AUTOMATIC PARALLEL CODE GENERATION FOR NUFFT DATA TRANSLATION ON MULTICORES

YUANRUI ZHANG†, JUN LIU‡, EMRE KULTURSAY§ and MAHMUT KANDEMIR¶

Department of Computer Science and Engineering,
The Pennsylvania State University,
University Park, PA, 16802, USA
†yuazhang@cse.psu.edu
‡jxl1036@cse.psu.edu
§euk139@cse.psu.edu
¶kandemir@cse.psu.edu

NIKOS PITSIANIS|||†† and XIAOBAI SUN**‡‡

|||Department of Electrical and Computer Engineering,
Aristotle University, Thessaloniki, Greece
**Department of Computer Science,
Duke University,
Durham, NC, 27708, USA
††nikos@cs.duke.edu
‡‡xiaobai@cs.duke.edu

Received 25 July 2010
Accepted 23 January 2011

The nonuniform FFT (NuFFT) is widely used in many applications from embedded systems to scientific computing domain. Focusing on the most time-consuming part of the NuFFT computation, the data translation step, in this paper, we develop an automatic parallel code generation tool for data translation targeting emerging multicores. The key components of this tool are two scalable parallelization strategies, namely, the source-driven parallelization and the target-driven parallelization. Both these strategies employ equally sized geometric tiling and binning to improve data locality while trying to balance workloads across the cores through dynamic task allocation. They differ in the partitioning and scheduling schemes used to guarantee mutual exclusion in data updates. This tool also consists of a code generator and a code optimizer for the data translation. We evaluated our tool on a commercial multicore machine for both 2D and 3D inputs under different sample distributions with large data set sizes. The results indicate that both parallelization strategies have good scalability as the number of cores and the number of dimensions of data space increase. In particular, the target-driven parallelization outperforms the other when samples are nonuniformly distributed. The experiments also show

*This paper was recommended by Regional Editor Alex Jones.
†Corresponding author.
that our code optimizations can bring about 32%–43% performance improvement to the data translation step of NuFFT.

Keywords: Nonuniform FFT; data translation; convolution; automatic parallelization; code generation.

1. Introduction

Fourier transform has been playing an important role in a variety of applications, from medical imaging\(^1\) to telecommunications,\(^2\) synthetic radar imaging,\(^3\) and geoscience and seismic analysis.\(^4\) When the sampling is uniform, e.g., the Fourier transform is desired at equispaced frequencies, the conventional fast Fourier transform (FFT) can be directly used. However, when the sampling in the data or frequency domain is unequally spaced, the conventional FFT does not apply. Over the past years, a number of algorithms have been developed to overcome this limitation — generally referred to as nonuniform FFTs (NuFFTs).\(^5\)–\(^7\) They achieve the same \(O(N \log N)\) computational complexity as the FFT by first applying a local support convolution that translates the sample points from the nonCartesian grid to the Cartesian grid, and then performing the conventional FFT on the translated Cartesian points. The former step is also called data translation or resampling, whose complexity is linear to the size of the sample ensemble.

Although there are two other operations, the density compensation and the scaling, at the beginning and the end of the NuFFT computation, respectively, the performance of NuFFT is determined mostly by the data translation and the FFT steps. Efficient software and hardware implementations of the FFT have already been well developed, including different libraries, field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs). In contrast, much less work has been done on the data translation. Although the data translation has lower arithmetic complexity compared to the FFT, this computation is more time-consuming since it has irregular data access patterns that lead to very poor memory performance in modern parallel architectures.\(^8\)

Consequently, instead of trying to accelerate the data translation from a mathematical point of view as the prior work\(^5\)\(^,\)\(^9\)\(^–\)\(^11\) have done (e.g., reaching the linear complexity of data translation by novel design of the translation kernel functions), we attempt to improve its memory performance on multicores through parallelization strategies that take into account architectural features, such as cache sizes and data access latencies. However, as multicore designs have been emerging and evolving in increasing diversity and complexity, manual parallelization and fine-grain performance tuning for each target architecture become impractical. Traditional compiler-based parallelization techniques\(^12\) cannot help in this case, since those techniques work well mostly for regular dense matrices, whereas data translation is essentially a matrix-vector multiplication with an irregular and sparse matrix.\(^a\) As a

\(^a\)Since the matrix is very sparse, the computation time of matrix-vector multiplication (data translation) is linear to the vector length (the size of the sample ensemble).
result, automated parallelization and code generation that can efficiently utilize the underlying architecture characteristics and exploit the intrinsic parallelism in the application become urgent and necessary for this frequently used computation.

Motivated by this, we present and experimentally evaluate in this work an automatic parallel code generation tool for the NuFFT data translation, called NUDT, on multicores. Specifically, we make the following contributions:

• We design two scalable parallelization strategies, namely, the source-driven parallelization and the target-driven parallelization, which employ equally sized tiling and binning to cluster the unequally spaced source samples to improve data locality. Both these strategies ensure load balance across the cores through dynamic task allocation, and guarantee mutual exclusion in data updates via different partitioning and scheduling schemes. They can be adapted to different target architectures as well as different input sample distributions.

• We develop an automatic parallel code generator and code optimizer for the data translation computation. In particular, the code generator generates Pthread or OpenMP code by selecting the codelet of kernel function prepared beforehand, and combining it with other data structures and code that operate on the partitioned data. Although our code generator includes several optimizations, we provide a code optimizer to further improve the generated code at a high level, e.g., by inserting software prefetches into the program.

• We present an experimental evaluation of our parallelization strategies and generated code for both 2D and 3D inputs under different sample distributions with large data set sizes, on a commercial multicore machine. Specifically, we explore the relationship between tile sizes and cache sizes, investigate the effectiveness of different code optimizations and the efficiency of the two parallelization strategies, and study the execution times of the data translation and the FFT from FFTW. Our experiments indicate that both our proposed schemes have good scalability as the number of cores and the number of dimensions of data space increase. In particular, the target-driven parallelization outperforms the other when samples are nonuniformly distributed. The experiments also show that our code optimizations can bring about 32%–43% performance improvement to the data translation computation.

The rest of this paper is organized as follows. Section 2 explains the NuFFT data translation algorithm and its basic operations. Section 3 gives an overview of our proposed NUDT tool. Sections 4–6 describe the four major parts of our tool: geometric tiling and binning, data translation parallelization, code generation, and code optimization, respectively. Section 7 presents the experimental results. Section 8 discusses the related work, and finally, Sec. 9 gives our concluding remarks.

b This journal paper extends our prior work presented in Refs. 13 and 14. Compared to Refs. 13 and 14, this paper presents a complete tool with code optimizations as well as all new 1D and 3D results.
2. Background

2.1. Data translation algorithm

Data translation algorithms vary in the type of data sets they target and the type of
regridding or resampling methods they employ. In this paper, we focus primarily on
convolution-based schemes for data translation, as represented by Eq. (1), where the
source samples $S$ are unequally spaced, e.g., in the frequency domain, and the target
samples $T$ are equally spaced, e.g., in an image domain. The dual case with equally
spaced sources and unequally spaced targets can be treated in a symmetrical way.

$$v(T) = C(T, S) \cdot q(S).$$

In Eq. (1), $q(S)$ and $v(T)$ denote input source values and translated target values,
respectively, and $C$ represents the convolution kernel function. The set $S$ for source
locations can be provided in different ways. In one case, the sample coordinates are
expressed and generated by closed-form formulas, as in the case with the sampling on
a polar grid (see Fig. 1(a)). Alternatively, the coordinates can be provided in a data
file as a sequence of coordinate tuples, generated from a random sampling (see
Fig. 1(b)). The range and space intervals of the target Cartesian grid $T$ are specified
by the algorithm designer, and they can be simplified into a single oversampling
factor $\alpha$ since the samples are uniformly distributed. With an oversampling factor of
$\alpha$, the relationship between the number of sources and targets can be expressed as
$|T| = \alpha |S|$.

The convolution kernel $C$ is obtained either by closed-form formulas or numerically.
Examples for the former case include the Gaussian kernel and the central
B-splines, whereas those for the latter case include functions obtained numerically
according to the local least-square criterion and the min–max criterion. In each
case, we assume that the function evaluation routines are provided. In addition, the
kernel function is of spatial local support, i.e., each source is only involved in the
convolution computation with targets within a window, and vice versa. Figure 1(c)
shows an example for a single source in the 2D case, where the window has a side
length of $w$.

Fig. 1. Illustration of different sampling schemes and local convolution window. (a) A Polar grid
sampling, (b) a random sampling, and (c) convolution window for a source sample with side length $w$. 
2.2. Basic data translation procedure

The above data translation algorithm can be described by the following pseudo code:

```
BasicDataTranslationProcedure(S):
    for each source s_i in S
        for each target t_j in \( T \cap \text{window}(s_i) \)
            \( V(t_j) + = c(t_j, s_i) \times Q(s_i) \),
```

where \( Q \) is a one-dimensional array containing the source values regardless of the geometric dimension, \( V \) is a \( d \)-dimensional array holding the target values, and \( c \) represents a kernel function over \( T \times S \), e.g., the Gaussian kernel \( e^{(t_j-s_i)/\sigma} \), where \( |t_j - s_i| \) denotes the distance between a target and a source. The target coordinates are generated on-demand during the computation based on the source coordinates, the oversampling value (\( \alpha \)), and the size of the data space, e.g., \( L \times L \) in the 2D case. In the pseudo code given above, the outer loop iterates over sources. We name this type of computation the source domain-driven computation, which is driven by the source domain. The alternate computation is the target domain-driven computation, in which case the outer loop iterates over targets. Clearly, the source domain-driven computation is preferred since it is easy to find the specific targets on the Cartesian grid for a source within the convolution window, but not vice versa. The complexity of the code is \( O(w^d \times |S|) \); however, since \( w \) is usually very small (i.e., \( w^d \) is near constant), the data translation time is linear with the number of sources \(|S|\).

3. The Tool Overview

Before going into the technical details, in this section, we give an overview of our NUDT tool, as illustrated in Fig. 2. The tool hosts four major components (or steps): geometric tiling and binning, data translation parallelization, code generation, and code optimization. Given user input algorithmic parameters and architectural descriptions, the first step employs an equally sized tiling scheme to cluster the original source samples into cells/tiles based on their spatial locations and reschedules the data in the actual storage space through binning to improve data locality. The second step then applies one of the two parallelization strategies we developed, namely, the source-driven parallelization and the target-driven parallelization, to partition and schedule tiled source data as well as uniform target data on multicores. After that, the third step generates the Pthread- or OpenMP-based parallel code (depending on user specification) according to the parallelization and user chosen kernel function (the codelets of kernel functions are provided beforehand). Finally, the fourth step performs high-level code optimizations, such as inserting software prefetches, to further improve the performance of the output code on the target machine.

The algorithmic parameters of interest include the number of sources (input size) \( N \), the oversampling factor \( \alpha \), the size of the data space (e.g., \( L \times L \) of a 2D region),
the convolution window \( w \), and the kernel function \( c \). Note that, the target point coordinates can be generated based on \( N, \alpha \), and the size of the data space on the fly. The architectural parameters of interest include the number of cores, the size and access latency of each level of caches. Since we currently target homogenous multi-core systems with symmetric on-chip cache hierarchy, our tool employs a tuple form to facilitate describing the basic architectural features or information needed, for example, \((8, 3); (32\, KB, 3\, cycles); (128\, KB, 20\, cycles); (6\, MB, 250\, cycles)\), where the first tuple \((8, 3)\) gives the number of cores and the number of cache levels, and the subsequent tuples give the size and access latency of L1, L2, and L3 level caches, respectively.

\[ a_{ij} = e^{\frac{1}{2} \left| t_i - s_j \right|^2 / \sigma} \]

with the Gaussian kernel. While the sparsity stems from the local window effect of the kernel function, the irregularity is caused by the unequally spaced sources. The conventional tiling on dense and regular matrices\(^{12}\) cannot help to achieve high data reuse in this case. For instance, the source samples \( s_1 \) and \( s_2 \) from the same tile by conventional matrix tiling as shown in

4. NUDT Geometric Tiling and Binning

As stated earlier, the first step in our tool is geometric tiling and binning. This step is necessary for the convolution-based data translation, which is essentially a matrix-vector multiplication with sparse and irregular matrices. Figure 3(a) shows an example convolution matrix, where rows are formed by the target points from the data space and columns are formed by the source points. Each entry \( a_{ij} \) in the matrix represents the value after applying kernel function to the corresponding pair of source and target points, e.g., \( a_{13} = e^{\left| t_1 - s_3 \right|^2 / \sigma} \) with the Gaussian kernel. While the sparsity stems from the local window effect of the kernel function, the irregularity is caused by the unequally spaced sources. The conventional tiling on dense and regular matrices\(^{12}\) cannot help to achieve high data reuse in this case. For instance, the source samples \( s_1 \) and \( s_2 \) from the same tile by conventional matrix tiling as shown in
Fig. 3, may update different targets located far away from each other in the data space as indicated in Fig. 3(b), which means the affected targets by source samples $s_1$ and $s_2$ are not in the same matrix tile as $s_1$ and $s_2$. To exploit the data reuse, the geometric tiling in the data space is needed, which clusters the sources into cells/tiles based on their spatial locations, as illustrated in Fig. 4. Since the source samples in one geometric tile are spatially close to each other, there is a large chance that they update the same target points in their overlapped windows. Therefore, the target data reuse within each geometric tile can be exploited. The geometric tiling may be equally sized (as in Fig. 4(a)) or unequally sized (as in Fig. 4(b)). The latter can be
obtained through adaptive tiling based on the sample distribution. In either case, the data translation procedure can be rewritten as

for each nonempty source tile $T_k$ in $S$.

Call \textit{BasicDataTranslationProcedure}(T_k).

Associated with geometric tiling is a process called \textit{binning}. Basically, binning reshuffles the source data in the storage space, e.g., external memory or file, according to geometric tiles. In terms of data movements, the complexity of an equally sized tiling and binning is $O(|S|)$, whereas the cost of an adaptive recursive tiling and binning is $O(|S| \log |S|)$. The latency of the latter increases dramatically when the data set size is very large, as observed in Ref. 13. Consequently, in this work, we adopt the equally sized tiling, irrespective of the source distribution, i.e., uniform distribution or nonuniform distribution, as shown in Fig. 4. Equally sized geometric tiling essentially transforms the nonCartesian grid of sources to a Cartesian grid of tiles in the data space. From the convolution matrix point of view, it induces an unequal partitioning among columns (since each tile may hold different numbers of sources) and an equal partitioning among rows (since targets are uniformly distributed).

5. NUDT Parallelization

The second step is the data translation parallelization, which is performed on the data space with transformed Cartesian grid of source tiles. Two critical issues that need to be considered during this phase are mutual exclusion of data updates and load balancing. On the one hand, a direct parallelization of the source tile loop for the code in Sec. 4 (as well as the source loop for the code in Sec. 2.2) can lead to incorrect results, since threads on different cores may attempt to update the same target when processing geometrically nearby sources concurrently. Though parallelizing the inner target loop can avoid this problem, it would cause significant synchronization overheads. On the other hand, a simple even partition in the data space can lead to unbalanced workloads across the cores, since the obtained data regions may contain different numbers of sources, especially when input samples are nonuniformly distributed.

To address these issues, we designed two parallelization strategies that aim to accelerate the data translation on multicores. One of these strategies is called the \textit{source-driven parallelization}, and the other is referred to as the \textit{target-driven parallelization}. Our observation is that, a data space containing both sources and targets can be viewed as two different domains that are overlapped with each other, where one contains only sources and the other contains only targets. An explicit partitioning of one domain will induce an implicit partitioning of the other domain because of the convolution window effect. Our source-driven parallelization carries out parallel partitioning and scheduling in the source domain, whereas our
target-driven parallelization operates on the targets. In both cases, the source
domain has been transformed into a Cartesian grid of tiles through geometric tiling
and binning (although inside each tile, the samples are still unequally spaced). Note
that, both these strategies are used in the context of the source domain-driven
computation, i.e., the outer loop iterates over sources in the computation code. To
ensure mutual exclusion for target updates, the first strategy employs a special
parallel scheduling in the source domain, whereas the second strategy employs a
customized partitioning in the target domain. Both schemes use recursive approach
and dynamic task allocation to achieve load balance across the cores.

5.1. Source-driven parallelization

In the first strategy, when partitioning the source domain, a special scheduling is
indispensable to guarantee the mutual exclusion of target updates. Consider Figs. 5
and 6 for example. No matter how the sources in a 2D region are partitioned, e.g.,
using one-dimensional or two-dimensional partition, adjacent regions cannot be
processed at the same time due to overlapped target windows of boundary samples,

![Fig. 5. One-dimensional partition with a two-step scheduling for the 2D case.](image)

![Fig. 6. Two-dimensional partition with a four-step scheduling for the 2D case.](image)
as indicated by the dashed lines. However, if further dividing the neighboring source regions into smaller blocks according to the target overlapping patterns, a scheduling can be found to process those adjacent source regions through several steps, where at each step, the sub-blocks with the same number (indicated in the figures) can be executed concurrently, and a synchronization is inserted as moving from one step to another. Our observation is that a one-dimensional partition needs a two-step scheduling to eliminate the contention, whereas a two-dimensional partition needs four steps. In general, an $x$-dimensional partition ($1 \leq x \leq d$) requires a $2^x$-step scheduling to ensure correctness.

Based on this observation, our source-driven parallelization is designed as follows. Given $m$ threads and a $d$-dimensional data space, we first factorize $m$ into $p_1 \times p_2 \times \cdots \times p_d$, and then divide dimension $i$ into $2p_i$ segments ($1 \leq i \leq d$), which results in $2p_1 \times 2p_2 \times \cdots \times 2p_d$ blocks in the data space, and finally schedule every $2^d$ neighboring blocks using the same execution-order pattern. Figure 7(a) illustrates an example for the 2D case with $m = 16$ and $p_1 = p_2 = 4$. The blocks having the same time stamp (number) can be processed concurrently, provided that the window side length $w$ is less than any side length of a block. In the case where $m$ is very large and this window size condition no longer holds, some threads will be dropped to decrease $m$, until the condition is met. Although a similar $d'$-dimensional partition ($d' < d$) with a $2^{d'}$-step scheduling can also be used for a $d$-dimensional data space, e.g., a one-dimensional partition for the 2D case, as shown in Fig. 7(b), it is not as scalable as a $d$-dimensional partition, especially when $m$ is very large.

The scheme explained so far works well with uniformly distributed sources, but not with nonuniformly distributed sources, as it can cause unbalanced workloads in the latter case. However, a slight modification can fix this problem. Specifically, since the amount of computation in a block is proportional to the number of sources it contains, the above scheme can be recursively applied to the source domain until the

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
2 & 3 & 2 & 3 & 2 & 3 & 2 & 3 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
2 & 3 & 2 & 3 & 2 & 3 & 2 & 3 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
2 & 3 & 2 & 3 & 2 & 3 & 2 & 3 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
2 & 3 & 2 & 3 & 2 & 3 & 2 & 3 \\
\end{array}
\]

(a)  \hspace{5cm}  \begin{array}{cccccccc}
01 & 01 & 10 & 10 & 10 & 10 & 10 & 1 \\
\end{array}

(b)

Fig. 7. Illustration of nonrecursive source-driven parallelization in the 2D case. (a) A two-dimensional partition with a four-step scheduling for $m = 16$, $p_1 = p_2 = 4$ and (b) a one-dimensional partition with a two-step scheduling for $m = 16$. 
number of sources in a block is less than a preset value ($C14$). The recursion needs to ensure that the window size condition is not violated. Figure 8(a) depicts an example of two-level recursive partitioning for the 2D case, where $m = 4$, $p_1 = p_2 = 2$, and the dark regions are assumed to have dense samples. The corresponding concurrent schedule is represented by the table shown in Fig. 8(b). The synchronization takes place every time after processing each group of blocks pointed by a table entry. However, within each group, there is no processing order for the blocks, which will be dynamically allocated to the threads at runtime. The selection of threshold $C14$ has an impact on the recursion depth as well as the synchronization latency. A smaller value of $C14$ usually leads to deeper recursions and higher synchronization overheads; but, it is also expected to have better load balance. Consequently, there is a tradeoff between minimizing synchronization overheads and balancing workloads, and careful selection of $C14$ is important for the overall performance.

5.2. Target-driven parallelization

Our target-driven parallelization performs recursive partitioning and scheduling in the target domain, which has the advantage of requiring no synchronization. Given $m$ threads and a $d$-dimensional data space, this scheme employs a $2^d$-branch geometric tree to divide the target domain into blocks recursively until the number of sources associated with each block is below a threshold ($C14$), and then uses a neighborhood traversal to obtain an execution order for those blocks, based on which it dynamically allocates their corresponding sources to the threads at runtime without...
any synchronization. Figure 9(a) shows an example for the 2D case with quadtree partition in the target domain and neighborhood traversal and (b) an illustration of aligned window to the source tiles for a target block. Especially, (b) shows the Cartesian grid of tiles and a target partition with bolded lines, where the window of a target block is aligned to the tiles, indicated by red color.

Fig. 9. (Color online). Example of target-driven parallelization in the 2D case. (a) An example of quadtree partition in the target domain and neighborhood traversal and (b) an illustration of aligned window to the source tiles for a target block. Especially, (b) shows the Cartesian grid of tiles and a target partition with bolded lines, where the window of a target block is aligned to the tiles, indicated by red color.

Finding the associated sources for a particular target block is not as easy as the other way around. Typically, one needs to compare the coordinates of each source with the boundaries of the target block, which takes $O(|S|)$ time. Our parallelization scheme reduces this time to a constant by aligning the window of a target block with the Cartesian grid of tiles, as depicted in Fig. 9(b), where the source tiles and thus sources associated with this target block are found directly. This method attaches irrelevant sources to each target block; however, the introduced execution overheads can be reduced by choosing proper tile size.

6. NUDT Code Generation and Optimization

After parallelization, the third and fourth steps generate and output an optimized parallel code of data translation for the user, which will be executed on the target multicore machine.

In particular, the code generator selects the codelet (a piece of code/function) of the kernel function defined by the user, and creates a parallel C code of data
translation using Pthread or OpenMP (also specified by the user), based on the
previous parallelization. It is worth mentioning that the codelets of kernel functions
are prepared beforehand, and are highly optimized for computation performance.
One optimization we applied in these codelets (in our experimental evaluation tar-
getting Intel multicore architectures) is the vectorization with Intel AVX SIMD
instructions.\textsuperscript{17} Using vectorization, the performance of data translation is signi-
cantly improved, as much as 29\% in some cases. Another optimization for compu-
tation is the customized look-up table implementation employed for elementary
function evaluation. More specifically, we observe that the major components in
many kernel functions are elementary functions, which take a large fraction of the
execution time. However, in some execution scenarios, user does not require very high
accuracy for the data translation, in which case we can sacrifice the accuracy a little
in order to save time significantly. Consequently, as an option, we allow user to
invoke the “special purpose compiler” developed in our previous work\textsuperscript{18} to auto-
matically generate customized look-up table as well as C code for elementary func-
tions in the kernel function with desired accuracy and performance, instead of using
the default implementations provided by C libraries.

In addition to these optimizations, we also provide another separate step to
further improve the quality of the generated code at a high level. Currently, this step
inserts software prefetches\textsuperscript{19,20} into the code, which issues prefetches at a tile gran-
ularity for each core. The number of iterations and tiles to prefetch ahead of time is
determined by the source tile size, the window size $w$, as well as the access latency
and size of last-level caches. We quantify the effectiveness of all the optimizations
mentioned above in the following section.

\section{Experimental Evaluation}

\subsection{Setup}

We evaluated our NUDT code generation tool on a commercial multicore machine
with different source sample inputs. Specifically, the underlying platform is a dual
quad-core Intel Harpertown machine,\textsuperscript{21} which operates at a frequency of 3 GHz, and
has eight private L1 caches of 32 KB each and four shared L2 caches of 6 MB each.
The architecture is depicted in Fig. 10. We used six sample inputs listed in Table 1 to
evaluate our tool, which have different dimensions. The data set sizes (134M in all

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{architecture.png}
\caption{Intel Harpertown quad-core architecture.}
\end{figure}
cases) and source distributions are also shown in the table. The uniform distributions are generated using a random scheme, whereas the nonuniform distributions are generated based on the three closed-form equations as below:

\[
X = R \times \cos \theta \quad R \in \left(0, \frac{\sqrt{2}}{2} L\right), \quad \theta \in (0, 2\pi),
\]

\[
Y = R \times \sin \theta
\]

\[
\begin{align*}
X &= R \times \cos \theta \quad R \in \left(0, \frac{\sqrt{2}}{2} L\right), \quad \theta \in (0, 2\pi), \\
Y &= R \times \sin \theta \\
Z &= R \times (\sin \theta + \cos \theta)
\end{align*}
\]

In all six cases, the samples are unequally spaced. For algorithm parameters, we set the kernel function to the Gaussian kernel, the oversampling value \((\alpha)\) to 1.5 in each dimension, and the convolution window \((w)\) to 5 in each dimension, in terms of the number of targets affected, which means in 1D, 2D, and 3D cases, each source is involved in the computation of 5, 25, and 125 targets, respectively. The results presented in the next section are generated from Pthread codes output by our tool, which are compiled using `icc` with option `-O3` on the target machine.

### 7.2. Results

We first investigated the relationship between the tile size and the cache size (both L1 and L2) on the target platform, and analyzed the impact of the tile size on the performance of binning and data translation computation. The tile size is determined based on the cache size so that all the sources and their potentially affected targets in each tile are expected to fit in the cache. For nonuniform distributions, the tile size is calculated based on the assumption of uniform distribution. Figure 11(a) gives the execution times of binning and data translation with Input 5 (see Table 1), using the source-driven parallelization. The cache size varies from 8 KB to 8 MB. The four

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Size</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input 1</td>
<td>1D</td>
<td>134M</td>
</tr>
<tr>
<td>Input 2</td>
<td>1D</td>
<td>134M</td>
</tr>
<tr>
<td>Input 3</td>
<td>2D</td>
<td>11586 \times 11586</td>
</tr>
<tr>
<td>Input 4</td>
<td>2D</td>
<td>11586 \times 11586</td>
</tr>
<tr>
<td>Input 5</td>
<td>3D</td>
<td>512 \times 512 \times 512</td>
</tr>
<tr>
<td>Input 6</td>
<td>3D</td>
<td>512 \times 512 \times 512</td>
</tr>
</tbody>
</table>
groups depicted from left to right are the results obtained using 1 core, 2 cores, 4 cores, and 8 cores, respectively. For binning, the time decreases as the cache size (or tile size) increases, irrespective of the number of cores used, whereas for the data translation, the time increases when the tile size becomes large, since the target reuses in each tile get reduced. As far as the total execution time is concerned, we can see that, there is a lowest point in each group, which is around the tile size of 1 MB, one third of L2 cache size per core (3 MB) in the Harpertown architecture. A similar behavior can also be observed with Input 6, as illustrated in Fig. 11(b).

We also evaluated the efficiency of code optimizations employed by our tool. The normalized execution times of four different versions with different optimization levels are presented in Fig. 11(c). In particular, the original version does not apply any optimization; Version 1 employs the vectorization for the kernel function; Version 2 inserts software prefetches into the code based on Version 1; and Version 3
adopts the customized look-up table implementation for elementary function evaluation based on Version 2. We tested all these four versions using the source-driven parallelization under Input 5. The results demonstrate that vectorization brings significant performance improvement to the data translation computation, e.g., about 29% on 8 cores. Tile-wise software prefetching on the other hand only speeds up the data translation by about 2%–3%, since the target architecture already uses hardware prefetching. The customized look-up table implementation accelerates the data translation by around 8%–9%. In summary, the overall performance improvement brought by all code optimizations is about 32%–43% on average.

We then evaluated the two parallelization strategies proposed, using the best tile size from the first group of experiments and all code optimizations. Figures 12(a) and 12(b) plot their respective execution times under Input 6. We can see that, in both

![Graphs showing execution times under different core counts for different parallelization strategies](image)

(c)

Fig. 12. (a) Execution times of the source-driven parallelization under Input 6, (b) execution times of the target-driven parallelization under Input 6 and (c) data translation execution times with the two parallelization strategies under Input 5 and Input 6.
graphs, the execution time of data translation scales well with the number of cores; however, the execution time of geometric tiling and binning reduces much slower as the number of cores increases, since this process involves only data movements in the memory. Further, the two parallelization strategies have comparable data translation performance under the uniform distribution input, as indicated by the group of bars on the left in Fig. 12(c); however, the target-driven parallelization outperforms the other by 13% and 28% on 4 cores and 8 cores, respectively, under the nonuniform distribution input, as depicted by the group of bars on the right in Fig. 12(c). This difference will be larger if the source distribution becomes more unbalanced across the data space, since (in that case) there will be more recursive levels and synchronization overheads introduced in the source-driven parallelization.

We also studied the scalability of our approach as the number of dimensions of the data space increases. Specifically, we kept the number of sources the same, but changed the data space dimensionality. Figure 13(a) gives the data translation execution times with uniform (left group) and nonuniform (right group) distribution inputs, respectively, using the target-driven parallelization. The theoretical runtime for 1D, 2D, and 3D inputs are $5N$, $25N$, and $125N$, respectively, where $N$ is the number of sources. However, our results show that the execution times with Inputs 1 and 2 are about 5.1 times faster than those with Inputs 3 and 4, and Inputs 3 and 4 are 5.2 times faster than Inputs 5 and 6. This slight slowdown with the higher dimensions is due to the reduced target reuses when the same number of sources spread out in larger data spaces.

In addition, we studied the execution time of the data translation using the target-driven parallelization on the sources, and the execution time of the FFT obtained from FFTW with “FFTW_MEASURE” option on the translated

![Graph](image)  
**Fig. 13.** (a) Data translation execution times with the target-driven parallelization under uniform (left group) and non-uniform (right group) distributions, (b) execution times of the data translation and the FFT from FFTW when using Input 3 and (c) execution times of the data translation and the FFT from FFTW when using Input 5.
target points. The sum of these two parts gives an estimation on the total execution time of the NuFFT computation on the target machine. Figures 13(b) and 13(c) present the collected results under the uniformly distributed sources for 2D and 3D, respectively. The graphs show that the performance of the data translation and the FFT are closer in the 2D case than in the 3D case. The reason for this behavior is that, from 2D to 3D, the runtime of the data translation increases five times, from $25N$ to $125N$, whereas the runtime of the FFT increases only about 1.5 times, from $5\alpha^2N(\log \alpha^2N)$ to $5\alpha^3N(\log \alpha^3N)$, with $N = 134M$. This breakdown is useful for further investigation of pipelined implementation of the NuFFT computation on multicores.

8. Related Work

Most of the previous work for the NuFFT data translation (or convolution) focus on the algorithm aspect, e.g., the design of an efficient kernel function with high accuracy. Dutt and Rokhlin\textsuperscript{6} used the Gaussian function as the convolution kernel in the first NuFFT algorithm. Beylkin\textsuperscript{5} proposed applying the central B-spline to the kernel with analytically finite support. Liu and Nguyen\textsuperscript{10} employed the cosine function as the scaling function, and determined the convolution coefficients numerically by a local least-squares interpolation of the discrete unequally spaced Fourier transform matrix. Fessler and Sutton\textsuperscript{9} presented a min--max criterion as an interpolation method, which is optimal in the min--max sense of minimizing the worst-case approximation error over all signals of unit norm. Greengard and Lee\textsuperscript{11} accelerated a Gaussian-based scheme without precomputation of the interpolation weights.

The studies that take into account the architectural aspects to accelerate the NuFFT convolution include Refs. 8 and 24. Sorensen \textit{et al.}\textsuperscript{24} implemented a parallel algorithm targeting GPUs. They allocate registers as local buffer to hold target tiles,
where each tile contains a fixed number of targets, and distribute those target tiles to
different processors to perform the convolution for the sources inside each target tile.
The explicit memory loading of targets as well as the fine-granularity parallelization
(the target tiles are usually small because of register size) is suitable for a GPU
architecture but not for general-purpose cache-based multicores. In Ref. 8, Debroy
et al. discussed an effective parallelization approach to improve memory performance
for radial data on multicores. Their scheme is limited to this specific source sample
distribution, and consequently is not as general as our parallelization strategies.

Another area of research related to our work is domain-specific code generation
utilizing mathematical structures and architectural properties for performance
improvement and optimization. PHiPAC\textsuperscript{25} is an autotuning system for dense matrix
multiplication, which generates portable C code and search scripts to tune for specific
systems. ATLAS\textsuperscript{26,27} utilizes empirical autotuning to produce a cache-contained
matrix multiplication, which is then used for larger matrix computations in BLAS
and LAPACK. FFTW\textsuperscript{22} uses empirical autotuning to combine solvers for FFTs.
Other autotuning and code generation systems include SPARSITY\textsuperscript{28–31} for sparse
matrix computations, SPIRAL\textsuperscript{32} for digital signal processing, UHFFT\textsuperscript{33} for FFT on
multicore systems, OSKI\textsuperscript{34} for sparse matrix kernels, and autotuning frameworks for
optimizing sequential\textsuperscript{35,36} and parallel\textsuperscript{37} sorting algorithms. Different from all these
prior efforts, our work presents two new parallelization strategies for the NuFFT
data translation, and generates customized code for target platforms without
empirical tuning, but based on user input algorithmic and architectural information
(although the work in Ref. 31 is a model-driven autotuning for sparse matrix-vector
multiply on GPUs, this model still needs the empirical tuning framework in Ref. 29 to
select parameters first).

9. Conclusion

In this work, we proposed and evaluated an automatic parallel code generation tool
for the NuFFT data translation on multicores. The proposed NUDT tool hosts four
components: geometric tiling and binning, parallelization, code generation, and code
optimization. Specifically, we presented two parallelization strategies that utilize
equally sized geometric tiling and binning to improve data locality in shared cache-
based multicores. Both our schemes ensure the mutual exclusion in data updates and
load balancing across the cores. We evaluated our tool for both 2D and 3D inputs
with different sample distributions and large data set sizes on Intel Harpertown
multicore machine. In particular, we explored the relationship between tile size and

cache size, and investigated the effectiveness of different code optimizations and the
performance of the two parallelization strategies. Our experiments indicate that both
parallelization schemes scale well as the number of cores and the number of
dimensions of data space increase, and the target-driven parallelization outperforms
the other when source samples are nonuniformly distributed. The experiments also
show that our code optimizations can bring about 32%–43% performance improvement to the data translation. In the future work, we will investigate the application of our parallelization strategies to heterogeneous architectures.

Acknowledgments

This research is supported in part by NSF grants #1017882, #0963839, CNS #0720645, CCF #0811687, and CCF #0702519, a grant from DARPA (W911NF-05-1-0248) and a grant from Microsoft Corporation.

References

18. Y. Zhang, L. Deng et al., A special-purpose compiler for look-up table and code generation for function evaluation, in Proc. of DATE (2010).