (2006).

2 The Fast Multipole Method

2.1 Mathematical Background

In the fast multipole method the electrostatic field is represented by *multipole expansion* and so-called *local* or *Taylor expansion*, both expressed using spherical harmonics. The simplified form of these expansions in solid harmonics introduced by Wang and LeSar (1996) will be used here. The notation by Elliott (1995) will be followed.

The multipole expansion for a set of N charges $q_1, q_2, \dots, q_N$ with spherical coordinates $\vec{r}_1 = (\rho_1, \alpha_1, \beta_1), \vec{r}_2 = (\rho_2, \alpha_2, \beta_2), \dots, \vec{r}_N = (\rho_N, \alpha_N, \beta_N)$ located *inside* a sphere S centered at the origin is defined by the formula

$$M_n^m = \sum_{i=1}^N q_i F_n^{*m}(-\vec{r}_i), \quad (1)$$

where $F_n^m(\vec{r})$ defines regular solid harmonics

$$F_n^m(\vec{r}) = \frac{(-1)^{n-m} \rho^n}{(n+m)!} e^{im\beta} P_n^m(\cos\alpha), \quad (2)$$

and $P_n^m(x)$ are *associated Legendre polynomials*, which can be defined in terms of the *ordinary Legendre polynomials* $P_n(x)$ by the Rodrigues formula

$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x). \quad (3)$$

For any point $\vec{r} = (\rho, \alpha, \beta)$ located *outside* the sphere S containing all the charges the potential due to these charges is given by the formula

$$\Phi(\vec{r}) = \sum_{n=0}^{\infty} \sum_{m=-n}^n M_n^m G_n^m(\vec{r}), \quad (4)$$

where $G_n^m(\vec{r})$ defines *irregular* solid harmonics

$$G_n^m(r) = \frac{(-1)^{n-m}(n-m)!}{\rho^{n+1}} e^{im\beta} P_n^m(\cos\alpha). \quad (5)$$

The local expansion for a set of N charges $q_1, q_2, \dots, q_N$ with spherical coordinates $\vec{r}_1 = (\rho_1, \alpha_1, \beta_1), \vec{r}_2 = (\rho_2, \alpha_2, \beta_2), \dots, \vec{r}_N = (\rho_N, \alpha_N, \beta_N)$ located *outside* a sphere S centered at the origin is defined by the formula

$$L_n^m = \sum_{i=1}^N q_i G_n^m(-\vec{r}_i). \quad (6)$$

For any point $\vec{r} = (\rho, \alpha, \beta)$ located *inside* the sphere S the potential due to these charges is given by the formula

$$\Phi(\vec{r}) = \sum_{n=0}^{\infty} \sum_{m=-n}^n L_n^m F_n^{*m}(\vec{r}). \quad (7)$$

The most important operation in the FMM is the multipole to local, or M2L, translation by a vector $\vec{r}_{M2L}$, which converts a multipole expansion to a local expansion with a new center. In Elliott's notation the expression for the M2L translation takes a very simple form

$$L_{n'}^{m'} = \sum_{n=0}^{\infty} \sum_{m=-n}^n M_n^m G_{n'+n}^{m'+m}(\vec{r}_{M2L}). \quad (8)$$

and, as can be seen, is a linear convolution of quadratic complexity.

For computer implementations of the calculations, the infinity limit of the summations is replaced with a finite parameter p , which is the order of the expansion, and determines the accuracy of force computation.

In classic molecular dynamics simulations values of p between 6 and 16 are of practical use. Values smaller than 6 cause unacceptable energy drift for any kind of simulation, whereas values larger than 16 exceed the accuracy of commonly used force integrators.

2.2 FMM Overview

The FMM utilizes hierarchical space domain decomposition, in which the simulation box is subdivided into eight smaller boxes, each of which is in turn further subdivided into eight yet

smaller boxes. The process continues recursively until the desired level of refinement is reached. Then the fast multipole algorithm proceeds in the following steps:

1. Multipole calculations

- a. *Multipole expansion calculation* – the multipole expansions are calculated for the cells of the finest granularity;
- b. *Upward pass* – the multipole expansions are propagated to the higher levels of the decomposition;
- c. *Well separated cell interactions* – for all cells, on all levels, the multipole expansions of the distant (*well separated*) cells are accumulated into the local expansions by the M2L translations;
- d. *Downward pass* – the local expansions from the higher levels are propagated down to the lower levels of the decomposition;
- e. *Local expansion evaluation* – the forces and the potentials are calculated from the local expansions.

2. **Direct particle to particle calculation** – the forces and the potentials are calculated from the particle positions and the charges.

Figure 1 depicts the multipole operations of steps 1a–1d in the FMM algorithm.

More refinement levels result in more time spent in the multipole part and less time spent in the direct part. Depending on the size of the system, the number of refinement levels is chosen in such a way that approximately the same amount of time is spent in the multipole part and the direct part. Since the direct part simply applies Coulomb's Law

$$\vec{F} = q_1 q_2 \frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3} \quad (9)$$

to calculate electrostatic interactions, little can be done to improve its performance. One possible improvement is the application of the Newton's Third Law. Much more can be done in the multipole part. Here the huge number of M2L translations in step 1c completely dominates the time of the computation and is the main target of performance-improving techniques.

2.3 FFT Acceleration

The FFT acceleration technique used here is basically the one proposed by Elliott and Board (1996), except the block decomposition has been dropped, and scaling of multipole coefficients has been introduced, which is schematically represented on Figure 2. Here the approach will be briefly outlined. For detailed derivation of the method the reader is referred to the original article.

The basic idea behind speeding up the M2L translation is to convert the multipole expansion M_n^m and the transfer function $G_n^m(\vec{r})$ to Fourier space and replace the convolution with the element-wise product. Commonly the conversion from the FFT circular convolution to the M2L linear convolution is achieved with zero padding. Since in the FMM the coefficients run from $n = 0$ to $p - 1$ and $m = -n$ to n , normally the input arrays in Fourier space would have to be of size $4p \times 2p$. However, because of the triangular nature of the arrays, the wrap-around effect of the FFT circular convolution in the m direction does not affect the values within the

allowed range, and the zero padding in the m direction can be dropped altogether, which results in the size of the input arrays of $2p \times 2p$. Moreover, the following symmetries are present in the multipole and local M_n^m and L_n^m expansions and the $G_n^m(\vec{r})$ and $F_n^m(\vec{r})$ functions:

$$F_n^{-m}(\vec{r}) = (-1)^m F_n^{*m}(\vec{r}), \quad (10)$$

$$G_n^{-m}(\vec{r}) = (-1)^m G_n^{*m}(\vec{r}), \quad (11)$$

$$M_n^{-m} = (-1)^m M_n^{*m}, \quad (12)$$

$$L_n^{-m} = (-1)^m L_n^{*m}. \quad (13)$$

As a result of these symmetries, the coefficients for the negative values of m are never stored in the FMM and always derived from the coefficients for positive m . By the same token, the Fourier space representation is an array of size $p \times 2p$.

The input matrix $\tilde{M}_n^m$ is constructed simply by copying the values from its triangular representation for $n \in [0, p-1]$ and $m \in [0, n]$ to the rectangular matrix $p \times 2p$.

$$\tilde{M}_n^m = M_n^m. \quad (14)$$

The $\tilde{G}_n^m$ input matrix is constructed by copying the values from its triangular representation for $n \in [0, p-1]$ and $m \in [0, n]$ to the rectangular matrix $p \times 2p$ according to the formula

$$\tilde{G}_{p-n-1}^m = (-1)^m G_n^{*m}. \quad (15)$$

The two-dimensional FFT of the input arrays is performed by applying one-dimensional FFTs to the rows of the input arrays and one-dimensional FFTs to the columns of the arrays. The row FFT is of size $2p$, but only operates on indices from 0 to $p-1$, relying on the knowledge of the symmetry to derive the values for indices from -1 to $-(p-1)$, and only storing indices of the output vector from 0 to $p-1$. Also, since only rows from 0 to $n-1$ contain the data and rows from n to $2n-1$ contain zero padding, only n row FFTs on the first half (lower indices) of the arrays need to be performed. The column FFTs are one-dimensional FFTs with stride p including the knowledge that the second half (higher indices) of the input vector is filled with zeros.

The result of the element-wise multiplication of the $p \times 2p$ $\tilde{M}_n^m$ matrix with the $p \times 2p$ $\tilde{G}_n^m$ matrix is a $p \times 2p$ $\tilde{L}_n^m$ matrix in Fourier representation, which has to be transformed back to the coefficient space. Analogously to the forward two-dimensional FFT, here two-dimensional inverse FFT is implemented by first applying inverse column FFTs, and then inverse row FFTs,

followed by scaling of the output matrix by $1/4p^2$. The inverse column FFTs read in and store the full input vectors. The inverse row FFTs read in indices of the input vector from 0 to $p-1$, and reconstruct the indices from -1 to $-(p-1)$ using the relation

$$\tilde{L}_{FFT\ n}^{-m} = \tilde{L}_{FFT\ n}^{*-m}. \quad (16)$$

When this operation completes, the L_n^m matrix in the coefficient space is reconstructed from the resulting matrix by combining the elements from the rows 0 to $n-1$ and the elements from the rows n to $2n-1$ in the following way:

$$L_n^m = (-1)^m \left(\tilde{L}_n^m + \tilde{L}_{2p-n-1}^m \right)^*. \quad (17)$$

The numerical stability problems with the FFT acceleration are caused by the ρ^n term in the $F_n^m(\vec{r})$ function and the $1/\rho^{n+1}$ term in the $G_n^m(\vec{r})$ function, along with the factorial terms present in both of these functions. A simple solution to these problems exists for expansion sizes up to 16. Before the translation, the multipole expansion coefficients of a cubic box of dimension d are scaled to the coefficients of a box of unit size by applying the formula

$$\tilde{M}_{n'}^{m'} = \frac{M_n^m}{d^{n+1}}, \quad (18)$$

and the transfer function $G_n^m(\vec{r})$ is constructed using a vector $\vec{r}' = \vec{r}/d$. After translations the local expansion coefficients are scaled back to coefficients of the original box by applying the formula

$$\tilde{L}_{n'}^{m'} = \frac{L_n^m}{d^n}, \quad (19)$$

For expansion sizes greater than 16 the FFT with block decomposition proposed by Elliott and Board (1996) can be used, or the new algorithm by Greengard and Rokhlin (1997). Both solutions allow for the use of big expansion sizes, but both have drawbacks. The block decomposition scheme, being a coarse convolution on the blocks of the input array, approximately doubles the time of computing the product of input matrices in the Fourier space. The latter algorithm computes the product in approximately the same time as the FFT algorithm with scaling, but introduces a very time consuming operation of conversion from spherical harmonics to plane waves. At the same time higher accuracy can be achieved by increasing the separation criterion, which can be done gradually if the spherical interaction regions proposed by Elliott (1995) are used.

3 FFT Code Generation

3.1 Mathematical Background

The fast Fourier transform (FFT) is a method of evaluating the discrete Fourier transform (DFT), which in principle is a matrix vector product, which requires an $O(n^2)$ operation. FFT reduces the number of required operations to an $O(n \log n)$. In this section a brief overview is

given of the FFT algorithms utilized by the UHFFT code generator. For a complete discussion the reader is referred to the literature (Heideman 1988; Tolimieri, An, and Lu 1989, 1997; Duhamel and Vetterli 1990; Van Loan 1992).

If $\mathbf{C}^n$ denotes the space of complex n -dimensional vectors with components indexed from zero to $n - 1$, then in matrix vector notation the DFT of $\vec{x} \in \mathbf{C}^n$ can be defined by

$$\vec{y} = W_n \vec{x},$$

where $W_n = (w_{lk})_{l,k=0}^{n-1}$ is the DFT matrix with elements $w_{lk} = \omega_n^{lk}$, and $\omega_n = e^{-2\pi i/n}$ are complex roots of unity, called *twiddle factors* in the context of DFT. The fast evaluation is obtained through a factorization of W_n into a product of $O(\log n)$ sparse matrices:

$$W_n(\vec{x}) = (A_1 A_2 \dots A_r) \vec{x},$$

where matrices A_i are sparse and calculation of the product $A_i \vec{x}$ involves an $O(n)$ operation. For a given n the factorization is not unique, and possible variations may have substantially different properties. the UHFFT code generator generates an abstraction of the FFT algorithm by using a combination of the mixed-radix algorithm (Cooley and Tukey 1965), the split-radix algorithm (Duhamel and Hollmann 1984), the prime factor algorithm (Good 1958; Thomas 1963) and the Rader's (1968) algorithm.

By far the most common FFT algorithm is the Cooley–Tukey mixed-radix algorithm (Cooley and Tukey 1965), which breaks down a DFT of any composite size $n = rq$ into two smaller DFTs of sizes r and q , along with an $O(n)$ multiplication by *twiddle factors*:

$$W_n = (W_r \otimes I_q) D_{r,q} (I_r \otimes W_q) \Pi_{n,r},$$

where $D_{r,q}$ is the diagonal *twiddle factor matrix*, and $\Pi_{n,r}$ is a mod- r sort permutation matrix. Unfortunately, a substantial fraction of work in this algorithm is associated with the construction and application of the $D_{r,q}$ matrix.

The split-radix algorithm (Duhamel and Hollmann 1984) can be used when n is divisible by 4, and is based on a synthesis of one DFT of length $n/2$ together with two DFTs of length $n/4$:

$$W_n = B(W_{n/2} \oplus W_{n/4} \oplus W_{n/4}) \Pi_{n,n/2,2},$$

where B is the split-radix butterfly matrix and $\Pi_{n,n/2,2}$ is the split-radix permutation matrix.

The prime factor algorithm (Good 1958; Thomas 1963), also called the Good–Thomas algorithm, is based on the observation that, when q and r are relatively prime, i.e., $\gcd(r, q) = 1$, the scaling by *twiddle factors* can be eliminated. This algorithm is based upon a splitting of the form:

$$W_n = (P_1 (W_r \otimes I_q) (I_r \otimes W_q) P_2 = P_1 (W_r \otimes W_q) P_2),$$

where P_1 and P_2 are permutations.

For prime sizes, Rader's (1968) algorithm computes the DFT by expressing it as a cyclic convolution:

$$W_n = Q_1 \begin{bmatrix} 1 & e^T \\ e & C_{n-1} \end{bmatrix} Q_2,$$

where e is a vector of all ones, Q_1 and Q_2 are permutations and C_{n-1} is a circulant matrix. Since $n-1$ is a composite number, the action of C_{n-1} can be obtained efficiently by using the FFT. The main idea of the FFT algorithm is that, if n is not a prime number, splitting can be applied recursively, and if n is a prime number, Rader's algorithm can be used to reduce the problem to a non-prime size problem, which can then be subjected to further recursive splitting.

3.2 UHFFT Code Generator

The UHFFT (Mirkovic, Mahasoom, and Johnsson 2000; Mirkovic and Johnsson 2004) code generator is a special-purpose compiler in many aspects similar to the FFTW (Frigo 1999; Frigo and Johnsson 2005) code generator. Unlike FFTW, it is written in C, which makes it fast and efficient and ultimately portable. Figure 3 shows the structure of the UHFFT code generator. A brief description of its main blocks follows.

In the factorization and algorithm selection steps the size of the input array is split into smaller factors, and the algorithms described in the previous section are recursively applied, in order to produce an algorithm with the smallest number of arithmetic operations. By default, the following rules are used in the factorization and algorithm selection steps:

- If possible, the prime factor algorithm (PFA) is used for co-prime factors;
- Otherwise, if possible, the split-radix algorithm is used;
- Otherwise the mixed-radix algorithm is used;
- The Rader's algorithm is used for prime factors.

For the FMM specific array sizes, these rules result in the factorizations and algorithm selection presented in Table 1.

The arithmetic optimization step performs a number of classic code transformations including:

- Constant folding;
- Simplifications of expressions involving a binary operator and its algebraic identity element (e.g. multiplication by 0);
- Simplifications involving a unary operator or combinations of unary and binary operators (e.g. consecutive sign changes);
- Common subexpression elimination.

At the end, the direct acyclic graph (DAG) of the computation is used to resolve dependencies and schedule instructions. Finally, the abstract code is unparsed to produce the desired C code of the FFT routine.

3.3 FMM Specific Transforms

In the arithmetic optimization phase the code generator is provided with the information about the symmetries and zero elements in the input arrays, and the discarded elements in the output

arrays, and the unnecessary operations are eliminated through the optimizations applied to the graph of the computation.

In particular, the following features are specified about the transform operations. The *forward row* FFT is a one-dimensional complex FFT of the size $N = 2p$, where the input vector has the symmetry

$$\begin{array}{cccccc} 0, & 1, & 2, & \dots, & \frac{N}{2} - 1, & \frac{N}{2}, \\ X_0, & X_1, & X_2, & \dots, & X_{\frac{N}{2}-1}, & 0, \\ & & \frac{N}{2}+1, & \dots, & N-3, & N-2, & N-1 \\ & & \pm X_{\frac{N}{2}-1}^*, & \dots, & -X_3^*, & X_2^*, & -X_1^* \end{array} \quad (20)$$

and only the elements from 0 to $p-1$ are stored in the memory for both the input and the output vectors. The *inverse row* FFT is a one-dimensional complex FFT of the size $N = 2p$, where the input vector has the symmetry

$$\begin{array}{cccccc} 0, & 1, & 2, & \dots, & \frac{N}{2} - 1, & \frac{N}{2}, \\ X_0, & X_1, & X_2, & \dots, & X_{\frac{N}{2}-1}, & X_0^*, \\ & & \frac{N}{2}+1, & \frac{N}{2}+2, & \dots, & N-1 \\ & & X_1^*, & X_2^*, & \dots, & X_{\frac{N}{2}-1}^* \end{array} \quad (21)$$

and, again, only the elements from 0 to $p-1$ are stored in the memory for both the input and the output vectors. The *forward column* FFT is a one-dimensional complex FFT of the size $2p$ with the stride p , where the elements with indices from p to $2p-1$ are known to be zero, and the full output array is computed. The *inverse column* FFT is a one-dimensional complex FFT of the size $2p$ with the stride p , where there are no zero elements in the input array, and the full output array is computed.

4 Results

Tables 2 and 3 show numbers of floating point operations (additions, multiplications, sign changes and assigns) for individual codelets associated with multipole expansions of sizes p from 6 to 16. Figure 4 presents performance, in terms of execution time and Gflops, of the individual codelets for those expansion sizes. Figure 5 shows performance, in terms of execution time and Gflops, of the entire two-dimensional Fourier transform in comparison with other operations common in the FMM algorithm.

The FMM operations presented in Figure 5 are the following:

- *M2L conv* – multipole to local translation implemented as a convolution of an $O(p^4)$ complexity;
- *Rotation* – rotation of a multipole expansion by $\frac{\pi}{2}$ about the y axis of an $O(p^3)$ complexity, using a Wigner matrix;
- *FFT2L* – inverse Fourier transform of a local expansion;
- *M2FFT* – forward Fourier transform of a multipole expansion;

- *M2L FFT* – multipole to local translation implemented as an element-wise product in Fourier space of an $O(p^2)$ complexity;
- *Scaling* – scaling of multipole expansion coefficients of an $O(p^2)$ complexity.

It is important to notice, again, the better performance of the forward routines, which stems from the fact that more operations can be eliminated from the routines implementing these transforms. Figure 5 compares the performance of the entire two-dimensional Fourier transform when using the optimized codelets and a generic FFT library. The specific library used for this comparison is the FFTW (Frigo 1999; Frigo and Johnson 2005).

In the FFT community it is common to use the $n \log n$ scale for the transform sizes. Here, however, transform sizes are related to multipole expansion sizes for which $n \log n$ scale is never used. For that reason it was decided to use linear scale for Fourier transform sizes.

The benchmarking was performed on an Itanium2 1.4 GHz processor and an Opteron 2.0 GHz processor, which are both high performance processors, yet with different performance characteristics. It is not a goal here to make comparisons between those processors. The benchmarked routines have to be considered micro-kernels and their performance is not representative of the overall processor performance. Also, it should be pointed out that the results represent performance when both the code and the data resides in cache and are not representative of performance when run as a part of the entire FMM algorithm.

One of the most important observations is that, when using the techniques presented in this paper, there is no severe penalty for using uncommon sizes of FFTs, including sizes containing large prime numbers in their factorizations (Figure 4). In particular sizes 22 and 26, which correspond to multipole expansion sizes 11 and 13, perform very well, and interestingly, achieve the best Gflops values, which is because they have very close numbers of floating point additions and multiplications (Tables 2 and 3). Those instructions can be performed in parallel on most modern processors including those used for benchmarking here.

It has been proposed to perform redundant FFTs in a parallel implementation of the FMM in order to decrease network traffic (Kurzak and Pettitt 2005a, 2005b). In such cases the routine performed redundantly is the forward FFT. It turns out that the biggest improvements can be achieved when optimizing forward routines, and as a result, they perform faster than the inverse routines.

Although the routines perform quite well for all the sizes, some sizes are noticeably better than others. Size $p = 12$ performs particularly well, which is important since it gives a very good trade-off between accuracy and performance in molecular dynamics simulations.

As expected, being an $O(p^2 \log^2 p)$ operation, the FFT executes faster than the $O(p^3)$ operation and slower than the $O(p^2)$ operations. Not surprisingly, the most efficient operation, in terms of Gflops number, is the M2L translation in Fourier space, which is an element-wise product of two complex vectors, implemented by a single loop with exactly the same number of additions and multiplications. It may be noticed that on the Itanium2 processor the Gflops number of the 2-D forward FFT matches the Gflops number of the M2L translation for expansion sizes of 11 and 13, because of their good ratio of additions and multiplications. It may seem counterintuitive that higher complexity operations achieve lower Gflops numbers than lower complexity operations. It would not be the case if large data sets were used and memory access times would come into play.

Figure 6 shows the impact of the solution proposed in this article. For expansion sizes of 6, 7 and 8 the forward transform is roughly three times faster than a generic FFT routine and the inverse transform is roughly two times faster, which is the result of the reduced number of

floating point instructions and memory accesses when symmetries and zero elements are addressed.

The big performance difference for sizes of 9 and bigger is because the standard setting the FFTW library only contains codelets for transform sizes up to 16 ($p = 8$) and has to combine different codelets for bigger sizes, whereas in the proposed solution one-dimensional routines for each size are implemented as single codelets (single blocks of straight-line code) and eliminate the penalty of function calls.

5 Conclusions

In this publication a methodology was shown for crafting fast Fourier transform routines for a specific problem –translation of multipole expansions in spherical harmonics. A special purpose compiler was utilized to produce FFT routines, which take advantage of symmetries and zero elements in the input arrays and pruned elements of the output arrays, decreasing the number of floating point instructions and memory accesses. The resulting routines are at least twice as fast as a generic FFT library routines and an order of magnitude faster in some cases.

6 Software

The FFT routines presented in the paper are a key component of a parallel FMM package being developed by the Institute for Molecular Design at the University of Houston. Although the FMM code is currently not being publicly distributed, the FFT routines are available from the authors upon request.

Acknowledgments

The authors thank NASA and NIH for funding. The authors also wish to thank the referees for their helpful suggestions.

References

- Carrier J, Greengard L, Rokhlin V. A fast adaptive multipole algorithm for particle simulations. *J Sci Stat Comput* 1988;9(4):669–686.
- Cooley JW, Tukey JW. An algorithm for the machine calculation of complex Fourier series. *Math Comput* 1965;19:47–64.
- Duhamel P, Hollmann H. Split radix FFT algorithm. *Electronics Letters* 1984;20:14–16.
- Duhamel P, Vetterli M. Fast Fourier transforms: A tutorial review and a state of the art. *Signal Process* 1990;19(4):259–299.
- Elliott, WD. Ph D thesis. Duke University, Department of Electrical Engineering; Durham, NC: 1995. Multipole algorithms for molecular dynamics simulation on high performance computers.
- Elliott WD, Board JA Jr. Fast Fourier transform accelerated fast multipole algorithm. *SIAM J Sci Comput* 1996;17(2):398–415.
- Frigo, M. A fast Fourier transform compiler. *Proceedings of 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM; 1999. p. 169–180.
- Frigo M, Johnson SG. The design and implementation of FFTW3. *Proceedings of the IEEE* 2005;93(2): 216–231. Special issue on “Program Generation, Optimization, and Platform Adaptation.”
- Good IJ. The interaction algorithm and practical Fourier analysis. *J Roy Statist Soc B* 1958;20(2):361–375.
- Greengard, L. Ph D thesis. Yale University, Department of Computer Science; New Haven, CT: 1987. The rapid evaluation of potential fields in particle systems.
- Greengard L, Rokhlin V. A fast algorithm for particle simulations. *J Comput Phys* 1987;73(2):325–348.
- Greengard, L.; Rokhlin, V. On the efficient implementation of the fast multipole algorithm. Yale University, Department of Computer Science; New Haven, CT: 1988. Technical Report RR-602

- Greengard L, Rokhlin V. A new version of the fast multipole method for the Laplace equation in three dimensions. *Acta Numer* 1997;6:229–269.
- Heideman, MT. Multiplicative complexity, convolution, and the DFT. Springer-Verlag; New York, NY: 1988.
- Hrycak T, Rokhlin V. An improved fast multipole algorithm for potential fields. *SIAM J Sci Comput* 1998;19(6):1804–1826.
- Kurzak J, Pettitt BM. Communications overlapping in fast multipole particle dynamics methods. *J Comput Phys* 2005a;203(2):731–743.
- Kurzak J, Pettitt BM. Massively parallel implementation of a fast multipole method for distributed memory machines. *J Parallel Distrib Comput* 2005b;65(7):870–881.
- Kurzak J, Pettitt BM. Fast multipole methods for particle dynamics. *Mol Simul* 2006;32(1011):775–790. [PubMed: 19194526]
- Mirkovic D, Johnsson SL. Automatic performance tuning for fast Fourier transforms. *Int J High Perf Comput Applic* 2004;18(1):47–64.
- Mirkovic, D.; Mahasoom, R.; Johnsson, SL. An adaptive software library for fast Fourier transforms. *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*; Dallas, TX. 2000. p. 215-224.
- Rader CM. Discrete Fourier transforms when the number of data samples is prime. *Proc IEEE* 1968;56(6):1107–1108.
- Thomas, LH. *Applications of Digital Computers*. Ginn and Co; Boston, MA: 1963. Using a computer to solve problems in physics.
- Tolimieri, R.; An, M.; Lu, C. *Algorithms for discrete Fourier transform and convolution*. Springer-Verlag; New York, NY: 1989.
- Tolimieri, R.; An, M.; Lu, C. *Mathematics of multidimensional Fourier transform algorithms*. Springer-Verlag; New York, NY: 1997.
- Van Loan, C. *Computational frameworks for the fast Fourier transform*. Society for Industrial and Applied Mathematics; Philadelphia, PA: 1992.
- Wang HY, LeSar R. An efficient fast-multipole algorithm based on an expansion in the solid harmonics. *J Chem Phys* 1996;104(11):4173–4179.
- White CA, Head-Gordon M. Rotating around the quartic angular momentum barrier in fast multipole method calculations. *J Chem Phys* 1996;105(12):5061–5067.

Biographies

Jakub Kurzak received his M.Sc. degree in electrical and computer engineering from Wroclaw University of Technology, Poland, and his Ph.D. degree in computer science from the University of Houston. He is research associate in the Innovative Computing Laboratory in the Department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville. His research interests include parallel algorithms, specifically in the area of numerical linear algebra, and also parallel programming models and performance optimization for parallel architectures spanning distributed and shared memory systems, as well as next generation multi-core and many-core processors.

Dragan Mirkovic received his B.Sc. and M.Sc. degrees in nuclear engineering from University of Zagreb, Croatia, and his Ph.D. degree in applied mathematics from State University of New York at Stony Brook. He is visiting assistant professor in the Department of Computer Science at the University of Houston. His research interests include parallel adaptive Fourier transforms, fast spherical and Fourier transform, 3-D image reconstruction, ultra-scale parallel computing, grid applications, and modeling and simulation of hemodynamics problems.

Montgomery Pettitt received his B.Sc. degrees in chemistry and mathematics and his Ph.D. degree in physical chemistry from the University of Houston. He is Hugh Roy and Lillie Cranz Cullen Distinguished Professor of Chemistry, professor of Computer Science, Physics, Biology and Biochemistry at the University of Houston, director of the Institute for Molecular Design

and chair of the Keck Center for Interdisciplinary Biology. His interests include multidisciplinary research in computational science spanning the areas of chemical physics, physical chemistry, biochemistry and computer science.

Lennart Johnsson received his M.Sc. degree in engineering physics and his Ph.D. degree in control engineering from Chalmers Institute of Technology, Gothenburg, Sweden. He is Hugh Roy and Lillie Cranz Cullen Distinguished Professor of Computer Science, Mathematics and Electrical and Computer Engineering at the University of Houston, and Director of the Texas Learning and Computation Center. His research interests include computational and data grids, high-performance scientific computing, parallel algorithms.

Fig. 1.
Multipole operations in the FMM.

Fig. 2.
FFT acceleration of the M2L translation.

Fig. 3.
Operation of the UHFFT code generator.

Fig. 4. Execution time and Gflops number for optimized one-dimensional FMM FFT codelets.

Fig. 5. Execution time and Gflops number for optimized two-dimensional FMM FFT transforms compared with other common FMM operations. (A detailed explanation of each type of operation can be found at the beginning of Section 4).

Fig. 6.

Execution time for optimized two-dimensional FMM FFT transforms compared with a generic FFT library (M2FFT – custom forward Fourier transform of multipole expansion, FFT2L – custom inverse Fourier transform of local expansion, M2FFT_FFTW – forward Fourier transform of multipole expansion using FFTW, FFT2L_FFTW – inverse Fourier transform of local expansion using FFTW).

Table 1

FFT factorizations and algorithm selection for the FMM specific array sizes.

Expansion size p	Transform size N	Factorization
6	12	(2 m 2) p 3
7	14	(2 p 7)
8	16	sr 2
9	18	(3 m 3) p 2
10	20	(2 m 2) p 5
11	22	(2 p 11)
12	24	((2 m 2) m 2) p 3
13	26	(2 p 13)
14	28	(2 m 2) p 7
15	30	(2 p 3) p 5
16	32	sr 2

m – mixed-radix,

p – prime factor,

sr – split-radix.

Rader's algorithm is used for prime factors.

Table 2

Operation count for row FFT routines.

Type	Expansion size p	Transform size N	Number of operations			
			Adds	Muls	Sign Xchg	Assigns
Forward	6	12	64	14	13	90
	7	14	44	48	6	55
	8	16	108	26	4	130
	9	18	148	80	40	214
	10	20	126	44	18	166
	11	22	114	120	10	91
	12	24	204	42	35	258
	13	26	161	168	12	109
	14	28	220	120	16	209
	15	30	280	108	52	378
	16	32	320	86	10	362
						Total
Inverse	6	12	27	20	10	58
	7	14	127	72	6	105
	8	16	58	40	11	97
	9	18	175	80	45	241
	10	20	185	48	18	221
	11	22	291	200	10	173
	12	24	111	60	39	182
	13	26	397	288	12	207
	14	28	321	144	14	277
	15	30	339	112	57	433
	16	32	192	116	31	272
						Total

Table 3
Operation count for column FFT routines

Type	Expansion size p	Transform size N	Number of operations				
			Adds	Muls	Sign Xchg	Assigns	Total
Forward	6	12	72	16	16	112	216
	7	14	116	72	0	114	302
	8	16	112	24	4	148	288
	9	18	160	80	36	244	520
	10	20	160	40	14	224	438
	11	22	272	200	0	186	658
	12	24	204	44	38	284	570
	13	26	374	288	0	222	884
	14	28	258	108	3	290	659
	15	30	304	104	46	436	890
	16	32	308	84	10	380	782
Inverse	6	12	96	16	12	136	260
	7	14	148	72	0	140	360
	8	16	144	24	8	180	356
	9	18	196	80	36	280	592
	10	20	208	48	8	264	528
	11	22	324	200	0	228	752
	12	24	252	44	30	332	658
	13	26	436	288	0	272	996
	14	28	352	144	0	336	832
	15	30	372	112	42	496	1022
	16	32	372	84	22	444	922