FFT Program Generation for the Cell BE *

Srinivas Chellappa, Franz Franchetti, and Markus Püschel

Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh PA 15213, USA
{schellap, franzf, pueschel}@ece.cmu.edu

Abstract. The complexity of the Cell BE's architecture makes it difficult and time consuming to develop multithreaded, vectorized, high-performance numerical libraries. Our approach to solving this problem is to use Spiral, a program generation system, to automatically generate and optimize linear transform libraries for the Cell. To extend the Spiral framework to support the Cell architecture, we first show how to automatically generate high-performance discrete Fourier transform kernels that run on a single SPE. The performance of our kernels is comparable to hand tuned code, and reaches 16–20 Gflop/s on a single SPE for input vectors resident in the local memories. We then show how to produce optimized multithreaded code that runs on multiple SPEs.

Key words: fast Fourier transform, Cell, performance, library, code generation

1 Introduction

The Cell BE chip-multiprocessor is designed for high-density floating point computation required in multimedia, visualization, and other similar applications. Its innovative design includes multiple SIMD vector cores (called synergistic processing elements, or SPEs) with explicitly managed per-core local memory and inter-core communication. The Cell's single-precision peak performance is 204.8 Gflop/s using the 8 SPEs alone. However, the same features that allow for high theoretical performance make it difficult and time consuming to design and optimize specific real-world computational kernels for the Cell. Instead of using automated tools, these programs must explicitly address multithreading, SIMD vectorization, and data streaming in order to extract maximum performance.

In this paper we address the automation of program optimization for the Cell: we extend the program generation system Spiral [1] to support the Cell processor. Spiral automates the production, platform adaptation, and optimization of linear transform libraries and targets SIMD vector extensions [2], shared memory multicore CPUs [3], FPGAs [4], and other platforms. It is based on a domain specific, declarative, mathematical language to describe the algorithms, and uses rewriting to parallelize and optimize algorithms at a high level of abstraction.

* This work was supported by NSF through awards 0325687, 0702386, by DARPA (DOI grant NBCH1050009), the ARO grant W911NF0710416, and by Mercury Computer Systems, Inc.

We extend Spiral in two steps, focusing on the discrete Fourier transform (DFT): first, we extend Spiral's SIMD vector program generation capabilities to support the SIMD instruction set of the Cell SPE. Next, we extend Spiral to support explicit DMA transfers and single-program-multiple-data (SPMD) multithreaded code to generate parallel multi-SPE implementations. The performance of our single-threaded code is comparable to the best available hand tuned code. Further, we obtain up to a 2x speed-up with 4-SPE implementations computing single DFTs for sizes 2^9 to 2^{12} data points, with all data resident in the SPEs' local memories.

Related Work. Spiral has previously addressed the task of generating and optimizing scalar, vector, and parallel code [1–3, 5] for a variety of platforms including single and multicore Intel and AMD processors, GPUs, and FPGAs. However, features in the Cell including explicit DMA operations and the generation of large DMA packet sizes have not been addressed. Several other projects have implemented specialized DFT libraries tuned (by hand) for the Cell [6–9]. Cico et al. [7] achieve a performance of about 22 Gflop/s for their code on a single SPE for DFTs of input sizes 2^{10} and 2^{13} resident in the SPE's local memory. The remaining citations in this section assume input and output vectors resident in main memory, and use all 8 SPEs in parallel. Bader et al. [6] develop parallelized DFT algorithms for input sizes between 2^{10}–2^{14}, and achieve between 9 and 23 Gflop/s. Chow et al. [8] achieve 46.8 Gflop/s for a DFT with 2^{24} input samples. Cico et al. [9] implemented a 2^{16} DFT on the Cell that uses double buffering techniques to hide inter-SPE communication costs. FFTW [10] uses an adaptive DFT library, and achieves 18-23 Mflop/s for transforms of input sizes 2^{16}–2^{32}.

Organization. Section 2 provides an overview of Spiral and existing approaches for generating vectorized and parallelized DFT libraries. Section 3 presents our approach to generate high performance multithreaded DFT libraries for the Cell. We present our results in Section 4 and conclude in Section 5.

2 Spiral

Spiral is a program generator for linear transforms including the discrete Fourier transform (DFT), the Walsh-Hadamard transform (WHT), discrete cosine/sine transforms, filters, and others. For a given transform (e.g., a DFT of size 384), Spiral autonomously generates various algorithms, represented in a declarative form as mathematical formulas, and their implementations to find the best match for the target platform [1].

Fig. 1 shows the design of the Spiral system. Spiral uses *breakdown rules* to break down larger transforms into smaller kernels based on recursion. A large space of algorithms (formulas) for a single transform may be obtained using these breakdown rules. A formula thus obtained is structurally optimized to match the architecture using a rewriting system. The formula output is then translated into C code, possibly including intrinsics (for vectorized code) [2] and threading instructions (for multicores) [3]. The performance of this implementation is measured and used in a feedback loop to search over algorithmic alternatives for the fastest one.

SPL and Formula representation. A linear transform in Spiral is represented by a transform matrix M, where performing the matrix-vector multiplication $y = Mx$ trans-

forms the input vector x into the output vector y. Algorithms for transforms can be viewed as structured factorizations of the transform matrices. Such structures are expressed in Spiral using its own signal processing language (SPL), which is an extension of the Kronecker product formalism [11]. The Kronecker product $\otimes$ is defined as:

$$A \otimes B = [a_{k\ell}B],\ A = [a_{k\ell}].$$

Based on this, the well known Cooley-Tukey FFT algorithm's corresponding breakdown rule in Spiral is:

$$\mathrm{DFT}_{mn} \rightarrow (\mathrm{DFT}_n \otimes I_m) D_{n,m} (I_n \otimes \mathrm{DFT}_m) L^{nm}_n \tag{1}$$

where I_n is the $n \times n$ identity matrix, $D_{m,n}$ the diagonal matrix of twiddle factors (see [11] for details), and L^{nm}_m the stride permutation matrix which transposes an $n \times m$ matrix stored in row-major order.

Mapping formulas to target architectures. The key observation is that the tensor product representation of the transform algorithms in Spiral can be mapped to components of the target architecture. The tensor product can be viewed as a program loop. A loop featuring the appropriate structure can be implemented using multiple threads or SIMD vector instructions. For instance, in (1) the construct $I_n \otimes \mathrm{DFT}_m$ is an inherently parallel n-way loop, while the construct $\mathrm{DFT}_m \otimes I_n$ is easily translated into a SIMD vector loop. We use formula rewriting to manipulate vector loops into parallel loops and vice versa, when mapping a formula fragment to the multicore and SIMD parallelism found in CPU architectures [2, 3]. The question addressed by this paper is the mapping of (1) to the Cell architecture. We next introduce a related language called Σ-SPL, which our approach is based on.

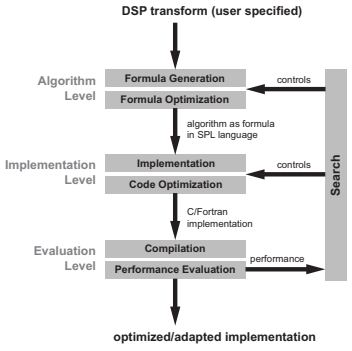

Fig. 1. Spiral's program generation system.

Σ-SPL. Σ-SPL [12] is a layer of abstraction below SPL, and was originally conceived to allow automatic loop merging, but is also useful for capturing and representing other useful information about a given DFT algorithm. Since the main approach in this paper uses Σ-SPL to represent and manipulate DFT algorithms, we provide a brief introduction to Σ-SPL here. A detailed description of Σ-SPL can be found in [12].

Σ-SPL is a matrix based representation that uses sums of products of scatter, gather, and kernel matrices. We first provide an intuitive introduction, followed by a formal definition. As an illustrating example, consider the transform $I_2 \otimes A$ for an arbitrary 2×2 matrix A, that operates on a vector of length 4. This construct can be written as:

$$I_2 \otimes A = \begin{bmatrix} A & \\ & A \end{bmatrix} = \begin{bmatrix} 1 & \cdot & \cdot \\ \cdot & 1 & \cdot \end{bmatrix} A \begin{bmatrix} 1 & \cdot \\ \cdot & 1 \\ \cdot & \cdot \\ \cdot & \cdot \end{bmatrix} + \begin{bmatrix} \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 \end{bmatrix} A \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \\ 1 & \cdot \\ \cdot & 1 \end{bmatrix} = \sum_{i=0}^{1} \mathrm{S}_i \, A_m \, \mathrm{G}_i, \quad (2)$$

where the dots represent zero entries. In each of the summands, the two vertically long matrices, called the gather matrices, select the elements of the input vector that A_m works on, and the horizontally long matrices, called the scatter matrices, select the part of the output vector the summand writes to, and can thus be parameterized as shown.

More formally, Σ-SPL as used in this paper contains matrices parameterized by functions. Functions in Σ-SPL map integer intervals to integer intervals. A function f with domain $\mathbb{I}_n = \{0, \ldots, n-1\}$ and range $\mathbb{I}_N = \{0, \ldots, N-1\}$ is written as $f^{n \to N} : i \mapsto f(i), \quad i \in \mathbb{I}_n, f(i) \in \mathbb{I}_N$. For convenience, we omit the domain and range where it is clear from the context. We now introduce the stride function used in this paper:

$$h_{b,s}^{n \to N} : \; i \mapsto b + is.$$

Σ-SPL as used in this paper contains two types of matrices parameterized by functions, the gather matrix and the scatter matrix: $\mathrm{G}\,(f^{n \to N}),$ and $\mathrm{S}\,(f^{n \to N})$. These are defined thus: Let $e_k^n \in \mathbb{C}^{n \times 1}$ be the column basis vector with the 1 in k-th position and 0 elsewhere. The gather matrix for the index mapping $f^{n \to N}$ is

$$\mathrm{G}(f^{n \to N}) := \left[e_{f(0)}^N \mid e_{f(1)}^N \mid \cdots \mid e_{f(n-1)}^N \right]^\top.$$

Gather matrices thus gather n elements from an array of N elements, and can gather at a stride. Scatter matrices are simply transposed gather matrices: $\mathrm{S}(f^{n \to N}) = \mathrm{G}(f)^\top$. Finally, the summation in Σ-SPL is mathematically a regular matrix sum, but does not incur operations since by design, each of the summands produce a unique part of the output vector. Based on our formal definitions, our previous example (2) is expressed in Σ-SPL as $\sum_{i=0}^{1} \mathrm{S}(h_{2i,1}) A_m \, \mathrm{G}(h_{2i,1})$.

SPL	Σ-SPL
$(I_n \otimes A_m)$	$\sum_{j=0}^{n-1} \mathrm{S}(h_{jm,1}) A_m \, \mathrm{G}(h_{jm,1})$
$(A_m \otimes I_n)$	$\sum_{j=0}^{n-1} \mathrm{S}(h_{j,n}) A_m \, \mathrm{G}(h_{j,n})$
$(I_n \otimes A_m) L_n^{nm}$	$\sum_{j=0}^{n-1} \mathrm{S}(h_{jm,1}) A_m \, \mathrm{G}(h_{j,n})$

Table 1. SPL to Σ-SPL translation.

Table 1 contains a list of SPL expressions used in this paper and their corresponding Σ-SPL representations. From the table, the Cooley-Tukey algorithm in (1) can be written in Σ-SPL as:

$$\mathrm{DFT}_{mn} \to \sum_{j=0}^{m-1} \mathrm{S}(h_{j,m}) \, \mathrm{DFT}_n \, \mathrm{G}(h_{j,m}) \;\; \sum_{j=0}^{n-1} \mathrm{S}(h_{jm,1}) D_j \, \mathrm{DFT}_m \, \mathrm{G}(h_{j,n}). \quad (3)$$

The algorithm developed for the Cell in this paper is based on manipulating (3).

Cell BE. As mentioned earlier, the Cell BE is a heterogeneous multicore processor with a single Power architecture based processing element (PPE) and 8 accelerated vector processing cores called the synergistic processing elements (SPEs). Each SPE has fast access to its own on-chip memory unit (256k) known as the local store.

In this paper, we only use the SPEs for our implementation. We view the Cell processor as a distributed memory multiprocessor connected by a DMA based bus, with each node consisting of a single SPE and its local store memory. Implementing and optimizing the FFT for the Cell processor presents the following challenges:

- Vector units on the SPE cores require the use of vector intrinsics to achieve high performance, and build on our work in [2].
- Multiple SPEs require that the algorithm must be parallelized and load balanced.
- Explicit DMA means that we should identify and issue DMA operations as required.
- DMA performance constraints require the use of large DMA packet sizes to effectively use the available DMA bandwidth.
- DMA costs can be hidden by multi-buffering techniques; this is not addressed in this paper.

3 Generating Fast Implementations for the Cell

Our goal is to generate high performance vectorized, multithreaded Cell code that computes a single 1-D DFT in parallel across multiple SPEs. We do this by designing DFT algorithms that take advantage of the architectural features in the Cell that are designed to speed up numerical computation.

Vectorization. Our first goal was to create a high performance vectorized FFT implementation that executes on a single SPE. Vectorization is based on our previous work [2]. The main idea is briefly described here. Consider the construct $I_n \otimes \mathrm{DFT}_m$ in (1), which cannot immediately be translated into an efficient SIMD vector program. However, it can be rewritten into $L^{mn}_n\big((\mathrm{DFT}_m \otimes I_{n/\nu}) \otimes I_\nu\big)L^{mn}_m$, which is a perfectly vectorized construct for vector size ν, except for the permutations which are handled subsequently by other rewriting rules. The work in [2] was targeted towards Intel's SSE and its variants. Vectorizing for the Cell consisted of porting this to the Cell by adapting it to the SPE's vector instruction set, which was straightforward.

Parallelization. The main problem addressed in this paper is parallelizing the FFT for the Cell architecture. There are two main challenges to address. First, we need a load balanced FFT algorithm that will execute on multiple SPEs in parallel. Second, our algorithm must identify the points at which explicit DMA is required to perform inter-core data exchanges. Also, to obtain high DMA performance on the Cell, DMA packets of large sizes must be used. We use formula rewriting to attack these problems at the algorithmic level, as described in the rest of this section. Our approach uses Σ-SPL to

express algorithms because we use several constructs to represent various types of I/O that cannot be conveniently expressed at the SPL level.

Parallelization through formula manipulation. Our approach is based on rewriting the Σ-SPL version of the Cooley-Tukey algorithm to map well to the Cell's architecture. Here, we briefly describe how rewriting is done, and provide a table of the rewriting identities that we use in this paper. As an illustrating example, consider the Σ-SPL expression:

$$\sum_{j=0}^{n-1} \mathrm{S}(h_{2j,1})\,\mathrm{DFT}_2\,\mathrm{G}(h_{2j,1}). \tag{4}$$

This expresses a loop with n independent iterations, and applies the DFT_2 kernel n times to chunks of the input vector. This expression thus also expresses an n-way parallel loop. We derive an expression for mapping this expression onto p processors by converting the original loop into a nested loop pair so that (4) becomes:

$$\sum_{k=0}^{p-1} \mathrm{S}(h_{2k(n/p),1}) \left(\sum_{j=0}^{n/p-1} \mathrm{S}(h_{2j,1})\,\mathrm{DFT}_2\,\mathrm{G}(h_{2j,1}) \right) \mathrm{G}(h_{2k(n/p),1}).$$

The outer sum in the preceding expression is our parallel loop.

To produce code that includes DMA operations, we introduce two new Σ-SPL constructs, the DMA-scatter $\mathcal{S}$, and the DMA-gather $\mathcal{G}$. These are similar to the regular scatters and gathers, except that they are parameterized by a different function q that better captures DMA operations on blocks of data:

$$q_{b,s,\mu}^{n \to N} \;:\; i \mapsto (b + \lfloor i/\mu \rfloor\, s)\mu + (i \bmod \mu).$$

DMA-scatters and gathers based on q always operate on chunks of size μ elements as opposed to single elements. $\mathcal{G}(q_{i,s,\mu})$ is thus a DMA-gather operation with stride s and DMA block size μ. Ultimately, the DMA-scatter and the DMA-gather are translated by Spiral into DMA get/put code.

Table 2 lists rewriting identities to rewrite the Cooley-Tukey FFT algorithm to a version that can be mapped onto the Cell.

Expression	Parallelized Form
$\sum_{j=0}^{n-1} \mathrm{S}(h_{jm,1}) A_m\,\mathrm{G}(h_{jm,1})$	$\sum_{k=0}^{p-1} \mathcal{S}(q_{km,1,\mu}) (\sum_{j=0}^{\mu-1} \mathrm{S}(h_{jm,1}) A_m\,\mathrm{G}(h_{jm,1}))\,\mathcal{G}(q_{km,1,\mu})$
$\sum_{j=0}^{n-1} \mathrm{S}(h_{j,n}) A_m\,\mathrm{G}(h_{j,n})$	$\sum_{k=0}^{p-1} \mathcal{S}(q_{k,n/\mu,\mu}) (\sum_{j=0}^{\mu-1} \mathrm{S}(h_{j,\mu}) A_m\,\mathrm{G}(h_{j,\mu}))\,\mathcal{G}(q_{k,n/\mu,\mu})$
$\sum_{j=0}^{n-1} \mathrm{S}(h_{jm,1}) A_m\,\mathrm{G}(h_{j,n})$	$\sum_{k=0}^{p-1} \mathcal{S}(q_{km,1,\mu}) (\sum_{j=0}^{\mu-1} \mathrm{S}(h_{jm,1}) A_m\,\mathrm{G}(h_{j,\mu}))\,\mathcal{G}(q_{k,n/\mu,\mu})$

Table 2. Σ-SPL Parallelization identities. The formulas in the second column are parallelized for p processors and a DMA packet size of $\mu = n/p$.

Parallelized FFT Algorithm. We manipulate the Cooley-Tukey FFT algorithm (3) to adapt it for the Cell, based on the rules in Table 2. Our modified version of this algorithm is shown below:

$$\mathrm{DFT}_{mn} \rightarrow \sum_{k=0}^{p-1} \mathcal{S}(q_{k,m/\mu,\mu}) \left(\sum_{j=0}^{m/p-1} \mathrm{S}(h_{j,\mu})\, \mathrm{DFT}_n\, \mathrm{G}(h_{j,\mu}) \right) \mathcal{G}(q_{k,m/\mu,\mu})$$
$$\sum_{k=0}^{p-1} \mathcal{S}(q_{km,1,\mu}) \left(\sum_{j=0}^{n/p-1} \mathrm{S}(h_{jm,1}) D_j\, \mathrm{DFT}_m\, \mathrm{G}(h_{j,\mu}) \right) \mathcal{G}(q_{k,n/\mu,\mu}). \quad (5)$$

(5) is obtained by rewriting the factors in (3) based on two parameters: p, the number of SPEs the algorithm will be executed on, and μ, the DMA packet size to be used for inter-core data exchanges. We now explain how our parallelized algorithm addresses the issues of obtaining load balanced parallelism and DMA operations.

Load balanced parallelism. Each factor in (5) is composed of a pair of nested sums. The outer sums are comprised of p completely independent iterations, and are thus p-way load balanced parallel sums that can be executed in parallel on p SPEs.

Inter-core communications using DMA operations. Note that three of the four DMA-scatters and DMA-gathers in (5) involve reading or writing at strides, which assumes a view of memory that is shared and global. However, since we view the Cell as a distributed memory processor for our purposes, we must explicitly ship data from SPE to SPE when required. From (5), we see that three such inter-SPE communication stages are required: one each at the beginning and at the end, and one in between the two factors. We first show how we reduce the number of communication stages by using a suitable data format, and then describe how we actually handle the remaining stage.

Block cyclic data format. Our multithreaded FFT assumes block cyclic input and output data formats: input and output vectors are assigned to processors in a round-robin fashion in blocks of size μ. Assuming a block cyclic format is both beneficial and practical. The assumption reduces the number of communication stages by cancelling out the initial and final DMA-gather/scatter stages, so that (5) becomes:

$$\mathrm{DFT}_{mn} \rightarrow \sum_{k=0}^{p-1} \mathcal{S}(q_{k,1,\mu}) \left(\sum_{j=0}^{m/p-1} \mathrm{S}(h_{j,\mu})\, \mathrm{DFT}_n\, \mathrm{G}(h_{j,\mu}) \right) \mathcal{G}(q_{k,m/\mu,\mu})$$
$$\sum_{k=0}^{p-1} \mathcal{S}(q_{km,1,\mu}) \left(\sum_{j=0}^{n/p-1} \mathrm{S}(h_{jm,1}) D_j\, \mathrm{DFT}_m\, \mathrm{G}(h_{j,\mu}) \right) \mathcal{G}(q_{k,1,\mu}). \quad (6)$$

We now have only a single communication stage (the strided DMA-gather) remaining between the two factors. This results in significant savings since each communication stage involves the cost of data transfer (depends on kernel size) and the cost of the required succeeding global synchronization (fixed at approximately 660 processor cycles). This allows us to obtain a parallel speedup for even the smaller DFT kernels. The

format is practical because DFTs are typically a single step in a series of operations on input data. The cost of conversion to the block cyclic format can be hidden or amortized using preceding and succeeding operations.

Implementing inter-SPE communication. To implement the single remaining communication stage in (6), consider the scatter-gather pair formed by the inner DMA-scatter and the inner DMA-gather. The scatter has a stride parameter of 1, which means that all its writes are to the SPE's local memory. The gather, however, has a non-unit stride, and thus reads from an assumed global memory (in other words, it reads elements that are resident in the local stores of the other SPEs). In this case, the produced code must translate the scatter (which operates on local memory space) into a null operation, and translate the gather into DMA-get operations. To achieve this, we build and use an address table to allow each scatter to know the ultimate destination of each data packet through the entire scatter-gather pair. This approach allows us to produce DMA instructions for scatters, and renders gathers into null operations, and adds a global synchronization barrier between the pair. Our actual implementation in Spiral, not described here due to lack of space, is general enough to be able to identify and nullify communications stages when possible, on other algorithms and signal processing transforms.

DMA performance. DMA requests that use larger packet sizes achieve higher interconnect bandwidth performance on the Cell [13]. Our algorithm is therefore tuned to produce DMA packets of the largest sizes, by implicitly choosing the maximum value for the μ parameter in rewriting rules shown in Table 1. One better approach to increasing packet sizes is to assemble and send only one packet from any given SPE to another during each communication stage. We are currently studying the tradeoff between the cost of creating these larger packets and the decreased communication costs.

Σ-SPL to code. To generate code for (6), we observe that the expression consists of two computation stages that are both p-way parallel. We thus generate an SPMD type program, parameterized only by the variable p, to be executed across p SPEs in parallel. Since the rightmost factor produces data that the leftmost factor is dependent upon, a synchronization barrier must be inserted between these stages after the all-to-all data exchange is completed. An annotated skeleton of the code produced by (6) follows:

```
void DFT1024(float  *Y, float  *X, int spe_id) // SPMD function
{   for(j=0; j<n/p; j++)               // Right-most sum
    { // declarations ...
        s0    = X[j]; s1 = X[j+1];     // Read from input vector
        T1[j] = spu_add(s0, s1);       // Perform DFT_m kernel
        ...
        // S(q_km,1,u) DMA puts to temporary vectors: (Scatter+subsequent gather):
        spu_mfcdma64(T1, addr_table(T2, j), ... , MFC_PUT_CMD);
    }
    all_to_all_sync_barrier();
    for(j=0; j<m/p; j++)               // Left-most sum
    {   // declarations ...
        // G(q_k,m/u,u) DMA gets: null ops (accounted for in dma puts above)
        s10  = T2[j]; s11 = T2[mj+1];  // Read from temporary vector
        s12  = spu_add(s10, s11);      // Perform DFT_n kernel
        ...
        Y[j] = s12 ...                 // Write to output vector
    }
}
```

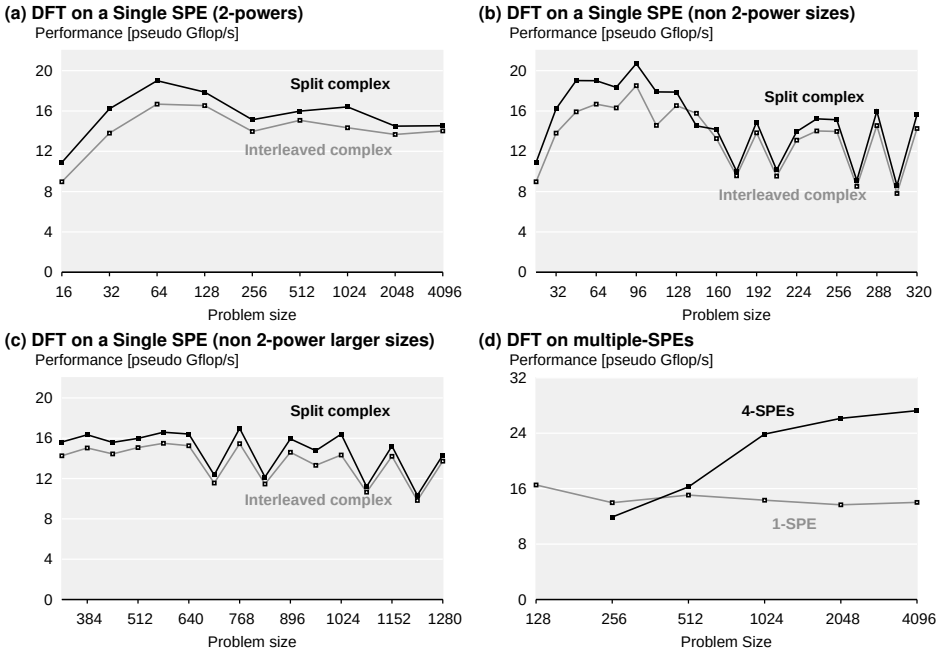

Fig. 2. Performance results for single precision 1-D complex DFTs on a PlayStation 3 (3.2 GHz Cell). Higher is better.

Automatic rewriting using Spiral. The entire process described above is automated with Spiral. For a given size, Spiral automatically rewrites the DFT into the parallelized version for user specified values of p with the maximum possible value for μ. The smaller DFTs in the parallelized formulas are automatically vectorized, and Spiral searches over a space of possible recursive breakdowns to find the fastest algorithm that is blocked appropriately for the Cell's register space. This makes it easy to generate DFT code for a given number of SPEs for a large range of DFT sizes.

4 Experimental Results

We evaluated our generated single-precision 1-D DFT implementations on a single SPE and on 4 SPEs of the Cell processor in a PlayStation 3 (6 available SPEs) running at 3.2 GHz. Our idea in this paper is a first attempt at formalizing the mapping of the DFT onto the SPEs, and does not yet target double buffering techniques. Furthermore, DFTs are usually a single stage in a series of operations on input data. Hence, we assume the input and output vectors are resident in the SPEs' local memories.

Fig. 2(a) displays the single-core performance of our generated DFT kernels for 2-power sizes for both the split-complex and the interleaved-complex data formats. Performance is computed in pseudo Gflop/s using $5n\log_2(n)/(\text{runtime(s)}\cdot 10^9)$. Figs. 2(b) and 2(c) show the performance for DFT kernels for input sizes that are multiples of 16 or 32. Spiral generated code achieves 16–20 Gflop/s which is is comparable to the best reported single-core performance for 2-power sizes in [7].

In Fig. 2(d) we compare our generated multithreaded DFT code (assuming a block cyclic data distribution) executed on 4 SPEs to the single-SPE kernel performance. The largest DMA packet sizes possible are used for all our DFT kernels. For $n = 512$ we reach the break-even point (the runtime is about 5,000 cycles), and for $n = 4,096$ we see close to a 2x speed-up, leading to 27 Gflop/s for a single, non-streamed, 1-D DFT. Our multithreaded DFT code cannot be easily compared with other existing libraries since we measure performance for input and output vectors resident in the local stores as opposed to main memory.

5 Conclusion

We presented preliminary work that extends the Spiral framework to automatically generate DFT code for the Cell BE. Our DFT libraries are performance competitive with hand-tuned code on a single SPE, and speed up small DFT sizes by using multiple SPEs. We are currently examining further algorithmic manipulations to increase the achieved communication bandwidth, and to enable double-buffering to hide communication costs.

References

1. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" **93**(2) (2005) 232– 275
2. Franchetti, F., Püschel, M.: A SIMD vectorizing compiler for digital signal processing algorithms. In: Intl. Parallel and Distributed Processing Symposium (IPDPS). (2002) 20–26
3. Franchetti, F., Voronenko, Y., Püschel, M.: A rewriting system for the vectorization of signal transforms. In: High Performance Computing for Computational Science (VECPAR). Volume 4395 of Lecture Notes in Computer Science., Springer (2006) 363–377
4. Milder, P.A., Franchetti, F., Hoe, J.C., Püschel, M.: Formal datapath representation and manipulation for implementing DSP transforms. In: Design Automation Conference. (2008)
5. Bonelli, A., Franchetti, F., Lorenz, J., Püschel, M., Ueberhuber, C.W.: Automatic performance optimization of the discrete Fourier transform on distributed memory computers. In: International Symposium on Parallel and Distributed Processing and Application (ISPA). Volume 4330 of Lecture Notes In Computer Science., Springer (2006) 818–832
6. Bader, D.A., Agarwal, V.: FFTC: Fastest Fourier transform for the IBM Cell Broadband Engine. In: IEEE Intl. Conference on High Performance Computing. (2007) 172–184
7. Cico, L., Cooper, R., Greene, J.: Performance and Programmability of the IBM/Sony/-Toshiba Cell Broadband Engine Processor. In: Proc. of (EDGE) Workshop. (2006)
8. Chow, A.C., Fossum, G.C., Brokenshire, D.A.: A programming example: Large FFT on the Cell Broadband Engine. Technical report (May 2005)
9. Greene, J., Cooper, R.: A parallel 64K complex FFT algorithm for the ibm/sony/toshiba cell broadband engine processor. In: Global Signal Processing Expo (GSPx). (2005)
10. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" **93**(2) (2005) 216–231
11. Van Loan, C.: Computational Framework of the Fast Fourier Transform. SIAM (1992)
12. Franchetti, F., Voronenko, Y., Püschel, M.: Loop merging for signal transforms. In: Proc. Programming Language Design and Implementation (PLDI). (2005) 315–326
13. Kistler, M., Perrone, M., Petrini, F.: Cell multiprocessor communication network: Built for speed. IEEE Micro **26**(3) (2006) 10–23