Automatic Parallel Code Generation for NUFFT Data Translation

Yuanrui Zhang and Mahmut Kandemir

Department of Computer Science and Engineering
Technical Report
CSE-09-003

January 26, 2009
Automatic Parallel Code Generation for NUFFT Data Translation

Yuanrui Zhang
Penn State University
yuazhang@cse.psu.edu

Mahmut Kandemir
Penn State University
kandemir@cse.psu.edu

Abstract

NUFFT (Non-Uniform FFT) is a powerful numerical tool that works with nonuniformly spaced data and has numerous application areas ranging from embedded signal processing to scientific computing. Data translation is an important component of NUFFT and is currently optimized by hand based on expert programmers knowledge on target parallel architecture and application domain. This strategy clearly is not a scalable option, given the fact that future multi-core architectures will have very different processor configurations and on-chip memory characteristics as compared to each other, making it very difficult to optimize the data translation step for each architecture individually. Instead, in this paper, we take a different approach, and present a customized automation tool that specifically targets the NUFFT data translation step. A unique feature of this tool is that it takes the architectural description of the underlying multi-core system as input, relieving the programmer from implementing custom optimizations. The proposed tool generates a data locality-optimized parallel output code and the preliminary results we collected are very encouraging on four different shared-memory multi-core systems.

1. Introduction

Fast Fourier Transform (FFT) has been widely used as a powerful numerical analysis tool in many areas, such as embedded signal processing and scientific computing since Cooley and Tukey (Cooley1965). A requirement for using FFT algorithms is that the input data must be equally spaced. In many applications, such as those in radar (Subia2003), medical imaging (Knoopp2007), telecommunications (Zhao2008), geoscience and seismic (Djuric2019), however, the input data is nonuniform (i.e., not equally spaced), and hence the regular FFT does not apply. To overcome this difficulty, several Non-Uniform FFT (NUFFT) algorithms have been developed (Dutt1993; Liu1998; Beylkin1995).

A typical NUFFT algorithm is composed of three steps, a data translation (also called convolution, or gridding) that translates the sample points between the Cartesian and non-Cartesian grids, a conventional regular FFT that operates on the Cartesian points, and a translation scaling that performs kernel rolloff correction. Although the order of these computation steps is slightly different between the time and frequency domains, the data translation and regular FFT steps typically constitute the significant fraction of the overall execution time. In particular, the former is by far the most time consuming part that takes about 83% to 95% of the overall execution latency (Sorensen2008).

Although from a mathematical point of view, the performance of data translation can be improved by choosing a proper kernel function for convolution computation (Dutt1993; Beylkin1995; Liu1998-2), for a particular kernel function with a specified accuracy requirement, data translation can be further improved using a parallel computing platform, e.g., a platform with multi-core architectures. Consequently, efficient parallelization of data translation that can utilize the computing capability provided by parallel processors may be very useful in practice.

Traditional compiler-aided parallelization techniques (Paavola1991; Gupta1992; Chow1996) are limited in this case since intrinsic data parallelism in the NUFFT data translation application itself cannot be easily extracted using a compiler; a specific code implementation makes this task very difficult in practice. As a result, exploiting parallelization in data translation usually requires programmers to write parallel code based on their understanding of the problem and the underlying parallel architecture features.

Instead of this manual process, we are concentrating on developing an automation tool that generates an optimal parallel NUFFT data translation code for a given target parallel computing platform, with user-specified algorithmic parameters and architectural description. This automation tool consists of two steps, namely, data parallelization and code generation. The first step determines a customized data parallelization strategy according to the on-chip memory (e.g., cache or local software-managed memory) size and structure and the number of processors for a particular parallel architecture, rearranges input data accordingly in the off-chip memory space or in an input file if the total data size is too large, and outputs a "table" that captures both data partitioning and data scheduling for multiple processors. The second step, code generator, then produces a parallel C code for the required data translation based on this table, which is expected to perform very well on the given execution platform since its data parallelism is tailored for the target architecture, exploiting both concurrency and data locality. Our automation tool can generate parallel code for both shared and distributed memory models. We currently tested our parallelization strategy for the shared memory based chip-multiprocessors (CMPs), which allows us study the parallel algorithm efficiency when exercising different on-chip cache memory structures. The simulation results show that our data parallelization scheme brings significant performance improvements compared to an alternate approach that does not optimize for on-chip cache hierarchy.

The rest of the paper is organized as follows. Section 2 introduces the NUFFT data translation algorithm and its parallelization. Section 3 describes automatic code generation for NUFFT data translation in detail. Section 4 presents our experimental results. Section 5 presents related work. And finally, Section 6 gives conclusions and future work.

2. NUFFT Data Translation Parallelization

To make graphical illustration simpler, we analyze and discuss parallel data translation algorithms for NUFFT in the context of two-dimensional (2D) case in the frequency domain. With minor modifications, our analysis and discussion are also valid for the NUFFT data translation in the time domain, and can be extended easily to the three-dimensional (3D) case.

In the NUFFT algorithm of frequency domain, the conventional FFT is preceded by a convolution step and followed by a kernel
rolloff correction. An optional density compensation step may be performed before the convolution (Debroya2008). In the frequency domain, the convolution translates the non-equally spaced sample points (Sources) to the Cartesian grid points (Targets) designated for the FFT. If the number of source sample points is N, the number of target points will be αN, where α is the oversampling factor that determines the size of the FFT. α is often less than 2 for 2D and 3D data translations.

There are different convolution kernel function alternatives for data translation. For a particular kernel function C(t, s) (t and s represent for target and source coordinates, respectively), a window W is selected to meet certain accuracy requirement; thus, a source (or target) sample only influences (or is influenced by) the nearby targets (or sources) within this window. In other words, a source is only involved in the convolution computation with targets in window W, and vice versa. The pseudo code for data translation can be expressed as follows:

for each source i
  for each target j in the window W
    targets[j] = targets[j] + sources[i] * C(t, s);

where the targets[] and sources[] arrays hold target and source sample values, respectively. If the outer loop iterates over all sources as shown above, we refer to it as Source Centered, otherwise, is called Target Centered. Source centered code is preferred because sources are irregularly spaced whereas targets are regularly spaced, and it is much easier and more straightforward to find surrounding targets within W for a source than the other way around. W is usually very small, i.e., a source affects q = 5 nearby targets in one dimension (Dutt1993) (the relationship of W and q for Gaussian kernel is discussed in (Dutt1993)). Parallelizing the outer (source) loop is expected to result in higher performance gains. One needs to ensure however that no two processors will attempt to update the value of the same target point concurrently as this can produce undefined results. In this case, a simple parallelization strategy that distributes iterations of the outer loop across processors cannot be applied as the order of input source samples is random in the memory space with respect to their spatial locations. To process sources safely in parallel that have no shared neighboring targets, partitioning in the data space, called Geometric Tiling (Tiling) (Chen2005) is needed, as depicted in Figure 1. With tiling, data locality can be exploited as well since neighboring sources may share targets. Hence, processing sources tile by tile for a particular processor can significantly improve data translation performance compared to a random order.

![Figure 1](image1.png)

Figure 1. Two source distribution shapes and geometric tiling in the data space.

After the geometric tiling, the data translation code can be expressed as follows:

for each source i
  for each target j in the window W
    targets[j] = targets[j] + sources[i] * C(t, s);

This transformation is similar in spirit to the conventional loop tiling (Renganarayana2004), an optimization technique used by compiler to improve data locality, but it takes a data-centric view. The inner code in italic format (above) has the same structure as the original code given earlier, and can be encapsulated into codelets (or libraries) with parameters such as window W. Assuming that codelets for different kernel functions are available, parallelization of data translation targets primarily the outer loop which iterates over source tiles. Thus, the way of tiling and tile scheduling across multiple processors directly determines the parallel code shape.

Intuitively, a naive tiling and scheduling scheme without consideration of underlying on-chip memory design of the target platform could not bring as performance improvement as a scheme that takes into account architecture features such as on-chip memory capacities and hierarchy. This is because emerging multi-cores are very different from each other as far as their on-chip memory (cache) structures are considered. Motivated by this observation, we have developed an architecture aware tiling and scheduling scheme, based on which our parallel data translation code is generated. In the rest of this paper, when no confusion occurs, we use "processor", "core" and "CPU" interchangeably.

3. Automatic Parallel Code Generation

Our automation tool for the NUFFT data translation consists of two main steps, data parallelization and code generation, as shown in Figure 2. Given the algorithmic and hardware parameters, the first step partitions and schedules the input data across multiple processors with concurrency and data locality in mind, and outputs a parallel scheduling table, based on which the second step generates parallel translation C code.

![Figure 2](image2.png)

Figure 2. Automatic parallel code generation for NUFFT data translation.

The algorithmic parameters of interest in NUFFT include source input size N, oversampling factor α, region size L x L of the data space (2D case), convolution window W, and kernel function C, whereas the hardware parameters include the number of cores and the size of the nearest local memory (or cache) for each core. Users need to provide source data array by sample coordinates, data values themselves, and weights (Dutt1993); the target point coordinates can be generated based on N, α, and L.

Since the sources are in a random order in the input array (stored in the memory or in a file) but data partition is performed on the logical data space, a process called Binning is applied to reorder the source data in the actual storage (file or memory) based on
tiling, which makes the data ready for the generated code to operate on. On the other hand, since the targets are regularly spaced, they can be stored in an order that favors the FFT computation to be performed next, such as column or row major orders in the 2D case.

3.1 Data Parallelization

Although the sources are always irregularly spaced, their distribution shape can be classified into two categories, namely, Uniform Density and Non-Uniform Density, depending on whether they are uniformly or non-uniformly distributed across the entire data space. We want to emphasize that our data parallelization strategy is applicable to both cases.

3.1.1 Parallelization Algorithm

The parallelization of data translation contains two key components, data partitioning (including tiling and binning) and parallel code scheduling. For numerical correctness, they must be designed to support mutual exclusion in data updates during concurrency exploration, i.e., two cores should not try to update the same targets of the sources being processed at the same time. We have developed an optimized Checkerboard Tiling and Scheduling scheme for achieving this, while ensuring workload balancing across the cores. Although our partitioning is based on the sources, with convolution window \( W \), targets are also implicitly partitioned based on the source tiles.

A naive checkerboard tiling and scheduling scheme (which is referred to, in the rest of this paper, as the baseline scheme, against which we compare our locality aware scheduling) satisfies the mutual exclusion condition, however, it may not be scalable with the increasing number of processors, and has unbalanced workload for the non-uniform density input. For example, given 4 processors, a data space can be divided into 16 equally-sized tiles, as shown in Figure 3 a). Tiles numbered 1 and 5 are scheduled on 4 processors at first to run concurrently, and then tiles with numbers 2 and 6 are processed. Tiles marked with numbers 3 and 7 and those with numbers 4 and 8 are executed subsequently. However, if we are given 5 processors (instead of the original 4), it is not easy to design a checkerboard pattern using this simple method, though it can fully exploit the parallelism.

Instead, we have designed an optimized checkerboard tiling and scheduling scheme, which can support any number of processors in the architecture. It operates as follows. Given \( P \) processors, the data space is first partitioned into tiles by a \( 2P \times 2P \) checkerboard pattern, assuming that the underlying 2D region is square. Then, the tiles are scheduled onto \( P \) processors as illustrated in Figures 3 a) and b), with \( P = 2 \) and \( P = 3 \), respectively. The number in each tile indicates the execution order. Tiles labeled using the same number will be processed in parallel by multiple cores. Example distributions in this figure are highlighted using ellipses. Yet a third option would be to partition the data space into \( 1 \times 2P \) as shown in Figure 3 c). However, intuitively, we prefer square tiles over rectangular ones since a square region has better geometric locality that can help us achieve more target reuse among the sources within a tile.

We want to emphasize that, our scheduling strategy not only supports concurrency, but also supports target data reuse between tiles. Specifically, it can be observed that each core executes a group of tiles in a rectangular region. If the sum of the source and target data sizes is smaller than the available on-chip memory (local memory or cache) capacity, data locality between the tiles can be exploited. In addition, our approach targets large input data sizes, in which case the convolution window \( W \) is much smaller than the size of a tile, and thus the concurrently-executed tiles meet the mutual exclusion condition, as illustrated using pink color of affected target regions in Figure 3 b). Mutual exclusion is also guaranteed through synchronization for each group of execution. If there are \( P \) processors, our checkerboard tiling and scheduling will need \( 4 \times P \) number of synchronizations.

The parallelization strategy explained so far works well for uniform density sources. However, it may lead to unbalanced workloads if the sources are non-uniformly distributed. In this case, an enhanced recursive approach is applied, as illustrated in Figure 3 d). Since a source is involved in the convolution computation of a fixed number of targets within \( W \), the workload of each tile is proportional to the number of sources it has. As a result, we can measure the standard deviation of sources among each group of concurrent tiles using the same number. If this deviation turns out to be larger than a preset value \( \delta \) in a parallel group (which indicates severe workload imbalance), the tiles in this group are sequentially executed. However, each tile will be further partitioned using the original tiling pattern and scheduling. For example, in d), the left tile with number 5 is processed by all cores before the right tile with number 5. For each of them, the same tiling and scheduling among two processors is further applied. This recursive approach stops when there is no source samples in a tile, or the standard derivation is under the threshold.

The selection of threshold \( \delta \) has an impact on the recursion depth as well as synchronization frequency. A smaller \( \delta \) usually will incur more synchronization overheads and higher partitioning latency due to increased recursion depth; however, it is also expected to lead better workload balancing. Thus, there is tradeoff between synchronization and workload balance. Proper value of \( \delta \) has to be chosen in order to improve overall performance.

Based on data partitioning and scheduling pattern, a parallel tile scheduling table for multiple processors can be generated, as shown in Figure 3 e). At each step, processors execute tiles in a column in parallel, and move to the next column after synchronization. The tile region size may not be the same for different columns in applying recursive approach for non-uniform density case, however they must be the same within the same column. It needs to be noted however that equally-sized tiles can have different number of sources or workloads. Using this table (shown in Figure 3 e), the code generator can easily generate parallel code for data translation computation.
3.1.2 Locality Optimization

Another important feature of our parallel algorithm distinguished from a naive checkerboard scheme is that a data locality optimization is applied to each tile generated, which utilizes on-chip memory characteristics to reduce expensive off-chip memory accesses.

As mentioned earlier, source data is partitioned into tiles through a binning process in the actual storage. However, within a tile, the source samples are stored in random order. If the total size of the source and target data is huge, e.g., much larger than a L1 cache of a processor, data locality will be deteriorated because of both increased capacity and conflict misses. To reduce the execution time of each tile on a processor, a recursive partitioning/tiling with Quad Pattern and Geometric Neighborhood scheduling is applied to reorder the source samples within each tile, so that data locality can be exploited to a large extent through target reuse.

For a tile of size $L \times L$, a partitioning with quad pattern is performed recursively to divide the original region into sub-tiles. It stops until the termination condition $N_s \ast width(source) + N_t \ast width(target) < M$ is satisfied, as shown in Figure 4 a).

Here, $N_s$ stands for the number of sources in a sub-tile, $N_t$ stands for the estimated number of targets involved in the convolution with sources, and $M$ is a threshold to be set to a value which is at least less than the size of an on-chip memory that is directly connected to a processor, e.g., a L1 cache. If an on-chip memory is connected to $P > 1$ processors, $M$ is set to be less than $T/P$ for each processor. A sub-tile has smaller data set size and region size compared to the original tile; and thus, it can reduce capacity competition between the sources and targets for on-chip memory, and also improve data locality if processing a sub-tile in a subset.

During partitioning, the source samples are reshuffled at each recursive level in the actual storage. Thus, $N_s$ can be obtained directly for each sub-tile. As to $N_t$, although the targets are regularly spaced, the number of targets needed in the computation may vary. If a sub-tile has very sparse sources, $N_t$ is estimated as $N_s \ast q^2$ (2D), where $q$ is the number of targets that a source affects in each dimension. If a sub-tile has dense sources, $N_t$ is calculated as $\alpha \ast N_s \ast (h + q \ast d)^2 / L^2$, where $\alpha$ is the oversampling factor, $N_s$ is the source input size, $h$ is the length of the one side of the sub-tile, $d$ is the target grid length, and $L$ is the length of the one side of the original tile, as shown in Figure 4 b). In this calculation, $\alpha \ast N_s$ is the total number of targets, and $\alpha \ast N_s(L + L)$ is the target density. In the case of non-uniformly distributed sources, $N_t = \min(N_s \ast q^2, \alpha \ast N_s \ast (h + q \ast d)^2 / L^2)$ which is suitable for both sparse and dense sub-tiles. In the case of uniformly distributed sources, $N_t = \alpha \ast N_s \ast (h + q \ast d)^2 / L^2$ is always used to estimate targets since oversampling factor, e.g., less than 2) is usually smaller than $q$ (e.g., 5).

After partitioning a tile, a neighborhood traversal shown in Figure 4 c) is used to schedule all sub-tiles so that target reuse between sub-tiles is exploited to improve data locality. A neighborhood order among all final sub-tiles can be always obtained when partition stops, independent of the partition shape. This is because, at each recursive level, the quad-pattern is always applied to a region which has the best neighborhood characteristics among its four child regions.

There are many possible traversal algorithms that can be employed to obtain a neighborhood ordering. In this work, we have designed and implemented one that executes in a top-down fashion recursively. Note that the sequence of four child nodes in a quad-pattern is represented by two factors, namely, the starting node and traverse direction (clock wise or counter-clock wise). In a sequence, we indicate that the starting node has an order 1, and the last node has an order 4; the second and third nodes have orders 2 and 3, respectively. As shown on the left side of Figure 4 c), the algorithm starts from the four nodes indicated using red numbers at the top level. Since they do not have a parent node, their sequence can be chosen arbitrarily. However, at each other recursive level, the sequence of four nodes are determined by the order of their parent node. The rules used for this can be listed as follows:

- If the parent node has order 1 (e.g., node 1 in red), start at a child node (node 1 in green) that has the same geometric location as the parent node, and end at a child node (node 4 in green) that has the same geometric location as the parent’s following node (node 2 in red).

- If the parent node has orders 2 or 3 (e.g., node 2 in pink), start at a child node (node 1 in blue) that is adjacent to the parent node’s previous node (node 1 in pink), and use the same direction (counter-clock wise) as used in the sequence of the parent node (counter-clock wise). In the cases where two nodes can be chosen as starting node (both nodes 1 and 4 in blue are adjacent to node 1 in pink), the choice is arbitrary.

- If the parent node has order 4 (e.g., node 4 in red), start at a child node (node 1 in green) that has the same geometric location as the parent node’s previous node (node 3 in red), and end at a child node (node 4 in pink) that has the same geometric location as the parent node.

By traversing recursively using these rules, a sequence of sub-tiles in geometric neighborhood order is obtained as shown on the right part of Figure 4 c). Figure 4 d) illustrates a neighborhood order of sub-tiles in a two-level partitioning with uniform density sources.

We apply this locality optimization with quad-pattern partition and neighborhood scheduling to sources for each tile that is assigned to a processor, as the link lists shown in the "parallel tile scheduling table" in Figure 5. This optimization is customized according to target on-chip memory size. It reduces the execution time of each tile on every processor and thus improves overall data translation performance. Moreover, since the scheduling is also a recursive process, data partitioning and scheduling can be accomplished together in one pass.

3.2 Code Generation

The code generation step takes as input the tile scheduling output from previous step and generates parallel code for multiple processors. The parallelization is only applied to the outer loop over
source tiles, as discussed in Section 2. The inner loop and convolution computation are encapsulated into codelets, parameterized with variables such as \( W \). We assume that different codelets are available for different kernel functions for convolution. The code generator generates complete code by using interfaces of codelets. Also, the user can specify whether MPI or OpenMP code is to be generated.

4. Experimental Results

We have evaluated the multi-threaded OpenMP data translation code generated by our automation tool for shared memory system, using the Simics simulator modeling four different multicore architectures, as shown in Figure 5. All 512KB caches have an associativity of 8, and the others have an associativity of 2. The memory access latency is 200 cycles, and cache access latency at each level is marked in the picture. Both uniform and non-uniform density input data of size \( 1K \times 1K \) and \( 2K \times 2K \) are tested. Specifically, the non-uniform input data has a sample distribution shape as depicted in Figure 5 f), where 90\% samples are scattered mostly in two regions. The data space has a region size of \( L \times L \), where \( L = 10 \). The oversampling factor is set to 2, and the window size \( q \) is equal to 5 in each dimension. The convolution kernel we used is Gaussian Kernel.

Figure 5. Four different multicore architectures used in the simulations.

The performance of the code generated by our automation tool depends largely on parallelization algorithms. In our algorithm, the data space is divided into 64 tiles by a \( 8 \times 8 \) checkerboard with locality optimization applied to each tile. We compared our scheme with the naive one mentioned earlier. Figure 6 shows the performance improvement brought by our algorithm over the baseline scheme for the uniform density case on four different architectures, with input sizes \( 1K \times 1K \) and \( 2K \times 2K \). It can be seen that our algorithm is more efficient on architectures 3 and 4 than on architectures 1 and 2, achieving almost 70\% and 80\% improvements for \( 1K \times 1K \) and \( 2K \times 2K \) input sizes, respectively. The same conclusion can be made for the non-uniform density case too in Figure 7. Also, our algorithm brings higher speedups with denser inputs, e.g., the performance improvement with \( 2K \times 2K \) is larger than that with \( 1K \times 1K \) input.

When comparing the corresponding numbers in these two bar-charts, we also observe that the performance improvement for uniform density is 3 to 6 points higher than non-uniform density. This is reasonable as uniformly distributed sources have less target reuse compared to the non-uniform case of the same input size, which can be exploited by our locality optimization algorithm. We further checked L1 and L2 cache statistics to verify this conclusion. Note that since convolution computation only updates target samples and there is always a target read before a target write (as shown in the

Figure 6. Performance improvement over baseline parallel algorithm for input sizes \( 1K \times 1K \) and \( 2K \times 2K \) on four architectures for uniform density source.

Figure 7. Performance improvement over baseline parallel algorithm for input sizes \( 1K \times 1K \) and \( 2K \times 2K \) on four architectures for non-uniform density source.

for loop in Section 2), the data write hit ratios of all caches are around 99.5\%. The locality due to target reuse is captured by the data read hit ratio. Table 1 shows the average L1 data read hit ratio improvement brought by our algorithm over the baseline scheme, with input size \( 1K \times 1K \) under both uniform and non-uniform density cases. It can be seen that our locality optimization scheme can improve cache performance significantly on architectures 2 and 3 with private L1 caches. Shared L1 caches in architectures 1 and 4 introduce more conflict misses. From this table, we observe that L1 data hit ratio on architecture 2 is as high as that on architecture 3; however, the performance improvement on architecture 2 is not as high as that on architectures 3 and 4. This is because the L2 cache data read hit ratios are also improved when using our algorithm by 15% and 22% on architectures 3 and 4, respectively.

Table 1. L1 data read hit ratio improvements for input size \( 1K \times 1K \) of both uniform and non-uniform density cases.

<table>
<thead>
<tr>
<th>Input Shape</th>
<th>Arch1</th>
<th>Arch2</th>
<th>Arch3</th>
<th>Arch4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform Density</td>
<td>0.8%</td>
<td>7.8%</td>
<td>7.6%</td>
<td>3.8%</td>
</tr>
<tr>
<td>Non-uniform Density</td>
<td>0.5%</td>
<td>7.6%</td>
<td>7.4%</td>
<td>3.5%</td>
</tr>
</tbody>
</table>

For the non-uniform density case, besides locality optimization, workload balancing through recursive tiling can also improve performance. We evaluated and analyzed the individual effects of these two techniques on architecture 2 and the results are given in Figure 8. For input size \( 1K \times 1K \), out of 29\% overall improvement, 23\% are brought by the locality optimization, while 6\% are due to the workload balancing optimization. As for \( 2K \times 2K \), out of 40\% over improvement, 35\% and 5\% are due to the locality and workload balancing optimizations, respectively. We can see that our locality optimization significantly speeds up the execution time. In
addition, the recursion depth is typically very small (e.g., 2) which helps us reduce tiling and binning time as well as synchronization frequency.

![Performance Improvement by Workload Balancing](image)

Figure 8. Performance improvement comparison between locality and workload balancing optimizations on architecture 2 for non-uniform density input sizes $1K \times 1K$ and $2K \times 2K$.

5. Related Work

Due to lack of space, we discuss in this section only the most relevant work to this paper. Our idea of hardware-aware parallel code generation for NUFFT data translation is inspired by the work from FFTW (Frigo1999), (Frigo1998) and Spiral (Puschel2005), (Chellappa2008), (Puschel2006), both of which can generate optimal sequential and parallel FFT code for the target platforms. FFTW uses a planner to find a fast composition of codelets (those automatically generated by a special-purpose compiler) through dynamic programming, whereas Spiral employs a search/learning engine to decide fast implementations (those automatically generated from a SPL compiler). They both generate optimized code by manipulating matrix operator to find the best factorization for FFT transform. Instead, our approach generates optimal parallel code for data translation through data parallelization in the data space. We also avoid FFTW and Spiral's time consuming code tuning process on the target platforms by encoding architectural features in the data parallel algorithms. A relevant work of accelerating NUFFT data translation on GPUs has recently been presented in (Sorensen2008). They allocate registers as local buffers to hold fixed size target tiles, and assign target tiles to multiple processors. However, their solution only works well for the uniformly distributed sources. For the non-uniform density sources on the other hand, equally sized target tiles will contain different number of sources which can lead to significantly unbalanced workload distribution across the processors.

References