Computer Generation of Fast Fourier Transforms for the Cell Broadband Engine

Srinivas Chellappa
schellap@ece.cmu.edu

Franz Franchetti
franzf@ece.cmu.edu

Markus Püschel
pueschel@ece.cmu.edu

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213, USA

ABSTRACT

The Cell BE is a multicore processor with eight vector accelerators (called SPEs) that implement explicit cache management through direct memory access engines. While the Cell has an impressive floating point peak performance, programming and optimizing for it is difficult as it requires explicit memory management, multithreading, streaming, and vectorization. We address this problem for the discrete Fourier transform (DFT) by extending Spiral, a program generation system, to automatically generate highly optimized implementations for the Cell. The extensions include multi-SPE parallelization and explicit memory streaming, both performed at a high abstraction level using rewriting systems operating on Spiral's internal domain-specific language. Further, we support latency and throughput optimizations, single and double precision, and different data formats. The performance of Spiral's computer generated code is comparable with and sometimes better than existing DFT implementations, where available.

Categories and Subject Descriptors

I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*algebraic algorithms*; D.3.4 [**Programming Languages**]: Processors—*compilers, optimization, code generation*; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*computation of transforms*

General Terms

Algorithms, performance, design, theory

Keywords

Fast Fourier transform, DFT, automatic performance tuning, Cell BE, program generation, performance library, multicore, parallelization, streaming, multibuffering

1. INTRODUCTION

The Cell Broadband Engine (Cell BE, or simply, Cell) is a multicore processor that is designed for high-density floating-point computation required in multimedia, visualization, and other science and engineering applications. As shown in Fig. 1, its design includes 8 SIMD vector cores (called synergistic processing elements, or SPEs) that provide a single-precision peak performance of 204.8 Gflop/s (using only the SPEs). Memory bandwidths issues are addressed by providing each core with its own fast on-chip local memory (local store). This, however, requires that the programmer performs all memory and inter-core data movement operations explicitly. One can view the local stores as private, per-SPE, software-managed caches. While this allows for finer control of the memory system as compared to a conventional cache-based processor, it drastically increases the programming burden and expertise required by software developers.

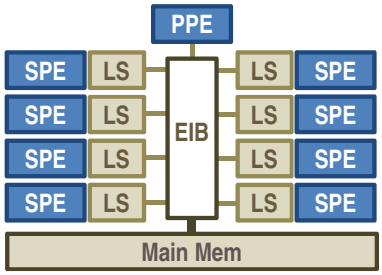

Figure 1: Cell processor.

Additionally, application demands come into play when considering a high-performance implementation of a computational kernel like the discrete Fourier transform (DFT) for the Cell. The DFT is the focus of this paper, and is arguably among the most important numerical algorithms used in science and engineering. An application might require either a *latency-optimized* version to compute a single DFT that is in the critical path of operations to be performed on the input data, or might require a *throughput-optimized* version to compute a large number of DFTs on a stream of input data. In contrast to multicore processors with hardware-controlled caches, libraries for the Cell must take into account such distinctions to provide maximum performance. Using a suboptimal variant may result in a substantial performance degradation.

The combination of such requirements and other specifications such as the size or data format of the DFT and the number of cores allocated gives rise to a plethora of possible usage scenarios as depicted in Fig. 2. A small latency-optimized DFT kernel can work on vectors resident on-chip (local store) on a single SPE as shown (a) Medium sizes DFTs must be parallelized, and can work on vectors resident on-chip or off chip as shown in (c) and (d). Large DFTs must be streamed in and out of the chip and computed in parts as

ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.

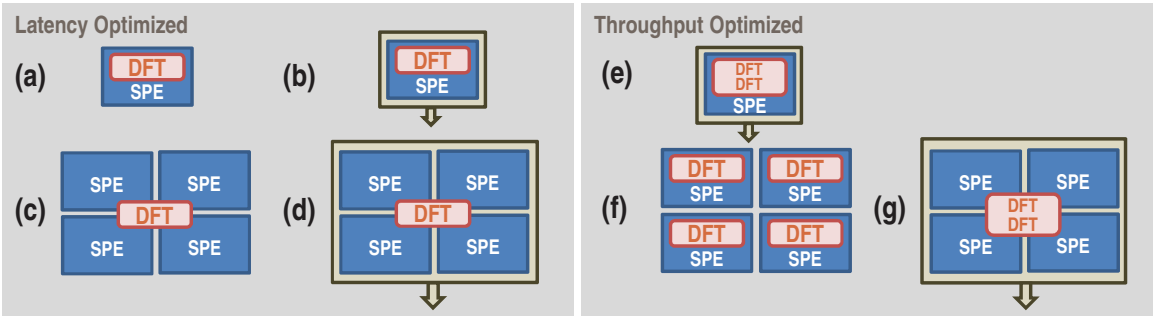

Figure 2: Usage scenarios: (a)–(d) have to be optimized for latency (notice that only a single DFT kernel is performed in these cases); (e)–(g) have to be optimized for throughput (multiple DFTs are performed). The box with the arrow around the SPEs denotes operations on vectors resident in main memory.

shown in (b). Throughput-optimized versions can use streaming to hide memory access costs, see (e), or execute small DFTs independently on multiple SPEs as shown in (f), or stream larger DFTs parallelized across multiple SPEs as shown in (g).

This large space of possible implementations requires many non-trivial trade-offs, many of which can only be resolved by actually implementing the candidates and picking the best-performing ones, leading to an enormous coding effort. Thus, for complex architectures such as the Cell, tool support is necessary to relieve the human programmer's burden.

Contribution. In this paper, we address the problem of automatically generating high-performance 1D and 2D DFT libraries for the Cell, supporting the above-mentioned usage scenarios. We do so by extending the program generation system Spiral [15] for the Cell processor, building upon and going well beyond the preliminary work in [2]. In particular, we add support for memory streaming (using explicit DMA transfers), extend Spiral's SIMD vector engine [9, 10] to support the Cell SPE vector instruction set, and support parallelizing across multiple SPEs. We also support the split and interleaved complex data formats, regular and block cyclic data distribution formats, and add support for throughput optimization of small and medium sized DFTs. We expose the choices summarized in Figure 2 to the user of Spiral and leave it to the system to find the appropriate implementation for a given usage scenario.

Limitations. In contrast to other approaches, our system automates the generation of fast source code directly from an input specification, but still has a couple of limitations. Firstly, our code generation system currently targets only on-chip problem sizes up to 16k (where input and output data fit completely into all the local stores used). Supporting larger off-chip problem sizes for both 1D and 2D transforms using multiple SPEs is ongoing research, (though our system already supports off-chip sizes, limited to using single SPEs for 2D DFTs, as shown in Section 4). Due to this limitation, we are unable to compare our performance with other libraries for larger problem sizes. Secondly, our system generates code only for fixed problem sizes. While this simplifies usage and application integration, code generated using our system for various problem sizes must be bundled for applications that demand a range of problem sizes. However, our approach should be extensible to general-size library generation as developed in [17].

Related work. Spiral is a program generation system that has previously addressed the task of generating and optimizing scalar, FMA, vector, and parallel code for linear transforms for a variety of platforms including single and multicore Intel, AMD and PowerPC processors, GPUs, and FPGAs [15, 9, 10]. Our work is built on [2], which introduced preliminary work on the generation of multithreaded code for the Cell using Spiral, for vectors resident in the local stores, distributed in a block cyclic format.

Several projects have developed hand-tuned DFT library implementations for the Cell. Cico et al. [5] achieve about 22 Gflop/s on a single SPE for DFTs of input sizes 2^{10} and 2^{13} resident in the SPE's local store. IBM's SDK for the Cell includes an FFT library [13]. Chow's single-SPE FFT library generator achieves 12–15 Gflop/s for a small range of 2-power sizes [3]. The remaining citations assume input and output vectors are resident in main memory, and use all 8 SPEs. Bader et al. [1] achieve 9–23 Gflop/s for parallelized DFTs of input sizes 2^{10}–2^{14}. Cico et al.'s 2^{16}-sized DFT is estimated to achieve a throughput of about 90 Gflop/s [12]. FFTW [11], a latency-optimized adaptive DFT library, achieves 18–23 Gflop/s for input sizes 2^{16}–2^{32}. Chow et al. [4] achieve 46.8 Gflop/s for a 2^{24}-sized DFT. Each of these projects are limited in the range of DFT sizes, input/output data formats, and memory streaming options.

Paper organization. In Section 2, we provide background on the Spiral code generation system with a focus towards the parts relevant for this paper. In Section 3, we first present our formal approach for parallelization and streaming. We then present strategies for combining parallelization and streaming to obtain libraries for various usage scenarios. We present our performance results in Section 4 and conclude in Section 5.

2. SPIRAL

Overview. Spiral [15] is a program generator that autonomously generates a high-performance software library for some selected functionality and target platform. Spiral originally focused on linear transforms, and in particular the DFT, however, latest developments expand Spiral beyond this domain to include libraries for coding (Viterbi decoder and EBCOT encoder in JPEG 2000), linear algebra (matrix-matrix-multiplication), and image processing [7]. While Spiral still only supports restricted domains, the generated code is very fast and often rivals the performance of expertly hand-tuned implementations. We limit the further discussion on Spiral to the DFT.

At the heart of Spiral is a declarative representation of algorithms, symbolic computation (rewriting systems), and high-level architecture models. Spiral represents algorithms and hardware models in the same internal language (signal processing language, SPL), which a extension of the Kronecker product formalism of multilinear algebra. An intelligent search mechanism accounts for hardware details that are difficult to model. Algorithms and program transformations are encoded as rewriting rules that operate on SPL formulas to produce code.

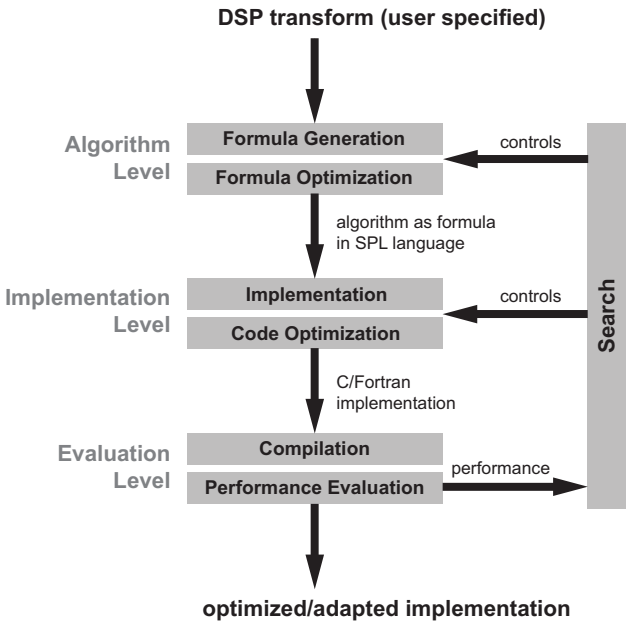

Figure 3: Spiral's program generation system.

System structure. Spiral's structure is shown in Figure 3. The input to Spiral is a problem specification containing the requested functionality and some target hardware properties, for instance "generate a 8,192-point DFT using 4 Cell SPEs," appropriately encoded in Spiral's internal language. 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) [9] and threading instructions (for multicore parallelism) [10]. The performance of this implementation is measured and used in a feedback loop to search over algorithmic alternatives for the fastest one.

We now discuss the internal representation of DFT algorithms, hardware architecture, and relevant program transformations in further detail.

SPL and formula representation. A linear transform in Spiral is represented by a transform matrix M, where performing the matrix-vector multiplication $y = Mx$ transforms 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 [16]. The Kronecker product $\otimes$ is defined as:

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

For example,

$$I_n \otimes A = \begin{bmatrix} A & & & \\ & A & & \\ & & \ddots & \\ & & & A \end{bmatrix}$$

is block-diagonal, and can be visualized as shown.

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

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

SPL Construct	Output stride	Input stride
$y = (I_n \otimes A_m)x$	1	1
$y = (A_m \otimes I_n)x$	n	n
$y = (I_n \otimes A_m)L^{mn}_n x$	1	n
$y = L^{mn}_m (I_n \otimes A_m)x$	n	1

Table 1: Basic SPL constructs. Strides refer to inputs and outputs of each of the A_m matrices.

where I_n is the $n \times n$ identity matrix, $D_{m,n}$ the diagonal matrix of twiddle factors (see [16] for details), and L^{nm}_m the stride permutation matrix which transposes an $n \times m$ matrix stored in row-major order.

The two-dimensional $\mathrm{DFT}_{m \times n}$ is defined as the tensor product of two 1D DFTs, and the row-column algorithm is given by the factorization of the tensor product into two tensor products with identity matrices,

$$\mathrm{DFT}_{m \times n} \rightarrow \mathrm{DFT}_m \otimes \mathrm{DFT}_n \rightarrow (\mathrm{DFT}_m \otimes I_n)(I_m \otimes \mathrm{DFT}_n). \qquad (2)$$

Performing multiple independent DFTs (which allows for throughput optimization) is simply expressed by implementing

$$I_n \otimes \mathrm{DFT}_m\,. \qquad (3)$$

Basic constructs. The transforms considered in this paper are built from a set of basic SPL constructs which differ only in the stride at which their inputs and outputs are accessed, as shown in Table 1. The strides determine which set of input and output points are connected to the inputs and outputs of each of the n A_m matrices in the constructs. The transform algorithms (1)–(3) use only the first three constructs shown in Table 1 (the diagonal matrix $D_{m,n}$ in (1) does not pose any additional problems and is handled with the methods described in [8]). Since the fundamental difference between these constructs is simply the presence or absence of a strided access pattern, we henceforth focus our discussions on how to implement $I_m \otimes A_n$ and $A_m \otimes I_n$ on the Cell efficiently under the scenarios summarized in Figure 2. Additionally, we study how to implement a composition of any two constructs, $A \cdot B$. This allows us to build implementations for 1D and 2D DFTs using (1)–(3).

Modeling target architectures with formulas. The key observation is that the tensor product representation of the transform algorithms in Spiral can be mapped to components of the target architecture. Using well known formula identities recast as rewriting rules, Spiral manipulates algorithms in SPL to match the target architecture [9, 10]. The issue addressed by this paper is the question of how to extend this idea to the Cell processor, which requires a combination of SIMD vectorization, multithreading, and explicit memory management via DMA.

The tensor product can be viewed as a program loop, where certain important loop properties are made explicit. This allows us to easily derive a high-performance implementation of the loop. As an example, take the construct $I_n \otimes \mathrm{DFT}_m$ in (1), which is an inherently parallel n-way loop with every iteration operating on a private, contiguous piece of the input vector. This construct can be mapped easily to execute in parallel on multiple SPEs. Conversely, the construct $\mathrm{DFT}_m \otimes I_n$ is easily translated into a SIMD vector loop.

Σ-SPL. Certain aspects of optimizing the DFT for the Cell requires the exposure of details that cannot be expressed with SPL formulas. Σ-SPL [8] is a extension of SPL that makes loops, and memory accesses as function of loop variables explicit, while retaining the matrix abstraction which is key to the power of SPL. Σ-SPL can be

seen as a abstraction layer between SPL and actual C code. Σ-SPL adds the iterative sum of matrices and matrices parameterized by index mapping functions (gather and scatter matrices) to SPL. This allows it to concisely describe loop-carried access patterns inside a matrix-based algorithm representation, which in turn, is an excellent level of abstraction to perform parallelization and streaming, as we will later see.

We first provide an intuitive introduction to Σ-SPL, 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:

$$I_2 \otimes A = \begin{bmatrix} A & \\ & A \end{bmatrix}$$
$$= \begin{bmatrix} 1 & . & . & . \\ . & 1 & . & . \end{bmatrix} A \begin{bmatrix} 1 & . \\ . & 1 \\ . & . \\ . & . \end{bmatrix} + \begin{bmatrix} . & . & 1 & . \\ . & . & . & 1 \end{bmatrix} A \begin{bmatrix} . & . \\ . & . \\ 1 & . \\ . & 1 \end{bmatrix}$$
$$= \sum_{i=0}^{1} \mathrm{S}_i \, A_m \, \mathrm{G}_i, \quad (4)$$

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 defines matrices parameterized by index mapping functions, which map integer intervals to integer intervals. A index mapping function f with domain $\{0, \ldots, n-1\}$ and range $\{0, \ldots, N-1\}$ is written as $f^{n \to N} : i \mapsto f(i)$. For convenience, we omit the domain and range where it is clear from the context. We now introduce the two index mapping functions used in this paper,

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

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

which abstracts strided memory access at scalar, and DMA packet granularities, respectively (b, s, and μ correspond to base, stride, and packet size). Index mapping functions are used to parameterize gather and scatter matrices, which encode data movement. 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. Scatter matrices are simply transposed gather matrices, $\mathrm{S}(f^{n \to N}) = \mathrm{G}(f)^\top$.

The iterative matrix sum, which encodes loops, is conventionally defined. However, it 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 (4) is expressed in Σ-SPL using index mapping functions to parameterize gathers and scatters as $\sum_{i=0}^{1} \mathrm{S}(h_{2i,1}) A_m \, \mathrm{G}(h_{2i,1})$.

Code generation. Since Σ-SPL formulas essentially represent loops, converting them into C code is straightforward. Gather matrices read from the input vector, which the kernel then operates on. Scatter matrices then write to the output vector. The matrix sum becomes the actual loop. Table 2 shows how the fundamental Σ-SPL constructs are derived from SPL constructs, and then translated to C code. By applying these rules recursively, any Σ-SPL formula can be translated into C code.

3. PARALLELIZATION AND STREAMING VIA FORMULA MANIPULATION

To automatically generate high-performance code for the Cell we must address three architectural characteristics: 1) within an SPE we need to produce SIMD vector code, 2) executing code across multiple SPEs requires parallelization, and 3) hiding the latency of data transfer between local stores and the main memory requires multibuffering, which is a software pipelining technique. Rather than building a Cell-specific system, we extend Spiral to support these general concepts (which we call paradigms), and then add a Cell-specific implementation layer, so our system can easily be ported to future architectures that implement the same paradigms. For vectorization, we use Spiral's existing SIMD paradigm, as developed in [9].

Program transformations as formula identities. We use formula identities that translate SPL formulas to mathematically equivalent (but structurally different) Σ-SPL formulas. This translation encodes program transformations like tiling, loop splitting, loop interchange, and loop pipelining, allowing us to tailor the generated code to the architectural features of the Cell processor. Using SPL as the starting point enables us to perform these techniques with simple analysis, as SPL encodes the necessary proofs explicitly in the formula. Our target, Σ-SPL, allows us to easily express the results of complicated restructuring.

Rewriting via tagging. The SPL-to-Σ-SPL identities are applied by Spiral's rewriting system. We use a *tagging system* to represent hardware parameters as part of the problem specification, and also, to guide our formula rewriting process. We introduce a tag for parallelization, and one for streaming, into Spiral's rewriting system. We express the request to rewrite a formula construct A_m into its parallel or streamed version respectively, by tagging it,

$$\underbrace{A_m}_{\mathrm{spe}(p,\mu)} \text{ (parallelization), or } \quad \underbrace{A_m}_{\mathrm{stream}(s)} \text{ (streaming).}$$

Tags are parameterized (number of processors p, DMA packet size μ, and s-way multibuffering), and we discuss these parameters and other paradigm-specific details in their respective sections. Note that tags can be nested when needed. We reuse the SIMD vectorization paradigm from [10] by porting it over to the Cell processor, and assume an implicit vectorization tag for all our kernels.

Formula template. The general idea is to rewrite the basic SPL formulas, $I_m \otimes A_n$ and $A_m \otimes I_n$, into nested sums that look like

$$\sum_{k=0}^{p-1} \mathrm{S}(v_k) \left(\sum_{j=0}^{\mu-1} \mathrm{S}(w_j) A_m \, \mathrm{G}(r_j) \right) \mathrm{G}(s_k), \quad (5)$$

that can be implemented efficiently as parallel program running on multiple SPEs, or as a multibuffered loop for streaming memory access. When mapped to a parallel or streaming program, the outer iterative sum in (5), and outer gather and scatter need to be interpreted and implemented with functionality beyond what is shown in Table 2. In the next two sections we discuss the functions v, w, r, and s (parameterized by the loop variables k and j) that make (5) mathematically equivalent to $I_m \otimes A_n$ or $A_m \otimes I_n$. We also show how these allow for the high performance when (5) is implemented on the Cell using multiple SPEs or under the context of memory streaming. The formal equivalence the left-hand side and right-hand side of the derived rewriting rules can be proven using formula identities outlined in [8].

SPL	Σ-SPL	Code
$y = (A_n B_n)x$	$y = (A_n B_n)x$	`t[0:1:n-1] = B(x[0:1:n-1]);` `y[0:1:n-1] = A(t[0:1:n-1];)`

$y = (I_n \otimes A_m)x$	$y = \left(\sum_{j=0}^{n-1} \mathrm{S}(h_{jm,1}) A_m \mathrm{G}(h_{jm,1})\right) x$	`for(i=0;i<n;i++)`
		`  y[i*n:1:i*m+m-1] =`
		`  A(x[i*n:1:i*m+m-1]);`

$y = (A_m \otimes I_n)x$	$y = \left(\sum_{j=0}^{n-1} \mathrm{S}(h_{j,n}) A_m \mathrm{G}(h_{j,n})\right) x$	`for(i=0;i<n;i++)`
		`  y[i:n:i+m-1] =`
		`  A(x[i:n:i+m-1]);`

$y = (I_n \otimes A_m)L_n^{mn}x$	$y = \left(\sum_{j=0}^{n-1} \mathrm{S}(h_{jm,1}) A_m \mathrm{G}(h_{j,n})\right) x$	`for(i=0;i<n;i++)`
		`  y[i*n:1:i*m+m-1] =`
		`  A(x[i:n:i+m-1]);`

Table 2: Translating SPL to Σ-SPL, and then to code. x denotes the input and y the output vector. The subscripts of A and B specify the size of the (square) matrix. We use Matlab-like notation: `x[b:s:e]` denotes the subvector of x starting at `b`, ending at `e` and extracted at stride `s`.

We now discuss parallelization and streaming in isolation. Later, we discuss the additional issues that arise from combining these two paradigms.

3.1 Parallelization

This section explains how to generate parallel multi-SPE code that computes a single DFT with its input and output data distributed across the local stores of the involved SPEs. The input specifications include p, the number of SPEs (if not all SPEs are involved, more than one instance can be run), μ, the required DMA packet size, and the data format (block cyclic or natural). In essence, we view the SPEs as a distributed memory platform in which data exchanges are performed through DMA calls, on which we execute our single-program-multiple-data (SPMD) code. This paper builds on earlier ork that addresses multicore platforms within Spiral [9] and extends the preliminary work presented in [2].

Rewriting rules for parallelization instantiate (5) but contain specially marked gathers, scatters, and iterative sums. We denote a sum to be parallelized such that every iteration k runs on a different SPE by $\sum_{\parallel}$. We introduce a specially marked gather $\overset{\text{DMA}}{G}$ and scatter $\overset{\text{DMA}}{S}$ to abstract the fact that an SPE must use DMA communication to access non-local data, due to the access pattern of the function parameterizing the gather or scatter.

Parallelizing $\mathbf{I_n \otimes A_m}$. We first examine the construct $I_n \otimes A_m$. This construct divides the input vector into n equal chunks, and multiplies each of those chunks with the kernel A_m. It is thus embarrassingly parallel, and completely load balanced for n processors. For a distributed memory platform (Cell), this represents each processor (SPE) performing a computation on vectors starting and ending in its own local memory (LS). Manipulating this construct to work on p processors where $p \leq n$ is accomplished by using the identity $I_n = I_p \otimes I_{n/p}$. This manipulation results in obtaining a formula that matches (5) and is equivalent to loop splitting. The rewriting rule for $I_n \otimes A_m$ is

$$\underbrace{I_n \otimes A_m}_{\text{spe}(p,\mu)} \rightarrow \sum_{k=0}^{p-1}{}_{\parallel} \mathrm{S}(h_{knm/p,1}) \left(\sum_{j=0}^{n/p-1} \mathrm{S}(h_{jm,1}) A_m \mathrm{G}(h_{jm,1})\right) \mathrm{G}(h_{knm/p,1}). \quad (6)$$

The outer scatter and gather in this case read and write with a stride of 1, and thus do not involve DMA instructions since they access only local data.

Parallelizing $\mathbf{A_m \otimes I_n}$. Finding the right functions v, w, r, and s to make $A_m \otimes I_n$ match (5) is more complicated. In this case we need to perform a formula manipulation that is equivalent to loop tiling (splitting a loop and subsequently interchanging it). The construct $A_m \otimes I_n$ also represents applying A_m n times to parts of the input data, however, the input data to a single A_m is now read from and written to at a stride of n, i.e., the data of multiple A_m are interleaved. Consecutive loop iterations touch contiguous data and thus need to be executed on the same SPE. We need to find a formula that is translated into two nested loops, where the outer loop operates across the multiple nodes (SPEs) and the inner loop operates on contiguous data within a single node. The outer loop in this setup needs to gather data from other SPEs, resulting in inter-node communication. On the Cell this communication is translated into inter-core DMA operations. There is a degree of freedom in performing the tiling operation, and we choose to maximize DMA performance by maximizing DMA packet size. A similar approach to the previous case allows us to arrive at the rewrite rule,

$$\underbrace{A_m \otimes I_n}_{\text{spe}(p,\mu)} \rightarrow \sum_{k=0}^{p-1}{}_{\parallel}^{\text{DMA}} \mathrm{S}(q_{k,n/\mu,\mu}) \left(\sum_{j=0}^{\mu-1} \mathrm{S}(h_{j,\mu}) A_m \mathrm{G}(h_{j,\mu})\right) \overset{\text{DMA}}{\mathrm{G}}(q_{k,n/\mu,\mu}), \quad \mu = n/p \quad (7)$$

A synchronization barrier at the end of the computation of this block is necessary because the SPEs send DMA packets to the LSs of the other SPEs at the end of the computation. The barrier guarantees the completion of all these DMAs, allowing each SPE to proceed working on newly arrived data. Our synchronization barrier is implemented using mailbox messages sent between the SPEs.

Parallelizing $\mathbf{A \cdot B}$. The fact that some constructs introduce inter-core communication requires us to investigate the composition $A \cdot B$ of two parallelized Σ-SPL constructs A and B. If at the interface between two parallelized constructs both require communication, these two communication steps can be combined into a single communication step. The first one performs a scatter operation (producer), followed by the second one that immediately performs a gather operation (consumer) on the same data, which hurts perfor-

mance. The most general solution to this is to build an address table for each such scatter to allow it to know the ultimate destination of each data packet through the entire scatter-gather pair. The scatter thus becomes DMA send operations, while the gather is rendered into a null operation. Another reason to perform this optimization is to allow the DMA packets to be sent as soon as they are ready, so communication can occur in the background.

Trade-offs and constraints. Like on most distributed memory machines, achieving performance on the Cell requires large packet sizes for reasonable DMA performance [14]. When applying the rules discussed in this section to (1), we obtain two parallel stages. The choice of m and n in (1) allows us to trade kernel size for DMA packet size to find an overall good mapping of the DFT to multiple SPEs. However, the maximally achievable DMA packet only grows with the square root of the DFT size, and only DFTs up to 2^{15} can fully be fit in the combined storage of 8 SPEs, limiting the achievable performance. By allowing a different data layout—the block-cyclic distribution—we change the index mapping function of the first stage's gather and the last stage's scatter into functions not requiring any communication. With this distribution, we therefore have the option of eliminating two of the three all-to-all communications to obtain significantly higher performance.

3.1.1 Example: SPL to code

We provide an example that illustrates the code generation process starting from a functional description at the SPL level, and ending in production of C code. We generate code for a 1D DFT of size 64, parallelized for 2 SPEs. We start by representing this at the SPL level using tags:

$$\underbrace{\mathrm{DFT}_{64}}_{\mathrm{spe}(2,4)} \rightarrow \underbrace{(\mathrm{DFT}_8 \otimes I_8) D_{8,8}}_{\mathrm{spe}(2,4)} \underbrace{(I_8 \otimes \mathrm{DFT}_8) L^{nm}_8}_{\mathrm{spe}(2,4)}$$

Note that the largest DMA packet size that can be specified for a DFT of size 64 is 4 complex elements, yielding 32 bytes in single precision. Cell DMA operations larger than 16 bytes must be in multiples of 16 bytes, which can be guaranteed using our tagging system. Data involved in DMA transfers must be aligned at 16-byte boundaries, which our rewriting rules implicitly take into account. Using a generalization of (6) and (7), we perform parallelization to obtain:

$$\sum_{k=0}^{p-1}{}_{\parallel}^{\mathrm{DMA}} \mathrm{S}(q_{k,2,4}) \left(\sum_{j=0}^{4-1} \mathrm{S}(h_{j,4})\, \mathrm{DFT}_8\, \mathrm{G}(h_{j,4}) \hat{D} \right) \overset{\mathrm{DMA}}{\mathrm{G}}(q_{k,2,4}) \cdot$$
$$\sum_{k=0}^{p-1}{}_{\parallel} \mathrm{S}(h_{32k,1}) \left(\sum_{j=0}^{8/p-1} \mathrm{S}(h_{8j,1})\, \mathrm{DFT}_8\, \mathrm{G}(h_{j,4}) \right) \overset{\mathrm{DMA}}{\mathrm{G}}(q_{k,2,4}) \quad (8)$$

The inner DFT kernels, which can optionally be tagged for vectorization, must be fully expanded before code can be generated (not shown). Also note that the twiddle factors can be merged to the right or to the left, and handled trivially. Since we have a composition of two parallel constructs, for performance reasons stated earlier, we convert the regular scatter to a DMA scatter which effectively handles the permutation in the succeeding DMA gather. The DMA gather at the composition interface is thus rendered into a null operation. Below, we show pseudo-C code generated from the expression above.

```
void DMA_SCATTER(source*, dest*, pk_size)
{ spu_mfcdma64(source, dest, size*8, 1, MFC_PUT_CMD) }

void DMA_GATHER(source*, dest*, pk_size)
{ spu_mfcdma64(source, dest, size*8, 1, MFC_GET_CMD) }

void  all_to_all_synchronization_barrier
{ spu_mfcstat(MFC_TAG_UPDATE_ALL);
  // Perform synchronization using mailbox msgs
}

/* Complex-to-complex DFT size 64 on 2 SPEs */
dft_c2c_64(float *X, float *Y, int spuid)
{ // Block 1 (IxA)L
  for(i:=0; i<=7; i++) // Right most gather
   { DMA_GATHER(gath_func(X,i), gath_func(T1,i), 4) }
  spu_mfcstat(MFC_TAG_UPDATE_ALL); // Wait on gather
  // compute vectoried DFT kernel of size m
  for(i:=0; i<=7; i++) // Scatter at interface
   { DMA_SCATTER(scat_func(T1,i), scat_func(T2,i), 4) }

  all_to_all_synchronization_barrier();

  // Block 2 (AxI)
  /* Gather is a no operation since the scatter above
     accounted for it */
  // compute vectoried DFT kernel of size n
  for(i:=0; i<=7; i++) // Left most scatter
   { DMA_SCATTER(scat_func(T1,i), scat_func(Y,i), 4) }

  all_to_all_synchronization_barrier();
}
```

The pseudo-code above shows the general structure of our code. In practice, several loop optimizations are performed before such code is generated, and the resulting code is compiled using a traditional compiler like `spu-gcc`.

3.2 Streaming

When working on data resident in the main memory with the Cell platform, we first need to issue explicit commands to bring them on-chip into the SPEs' local stores. Although memory access is usually expensive, the explicit control over the memory system handed to us by the Cell is an advantage since we can devise and implement an optimal caching strategy. Our caching strategy is to use a streaming technique known as multibuffering[1]. This involves computing on a block of data at a time, while simultaneously writing back the results of the previously computed block, and reading in the input for the next block to be computed, thus partially or fully hiding memory access costs. In this section, we discuss multibuffering using a single SPE.

Below, we derive rewrite rules for our basic SPL constructs that can be implemented efficiently using DMA commands implementing multibuffering. To mark multibuffered loops, we introduce an iterative sum specially tagged for streaming, $\sum\!\!\!\!\!\!\sum$. Such sums use software pipelined implementations: the gather $\overset{\mathrm{DMA}}{G}$ for iteration k needs to be executed in iteration $k-1$ and the scatter $\overset{\mathrm{DMA}}{S}$ for iteration k in iteration $k+1$. We discuss Cell-specific implementation details after deriving the rewriting rules.

Streaming $\mathbf{I_n \otimes A_m}$. Similar to parallelization, the $I_n \otimes A_m$ construct is easily streamed since it works on independent parts of the data with chunk size m. To stream this construct, we perform the same loop splitting technique as for parallelization but interpret the result as a multibuffered loop as shown below.

[1]The name arises from the multiple buffers that need to be maintained in order to use this strategy. We use the term "multibuffering" interchangeably with the term "streaming" in this paper.

$$\underbrace{I_n \otimes A_m}_{\text{stream}(s)} \rightarrow$$

$$\sum_{k=0}^{s-1} \overset{\text{DMA}}{\mathrm{S}}(q_{k,1,m\mu}) \left(\sum_{j=0}^{\mu-1} \mathrm{S}(h_{jm,1}) A_m \, \mathrm{G}(h_{jm,1}) \right) \overset{\text{DMA}}{\mathrm{G}}(q_{k,1,m\mu}), \quad \mu = n/s. \quad (9)$$

In (9), the scatter $\overset{\text{DMA}}{\mathrm{S}}(q_{km,1,m\mu})$ yields one DMA-put operation of size $m\mu$ for each iteration of the streamed loop. Similarly, the gather $\overset{\text{DMA}}{\mathrm{G}}(q_{km,1,m\mu})$ yields one DMA-get operation of size $m\mu$. A kernel of size $m\mu$ must thus fit on-chip. To illustrate what this loop does, and to show the structure of the resulting C code, we translate (9) into skeleton code:

```
// DMA_GET chunk 0; Block on completion of DMA
// DMA_GET chunk 1 (in background)
// Compute 0'th FFT chunk;
// Block on completion of outstanding DMAs
for(int k=1 to s-2) // Runs on p SPEs
{   // Initiate DMA_PUT chunk k-1;
    // Initiate DMA_GET chunk k+1;
    // Compute k'th chunk (DMAs proceed in background)
    // Block on completion of outstanding DMAs
}
// DMA_PUT chunk s-2; (in background)
// Compute chunk s-1
// DMA_PUT chunk s-1;
// Block on completion of outstanding DMAs
```

Streaming $\mathbf{A_m \otimes I_n}$. As with the parallelization, manipulating the $A_m \otimes I_n$ construct to perform streaming is more complicated. To compute each A_m within this construct, we need m points from the input vector. However, in this case, these points are non-contiguous in memory, and need to be gathered at a stride of n. Since we desire maximum possible DMA packet sizes (with a 16-byte architecture-imposed minimum) for reasonable DMA performance, our approach is to perform multiple A_m kernels at once. This allows selection of A_m kernels whose input points are contiguous in memory. We therefore perform tiling through loop splitting, leading to the rewrite rule

$$\underbrace{A_m \otimes I_n}_{\text{stream}(s)} \rightarrow \sum_{k=0}^{s-1} \overset{\text{DMA}}{\mathrm{S}}(q_{k,s,\mu}) \left(\sum_{j=0}^{\mu-1} \mathrm{S}(h_{j,\mu}) A_m \, \mathrm{G}(h_{j,\mu}) \right) \overset{\text{DMA}}{\mathrm{G}}(q_{k,s,\mu}), \quad \mu = n/s. \quad (10)$$

(10) results in m DMA-put and m DMA-get operations, each with a packet size of μ, for each iteration of the streamed loop. We use the Cell's DMA list feature for larger values of m to avoid overflowing the SPE's DMA queue.

Streaming $\mathbf{A \cdot B}$. Composing two or more streaming Σ-SPL constructs is conceptually simple: each of the stages simply read from memory and write back to memory. However, for kernels where the working set fits on-chip, we can avoid going off-chip to main memory in between the composed stages, and instead have the SPEs perform an all-to-all communication, taking advantage of the high bandwidth on-chip interconnect (Element Interconnect Bus).

Trade-offs and constraints. Similar considerations as in the parallelization case apply for streaming. Again, we maximize packet size and trade off kernel sizes between the two stages in (1).

Code generation for the streaming paradigm is similar to code generation for parallelization. We do not provide a detailed example due to space constraints.

Tagged Specification	Description
(a) Latency-optimized usage scenarios	
$\underbrace{\mathrm{DFT}_m}_{\text{spe}(p,\mu)}$	DFT parallelized across p SPEs, LS-LS
$\underbrace{\mathrm{DFT}_m}_{\text{stream}(1)}$	DFT running on a single SPE, mem-mem
$\underbrace{\underbrace{\mathrm{DFT}_m}_{\text{spe}(p,\mu)}}_{\text{stream}(1)}$	Large DFT parallelized across p SPEs, mem-mem, streamed in parts
$\underbrace{\mathrm{DFT}_{m \times n}}_{\text{stream}(min(m,n))}$	Large 2D DFT on a single SPE, mem-mem, streamed in parts
(b) Throughput-optimized usage scenarios	
$\underbrace{I_s \otimes \mathrm{DFT}_m}_{\text{stream}(s)}$	Streaming across s DFTs, using a single SPE
$\underbrace{I_p \otimes \underbrace{I_s \otimes \mathrm{DFT}_m}_{\text{stream}(s)}}_{\text{spe}(p,\mu)}$	Streaming across s DFTs, parallelization across p SPEs (small DFTs)
$\underbrace{I_s \otimes \underbrace{\mathrm{DFT}_m}_{\text{spe}(p,\mu)}}_{\text{stream}(s)}$	Streaming across s DFTs, each DFT parallelized to run on p SPEs (mid-sized DFTs)

Table 3: DFT usage scenarios for the Cell. All DFTs assume a vectorization tag with innermost nesting.

3.3 Generating optimized code for various usage scenarios

In the previous sections, we saw how we can apply our parallelization and streaming paradigms to constructs fundamental to our domain. In this section, we show how we can apply one or more of the three paradigms (including vectorization) available to us to obtain the various usage scenarios presented in Fig. 2. Table 3 shows various usage scenarios that can be described using our tags. Table 3(a) shows latency-optimized scenarios (can be used to compute a single DFT), and Table 3(b) shows throughput-optimized scenarios (available only when performing s DFTs). We now describe how to implement these cases using our framework, including a description of the formal loop merging where necessary.

Throughput optimization. In this case, we assume that we compute a large number of DFTs on contiguous chunks of the input data, with data resident in main memory. Execution of (small) DFTs in parallel across multiple SPEs and optimization using streaming (multibuffering) are the two main tools we use for throughput optimization. In the first case in Table 3(b) we specify that we want to compute s DFTs, using a single SPE (implied). This scenario is easily accomplished by using our rewrite rule (9) and generating code.

In the second case in Table 3(b) we specify that we want to compute a total of $p \cdot s$ DFTs, using p SPEs, with s DFTs streamed to each SPE. This is the simplest case we support, and thus serves well as an example to explain our framework. Converting this case to Σ-SPL, we obtain a nested sum as expected. However, the outer parallel sum that distributes the input vector across the SPEs assumes a global view of the local stores. In order to express each SPE independently reading and writing to its assigned chunk from memory, we move the outer loop's gather and scatter into the inner

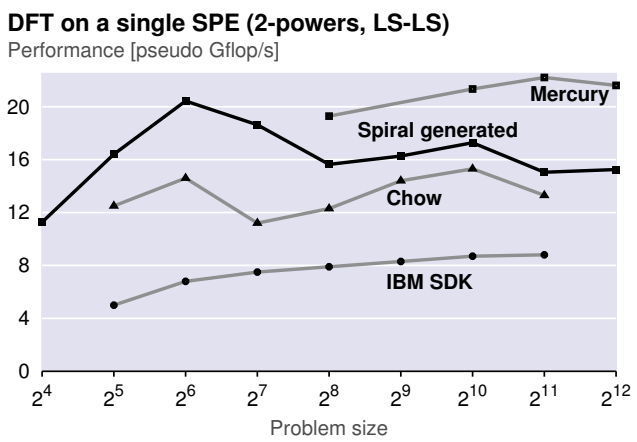

Figure 4: Single SPE performance comparison for single precision split-complex input. Higher is better.

loop, using identities in [8], as shown below:

$$\sum_{i=0}^{p-1}{}_{\parallel}^{\text{DMA}}\,\mathrm{S}(q)\left(\sum_{j=0}^{s-1}{}_{\equiv}\,\mathrm{S}(h)\,\mathrm{DFT}_m\,\mathrm{G}(h)\right)^{\text{DMA}}\mathrm{G}(q)\;\rightarrow\;\sum_{i=0}^{p-1}{}_{\parallel}\sum_{j=0}^{s-1}{}_{\equiv}^{\text{DMA}}\,\mathrm{S}(q)\,\mathrm{S}(h)\,\mathrm{DFT}_m\,\mathrm{G}(h)^{\text{DMA}}\mathrm{G}(q)\quad(11)$$

The third case in Table 3(b) involves streaming across s DFTs, each executed in parallel across p SPEs (notice the tag nesting order). This works best for medium sized kernels that benefit from parallelization. Similar to the previous case, the parallel scatter/gather must be manipulated such that each SPE to writes directly into its own chunk in memory. The difference is in the loop nesting order, as shown below.

$$\sum_{j=0}^{s-1}{}_{\equiv}^{\text{DMA}}\,\mathrm{S}(q)\left(\sum_{i=0}^{p-1}{}_{\parallel}\,\mathrm{S}(h)\,\mathrm{DFT}_m\,\mathrm{G}(h)\right)^{\text{DMA}}\mathrm{G}(q)\;\rightarrow\;\sum_{j=0}^{s-1}{}_{\equiv}\sum_{i=0}^{p-1}{}_{\parallel}^{\text{DMA}}\,\mathrm{S}(q)\,\mathrm{S}(h)\,\mathrm{DFT}_m\,\mathrm{G}(h)^{\text{DMA}}\mathrm{G}(q)\quad(12)$$

We can further optimize this scenario by using our formal loop merging framework [8] to fold the cost of converting to/from the block cyclic data format used in [2] into the DMA scatter and gather from main memory.

Latency optimization. In this case, our goal is to compute a single DFT, with vectors resident either in the local stores or in main memory. We handle the first case in Table 3(a) by breaking down the DFT into its factors as shown in (1), and applying our parallelization identities to each of them, followed by the composition rule. The second case in Table 3(a) is simply handled by wrapping our DFT kernel with DMA get and put instructions.

We handle the third case similarly, with the exception that a parallelized kernel is now used. Finally with the fourth case, we first break down the 2D DFT into its recursive factors, and then apply streaming to the composition of the factors, as presented in Section 3.2. This thus works even if the entire 2D kernel cannot be held within the local store of a single SPE, as we work on small chunks at a time. In principle, though our framework supports streaming and computing any large DFT in parts, along with parallelization, due to the complexity of the loop manipulations, these are outside the scope of this paper.

4. EXPERIMENTAL RESULTS

We now evaluate our generated DFT kernels on the Cell under various scenarios (single-SPE, multiple SPEs, and streaming from main memory), and compare our performance with the fastest available third-party implementations, whenever possible.

Experimental setup. We evaluated our generated single precision 1D and 2D DFT implementations on a single Cell processor of an IBM Cell Blade QS20 running at 3.2GHz, and double precision implementations on a PowerXCell 8i based system at Mercury Computer Systems, Inc. Our code was compiled with the GCC compiler for the SPE, `spu-gcc`, which is a part of the Cell SDK version 3.1. Our results are shown in Fig. 5, and we discuss them here.

Timing methodology. For experiments that measured the performance of latency-optimized kernels, we measured the execution time of a single kernel using the SPEs' decrementers. We timed an adequate number of iterations to ensure timing stability and precision. We measured the performance of throughput-optimized kernels by running several iterations and measuring the runtime of the steady state of the computation.

All our plots display normalized inverse runtime measured in pseudo Gflop/s, computed by $5n\log_2(n)/(\text{runtime [s]}\cdot 10^9)$, where n is the size of the 1D or 2D DFT kernel in complex samples (i.e., for a $\mathrm{DFT}_{k\times k}$, $n=k^2$).

Vector DFT on a single SPE. Fig. 5(a) and Fig. 5(b) show the single-core performance of our generated DFT kernels for 2-power and non 2-power sizes, respectively, for single precision (labelled "float") and double precision (labelled "double") data resident in the local store. These vectorized DFT kernels are important building blocks in all our parallel and streaming implementations. Our generated code achieves 16–21 Gflop/s for single precision, and 8 Gflop/s for double precision.

In Fig. 4, we compare the performance of our single-precision split-complex kernels to hand-tuned code from Mercury [5], Chow [3], and the IBM SDK FFT library [13] as measured in [3]. The maximum performance achieved by our single precision generated code is currently slower (0.73x–0.85x) than Mercury, although our code achieves its performance peak for smaller sizes. Our code is faster by a factor of 1.06x–1.6x when compared to Chow's library, and 1.83x–3x faster than IBM's SDK.

Parallel DFT across multiple SPEs, data in local stores. Due to space limitations, we henceforth focus on FFTs operating on single precision input data, although our system can generate similar code for double precision input too. Fig. 5(c) and (d) display the performance of our generated parallel, multi-SPE DFT kernels, again with data resident in the local stores. These kernels use single-SPE vectorized kernels as building blocks. We generate and evaluate code for two standard data distributions: Fig. 5(c) shows results for data in the *block cyclic* format, i.e., data is distributed across the SPEs' local stores in a round-robin fashion using a suitable block size. Fig. 5(d) shows results for the standard block distribution (i.e., data is cut into p contiguous blocks for p SPEs). We achieve close to 80 Gflop/s for a 2^{15}-point DFT across 8 SPEs using a block cyclic format, and about 53 Gflop/s for a 2^{14}-point DFT with a standard distributed format. The block cyclic data distribution requires less communication (and thus achieves better performance), compared to the standard distribution. We employ these kernels as building blocks for our streaming DFT implementations.

DFT on multiple SPEs, data in main memory. Fig. 5(e)–(g) display the performance of parallelized DFT kernels with data beginning and ending in main memory. Fig. 5(e) is a throughput-optimized version, and achieves about 30 Gflop/s on a DFT of size 2^{13} on 4 SPEs. It streams data from main memory to and the local stores, and converts data into a block cyclic data format "on the fly,"

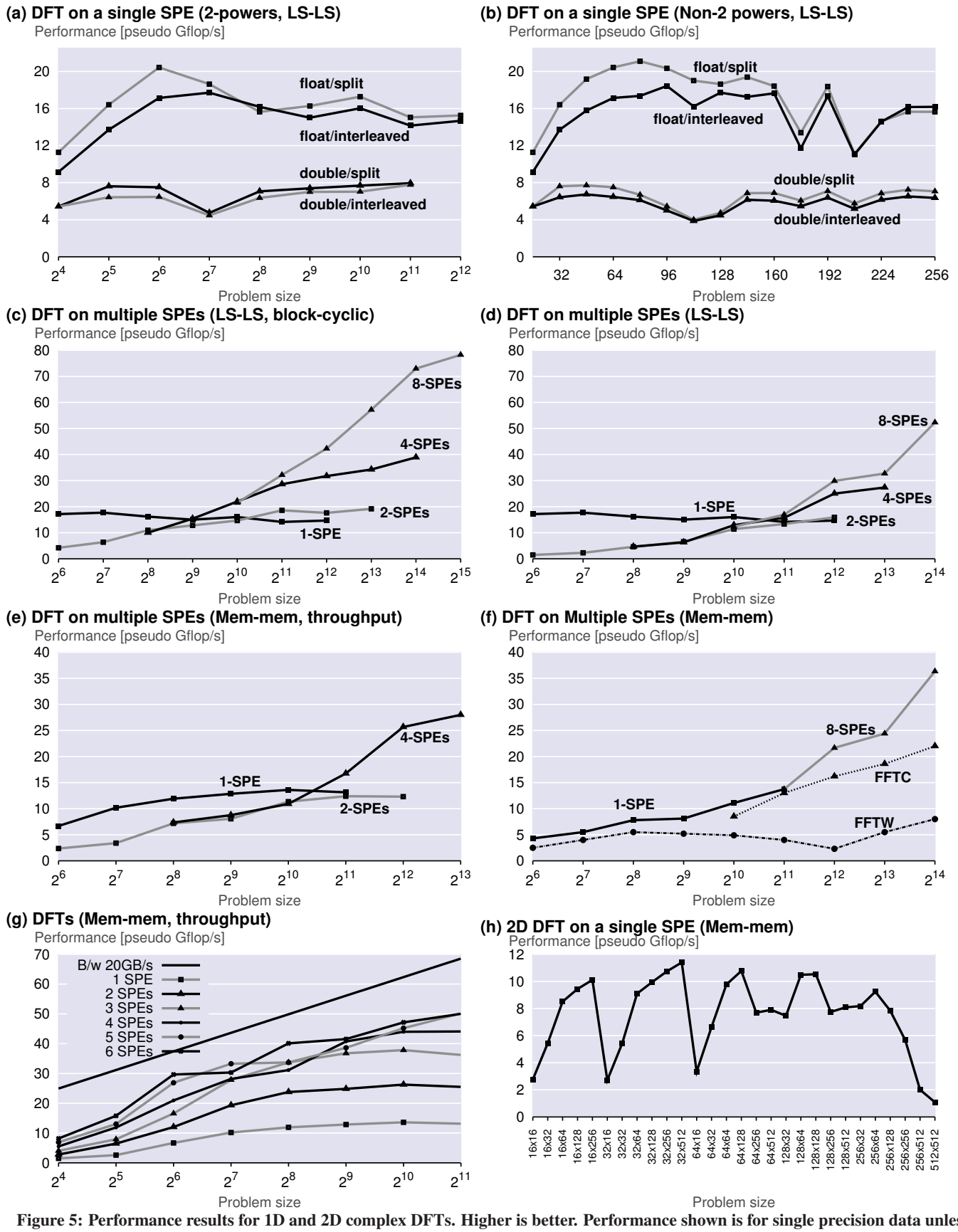

Figure 5: Performance results for 1D and 2D complex DFTs. Higher is better. Performance shown is for single precision data unless labelled "double". Plots (e) and (g) show throughput performance, all other plots present performance for latency-optimized code.

enabling the application of our fastest parallel block-cyclic multi-SPE DFT kernels. This technique does not scale well to 8 SPEs due to the small DMA packet sizes required.

Next, we compare our generated code to FFTW [11] and FFTC [1], with performance data taken from [6] and [1], respectively. Note that both implementations only provide *latency-optimized* DFT functions. In order to make a fair comparison with FFTW and FFTC, we simply added DMA load and store operations around our parallel multi-SPE kernels (shown in Fig. 5(d)). This approach does not result in highly optimized code as communication and computation are not performed in parallel. Despite this limitation, as shown in Fig. 5(f), our generated code already outperforms FFTW by a factor of 4.5x, and FFTC by a factor of 1.63x (for size 2^{14}). While we have shown all data points available for FFTC, FFTW achieves its highest performance of about 22 Gflop/s for larger sizes that are not shown in our results. Our system is currently unable to generate code for these sizes.

FFTC is limited in its effectiveness because it uses a single algorithm, hard-coded for the 5 problem sizes shown. It is also hard-coded to use 8 SPEs. While FFTW is more flexible and considers various algorithms based on its planning feature, it presumably uses the PPE to offload computation to the SPEs when possible, thus accumulating high communication costs, as opposed to being able to use algorithms that exploit the SPEs effectively.

Finally, we measure the performance of small DFT kernels (data resides in main memory but each DFT can fit in the local store of one SPE) in throughput mode, as shown in Fig. 5(g). An independent copy of the single-SPE kernel runs on each SPE. Memory bandwidth becomes the limiting factor once more than 4 SPEs are being used, and we see sustained performance of up to 50 Gflop/s.

2D DFTs. For our final experiment (Fig. 5(h)), we evaluated the performance of small-to-medium-sized 2D DFTs on a single SPE. The data begins and ends in main memory and includes problem sizes which cannot be held completely in the local store. Data thus has to make two round trips from memory to the local stores and the full 2D DFT kernel can only be latency-optimized. However, every stage by itself is streamed to reduce memory access costs. We observe performance of up to 11 Gflop/s for sizes that allow for reasonably large DMA packet sizes. Sizes that use smaller packet sizes see lower performance.

5. CONCLUSION

The Cell poses a real challenge for the developers of performance libraries. Like a shared memory multicore processor (such as Intel dual- and quadcores), memory hierarchy optimizations, vectorization, and parallelization must be performed carefully. However, both the explicit memory management coupled with complex tradeoffs betwen packet sizes and parallelism, and the possibility of latency and throughput optimizations add another level of complexity. While the automatic generation of high performance libraries from a mathematical specification is always desirable, for a platform like the Cell it is virtually a necessity. As we have shown for the DFT, using our generator (an extension of Spiral), it is possible to cover a broader range of usage scenarios and optimizations than existing hand-written libraries with potential gains in performance.

6. ACKNOWLEDGMENTS

This work was supported by NSF through awards 0325687, 0702386, by DARPA (DOI grant NBCH1050009), ARO grant W911NF0710416, and by Mercury Computer Systems. We also thank Mercury Computer Systems for allowing us to evaluate our libraries on their PowerXCell 8i based product.

7. REFERENCES

[1] D. A. Bader and V. Agarwal. FFTC: Fastest Fourier transform for the IBM Cell Broadband Engine. In *IEEE Intl. Conference on High Performance Computing*, pages 172–184, 2007.

[2] S. Chellappa, F. Franchetti, and M. Püschel. FFT program generation for the Cell BE. In *International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2008.

[3] A. C. Chow. Fast Fourier transform SIMD code generators for synergistic processor element of Cell processor. In *Workshop on Cell Systems and Applications*, 2008.

[4] A. C. Chow, G. C. Fossum, and D. A. Brokenshire. A programming example: Large FFT on the Cell Broadband Engine. Technical report, IBM, May 2005.

[5] L. Cico, R. Cooper, and J. Greene. Performance and Programmability of the IBM/Sony/Toshiba Cell Broadband Engine Processor. In *Proc. of (EDGE) Workshop*, 2006.

[6] FFTW on the cell processor. http://www.fftw.org/cell/.

[7] F. Franchetti, D. McFarlin, F. de Mesmay, H. Shen, T. W. Wlodarczyk, S. Chellappa, M. Telgarsky, P. A. Milder, Y. Voronenko, Q. Yu, J. C. Hoe, J. M. F. Moura, and M. Püschel. Program generation with Spiral: Beyond transforms. In *High Performance Embedded Computing (HPEC)*, 2008.

[8] F. Franchetti, Y. Voronenko, and M. Püschel. Formal loop merging for signal transforms. In *Programming Languages Design and Implementation (PLDI)*, pages 315–326, 2005.

[9] F. Franchetti, Y. Voronenko, and M. Püschel. FFT program generation for shared memory: SMP and multicore. In *Supercomputing (SC)*, 2006.

[10] F. Franchetti, Y. Voronenko, and M. Püschel. 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*, pages 363–377. Springer, 2006.

[11] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):216–231, 2005.

[12] J. Greene and R. Cooper. A parallel 64K complex FFT algorithm for the ibm/sony/toshiba cell broadband engine processor. In *Global Signal Processing Expo (GSPx)*, 2005.

[13] IBM. *Fast Fourier Transform Library Programmer's Guide and API Reference*, 3.0 edition, 2008.

[14] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[15] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[16] C. Van Loan. *Computational Framework of the Fast Fourier Transform*. SIAM, 1992.

[17] Y. Voronenko, F. de Mesmay, and M. Püschel. Computer generation of general size linear transform libraries. In *International Symposium on Code Generation and Optimization (CGO)*, 2009.