# Multi-dimensional Array Operations for Signal Processing Algorithms

Paolo Bientinesi[1], Nikos P. Pitsianis[2], and Xiaobai Sun[2]

[1] Aachen Institute for Advanced Study in Computational Engineering Science.
RWTH Aachen, Acchen, Germany
pauldj@aices.rwth-aachen.de
[2] Department of Computer Science, Duke University,
P.O.Box 90129, Durham, NC 27708-0129, USA
{nikos,xiaobai}@cs.duke.edu

**Abstract.** Game and graphics processors are increasingly utilized for scientific computing applications and for signal processing in particular. This paper addresses the issue of efficiently mapping high-dimensional array operations for signal processing algorithms onto such computing platforms. Algorithms for fast Fourier transforms and convolutions are essential to many signal processing applications. Such algorithms entail fast high-dimensional data array operations. Game and graphics processors typically differ from general-purpose processors in memory hierarchy as well as in memory capacity and access patterns. We characterize the memory structures from the perspective of high-dimensional array operations, identify the mismatch between algorithmic dimension and architectural dimension and then describe the consequent penalty in memory latency. An architecture supports $d$-dimensional accesses if a $d$-dimensional data array can be accessed along one dimension as fast as along any other dimension. We introduce an approach to reduce latency and provide experimental results on the STI Cell Broadband Engine as supporting evidence.

**Key words:** Signal Processing, FFT, Convolution, Game and Graphics Processors, Cell Broadband Engine.

## 1 Introduction

We present an approach for efficiently mapping important signal and image processing algorithms to multi-core processors, including game and graphics processing processors. We illustrate the approach by means of the two and three dimensional Fast Fourier transforms (2D and 3D FFT), which are ubiquitous in signal and image processing applications. The FFT represents a special class of fast algorithms that factor the operator, in this case the Discrete Fourier Transform (DFT) matrix, in order to reduce the amount of arithmetic operations. Computationally, the mathematical factorization translates to operations on data arrays such as partitioning, permutation and reintegration, and involves

different memory reference patterns and latencies. Reducing the latency in memory references without compromising the arithmetic properties of algorithms has always been a research topic of importance, given the constant evolution of the computer architectures.

We address the memory latency problem by characterizing the relationship between algorithms and architectures in terms of a match or mismatch in data array dimensionality. First we review the FFT algorithms in terms of multi-dimensional array operations. Next, we introduce the concept of architectural dimensions with respect to memory accesses to multi-dimensional arrays. From this perspective, many traditional memory systems are limited to one-dimensional array access; conversely, modern game and graphics processors typically support two or higher dimensional array accesses at one or more levels of the memory hierarchy. We identify the dimensional mismatch as a source for high latency in memory accesses and introduce an approach for resolving mismatches via algorithm transformations. Experimental results on the Sony-Toshiba-IBM Cell processor are provided, demonstrating performance up to 68 and 72 GFlops for the computation of 2D and 3D FFTs, respectively.

### 1.1 Related Work

In an early study, Williams et al. predicted that the computation of large 1D and 2D FFTs on the Cell would reach a performance of about 40-41 GFLOPs [9]. Greene and Cooper presented an algorithm, specifically optimized for a stream of 1D FFTs of size 64K, reaching a simulated performance of 90 GFLOPs [5]. Chow et al. were among the first to report performance results from a real implementation; their code, optimized for one 16K 1D FFT, attains 46 GFLOPs [3]. More recently, Bader and Agarwal presented an approach for 1D FFTs of size from 1K to 16K, showing performance in the range 10-18 GFLOPs [1]. FFTw is one of the standard libraries for computing 1D, 2D and 3D FFTs. FFTWs includes routines optimized for the Cell processor. When the input array fits in the combined local stores of the SPEs, FFTw attains around 30 GFLOPS and 15 GFLOPS for 2D and 3D FFTs respectively. These figures account for the computation of a single FFT, when the input array is in main memory. The Spiral project is also a common source of fast FFT libraries for both distributed and multicore processors [4, 2], but a library specifically optimized for the Cell is not available at this time.

## 2 Multi-dimensional Array Operations

### 2.1 The mathematical array dimension in the FFT

FFTs of dimension higher than one involve multi-dimensional data arrays. We show how multi-dimensional array operations naturally arise in one-dimensional FFTs too. The FFT of a 1D input data array $u$ is defined as $v := F_n\, u$, where $F_n$ is the DFT matrix of order $n$ with elements $F_n(k, k') = e^{-2\pi kk'\sqrt{-1}/n}$,

$k, k' = 0, 1, \cdots, n-1$. The input and output data arrays ($u$ and $v$) may be real or complex, depending on the application. If the size of the input vector is a composite number ($n = p\ q$), the DFT matrix can be represented and applied to the input data in the following factored form

$$v = F_n u = (F_q \otimes I_p)\ D_{q,p}\ (I_q \otimes F_p)\ P_{n,q}\ u. \tag{1}$$

In this expression, $I_k$ is the identity matrix of order $k$, $\otimes$ denotes the Kronecker product and the factors are applied to the array $u$ from right to left. First, $P_{n,q}$ permutes $u$ with stride $q$; next, $I_q \otimes F_p$ specifies the independent application of DFTs of size $p$ to $q$ vector segments. The diagonal matrix $D_{q,p}$ represents the "twiddle" scaling,[3] and finally the factor $F_q \otimes I_p$ specifies the independent application of DFTs of size $q$ to $p$ vector segments, each segment consisting of elements in stride $p$.

In order to facilitate the mapping from algorithms to architectures, the factorization (1) can be described in terms of two-dimensional data array operations as

$$V_{p\times q} = [\, W_{p\times q} \odot (F_p U_{p\times q})\,]\, F_q. \tag{2}$$

Here the $p \times q$ data array $U_{p\times q}$ is obtained by folding $u$ row-wise, which corresponds to the stride-$q$ permutation (factor $P_{n,q}$ in eqn. (1)). The simultaneous transforms of the columns in $U_{p\times q}$ correspond to the application of the Kronecker product factor $(I_q \otimes F_p)$. The operator $\odot$ denotes element-wise multiplication, and the $p \times q$ array $W$ is the leading $p \times q$ subarray of $F_n$, which unfolds column-wise into the diagonal elements of $D_{q,p}$. The post-multiplication of $F_q$ to the rows in the scaled 2D array corresponds to that by $F_q \otimes I_p$ in the 1D-array expression of (1). Finally, the array $V_{p\times q}$ unfolds column-wise to render the vector $v$.

1D and 2D FFTs are related through a simple relationship from a 2D-array operation perspective. Consider the 2D FFT of a $p \times q$ data array $U$:

$$V_{p\times q} = F_p\, U_{p\times q}\, F_q. \tag{3}$$

Two comments on formula (3) follow. First, the 2D FFT can be seen as consisting of two sweeps of 1D FFTs, one on the columns of size $p$, the other on the rows of size $q$. Second, notice that when we replace the scaling array $W$ in (2) with a matrix of 1s, the 2D-array expression for 1D FFT of size $n = pq$ becomes that for the 2D FFT. Moreover, formula (3) for 2D FFTs differs from formula (2) as it does not impose a specific ordering between $F_p$ and $F_q$.

Higher dimensional array operations are involved in both 1D FFT and higher dimensional FFTs, when the basic factorization in (1) is recursively applied to $F_p$ and $F_q$.

### 2.2 Dimensional match and mismatch

When memory access latency is uniform across the multi-dimensional array in the FFT, the cost in memory latency is proportional to the cost in arithmetic op-

[3] $D_{q,p}$ contains the $q$ powers $(0, \ldots, q-1)$ of $e^{-2\pi\sqrt{-1}[0:n-1]/n}$, with $n = p\ q$.

erations. This is the ideal case when the mathematical and architectural dimensions match. On most modern hierarchical memory structures, however, memory access latency is not uniform. In the extreme case of 1D array memory accesses, 2D data arrays are sequentialized row-wise or column-wise. If the data placement is row major, then the row-wise accesses are much faster than those column-wise and the latency for the transform along the major axis is much lower than that along the minor axis. We refer to such a situation as a dimensional mismatch. A common technique to reduce the overall memory latency cost due to dimensional mismatch is to carry out the row-wise transform first, then transpose the array, and perform another row-wise transform. The transposition exposes additional latency in memory accesses. Recent game and graphics processors support equal memory accesses for two or higher dimensional arrays. We now extend the concept of dimensional match to high-dimension memory architecture. When the mathematical and architectural dimensions mismatch, one may map a high-dimensional array to a lower-dimensional memory by data placement in subarrays. When mapping a 2D array onto a 1D memory, the subarrays in data placement are rows. The dimension of the subarrays matches the architectural dimension. The dimensional axes of the subarrays are major ones. The memory accesses across the subarrays, or along a minor axis, may incur substantially higher latency. The corner turn of a 2D array to correct a dimensional mismatch, in general requires to swap a minor axis to a major one, with respect to the architectural dimension.

It is important to identify the architectural dimension at each memory level in which a multi-dimensional array resides and the relative match or mismatch with respect to the mathematical dimension of the array. In the case of a match, unnecessary dimensional swaps should be eliminated; for a mismatch instead, one shall determine a data placement scheme and a FFT factorization to reduce unnecessary dimensional swaps and subsequent data movements. In a parallel architecture, one has an additional opportunity to reduce the overall latency because the data accesses across the subarrays in the major axes can be potentially carried out in parallel with arithmetic operations. In the next section we illustrate this approach with a distributed FFT on the STI-Cell.

## 3 Distributed FFT on the Cell

### 3.1 The Cell Broadband Engine

The Cell Broadband Engine (Cell) is a low-power multi-core processor that provides high computing performance; it was designed targeting gaming and multimedia applications on small electronic devices. Here we give a brief description of the architectural features that play a key role in the development of algorithms for FFTs; for further details on the architecture please refer to [6].

The Cell is composed of one master processor (PPE), up to eight Synergistic Processing Elements (SPEs), and a broadband interconnect bus (EIB). Parallelism is supported at different granularity levels through a number of mechanisms. Our discussion focuses on the computation on data arrays that are dis-

tributed among the eight SPEs. Each SPE has a theoretical peak performance of 25.6 GFLOPs when operating in single precision floating point arithmetic, for a total of 204.8 GFLOPs. It has 128 local 128-bit registers, and a local store of 256KB, for both instructions and data. Most of the instructions can process 128-bit operands in single-instruction-multiple-data (SIMD) mode; in particular, single-precision floating-point operands, or 32-bit integer operands, are processed as 4-way SIMD vectors, fully pipelined.

For data movements, the SPEs can issue direct memory access (DMA) commands for explicit scratch-pad memory accesses across the local stores and to or from the main memory. Data transfers between any local store and the main memory are handled by the memory interface controller (MIC). The DMA controllers and the MIC are all connected to the EIB and operate independently from the SPEs, providing parallelism between computation and data movement. The EIB supports a peak bandwidth of 204.8 Gbytes/s for data transfers among the PPE, the SPEs and the MIC, while the MIC provides a peak bandwidth of 25.6 Gbytes/s to and from main memory.

In order to gain insights on the features of the Cell, we performed a series of benchmarks. The most important finding concerns the dimensionality of the SPE's local store. We observed that a two-dimensional array of properly aligned 128-bit elements, when stored in the SPE's local memory, can be accessed by rows or by columns essentially at the same speed; the same conclusion can be drawn for a three-dimensional array. In other words, SPEs' local memory supports 2 and 3 dimensional memory accesses. The immediate consequence is that array transposition can and should be avoided, since memory accesses with stride do not incur penalties (some of the currently available libraries do not exploit this feature and have explicit transposition).

In our analysis we studied the different mechanisms provided by the Cell for data transfers (SPE-to-SPE and SPE-to-main memory) and synchronization: DMA transfers, "mailbox" messages and signals. We concluded that the DMA is the only effective way for moving data between main memory and the SPEs' local stores, but, to attain high bandwidth, data needs to be transferred in large parcels (size $\geq$ 2 KB) and must be aligned. Conversely, the DMA is not suited for transfers of small size (size $\leq$ 128 Bytes) and for synchronization purposes. Synchronization and very small PPU-SPE transfers are better achieved through mailboxes. In terms of communication patterns, we observed that the EIB sustains high bandwidth for pairwise exchanges among the SPEs, and that the "DMA-list", a mechanism that allows many DMA transfers to be scheduled at once, attains significantly better bandwidth than a sequence of individual DMA transfers. Our measurements are in line with those reported by Kistler et al. [7].

In terms of array operations, the SPEs support a 4-way parallelism at the instruction level (SIMD): 128-bit operands can be effectively used as 4 independent 32-bit floating point numbers. Therefore, data arrays can be logically seen as vectors of quadruplets (QUADs), and the quadruplet becomes the unit of storage and computation. Our benchmarking showed that any array operation

**Algorithm 1** Distributed algorithm for 2D FFTs on 4 SPEs.

**Distributed Algorithm** for a $n_1 \times n_2$ FFT with 4 SPEs.

**0.** Data array partition and placement. $P_{\text{in}} = P_{4,2}$.

$Y_i := X\,(1{:}n_1,\ P_{\text{in}}(i) \times n_2/4{+}[1{:}n_2/4]),$ $\quad i = 0, \ldots, 3.$

**1.** Data exchange, butterfly and twiddle operations.

$Y_i := Y_i + Y_{i+1};$ $\quad i \in \{0, 2\}.$

$Y_i := Y_{i-1} - Y_i;$ $\quad i \in \{1, 3\}.$

$Y_i := (Y_i + Y_{i+2}) \odot W_i;$ $\quad i \in \{0, 1\}.$

$Y_i := (Y_{i-2} - Y_i) \odot W_i;$ $\quad i \in \{2, 3\}.$

**2.** Local FFTs.

$Y_i := \text{FFT}_2(Y_i),$ $\quad i = 0, \ldots, 3.$

**3.** Data permutation and write out. $P_{\text{out}} = P_{n_2,4}$

$(P_{\text{out}}^{-1} X)\,(1{:}m,\ i \times n_2/4{+}[1{:}n_2/4]) := Y_i.$ $\quad i = 0, \ldots, 3.$

that requires access to any of the 4 floating point numbers composing one QUAD is significantly slower than a corresponding algorithm that maintains the QUAD structure intact.

### 3.2 The Algorithm

We start by discussing the design of an algorithm for 2D FFTs targeting 4 SPEs; the algorithm will then be extended to 8 SPEs and to 3D FFTs. The challenge in tailoring an algorithm for the Cell consists in balancing the latencies due to data movements and computation. The granularity of the data transfers is dictated by the bandwidth of DMA transfers. The DMA, to be interpreted as a dedicated thread responsible for data movements, reaches high bandwidth only when large parcels of data are transferred. This results in strict constraints on the granularity of the algorithm at the distributed level.

Let $X, Y \in R^{n_1 \times n_2}$; our goal is to compute the array $Y$ such that

$$Y = F_{n_1} X F_{n_2}. \tag{4}$$

Array $Y$ is the 2D FFT of $X$. In order to distribute the work over the SPEs, we factor $F_{n_2}$ as in Eqn. 1, setting $q = 4$ and $p = n_2/4$:

$$F_{n_2} = \left(F_4 \otimes I_{n_2/4}\right) D_{4,n_2/4} \left(I_4 \otimes F_{n_2/4}\right) P_{n_2,4}. \tag{5}$$

Similarly, setting $q = 2$ and $p = 2$, $F_4$ reduces to $P_{4,2}\,(I_2 \otimes F_2)\,D_{2,2}\,(F_2 \otimes I_2)$; now, by substituting $F_4$ into (5), $F_{n_2}$ becomes

$$\left(P_{4,2} \otimes I_{n_2/4}\right)\left(I_2 \otimes F_2 \otimes I_{n_2/4}\right)\left(D_{2,2} \otimes I_{n_2/4}\right)\left(F_2 \otimes I_2 \otimes I_{n_2/4}\right) D_{4,n_2/4} \left(I_4 \otimes F_{n_2/4}\right) P_{n_2,4},$$

where the factors are applied to the input matrix $X$ (in Expression (4)) from left to right. In details:

1. $\left(P_{4,2} \otimes I_{n_2/4}\right)$ corresponds to the partitioning of the the input data array and to the transfer of the array from main memory to the SPE's local storage. The granularity for this operation is the data volume divided by the number of SPEs, i.e., it is very coarse.
2. $\left(I_2 \otimes F_2 \otimes I_{n_2/4}\right)$ translates to a distributed butterfly, the sum/difference of the data array that is local to an SPE with the data array that is local to a partner SPE. This stage requires SPE-to-SPE data transfers.
3. $\left(D_{2,2} \otimes I_{n_2/4}\right)$ represents a twiddle scaling operation, performed locally by each SPE. For performance reasons, this stage has to be merged with a butterfly stage.
4. $\left(F_2 \otimes I_2 \otimes I_{n_2/4}\right)$ corresponds to a distributed butterfly stage; it is the same as step 2 but with different partner pairs.
5. $D_{4,n_2/4}$ equals a twiddle scaling; it is the same as step 3.
6. $\left(I_4 \otimes F_{n_2/4}\right)$, in conjunction with the factor $F_{n_1}$, result in the computation of 2D FFTs local to each SPE.
7. $P_{n_2,4}$ corresponds to a final permutation of the local data followed by a transfer to main memory. The granularity for this stage is the size $n_1$ of the input data array.

A distributed algorithm that implements these stages is illustrated in Alg. 1. Arrays $X$ and $Y_i$ are stored in main memory and local to the $i$-th SPE, respectively; operations of the form $Y_i := Y_i \pm Y_{i+j}$ require data exchanges between SPEs $i$ and $i + j$. In order to avoid idle time due to communication latency, the data arrays are segmented and exchanged in a streaming fashion; this allows the computation to start as soon as the first block of data is exchanged, and all the successive data transfers overlap with computation. The time diagram in Fig. 1 illustrates the activity of the SPEs and DMA during the execution of the distributed algorithm. The arrows indicate data exchanges; the gray and white boxes show the cost of data movement and synchronization. These boxes overlap almost perfectly with computation, represented by the pink boxes with brick pattern. At the top of each bar, the blue box with diagonal pattern indicates the local FFT computation; this stage is the most time consuming and requires no communication among SPEs. Many applications require the computation of a sequence of FFTs; in this scenario, while the SPEs are calculating one FFT, the DMA is responsible of downloading to main memory the result of the previous iteration and load the input for the next one. When the size of the input data is large enough for the DMA to perform at high speed, the overlap between computation and data movement is nearly complete.

An algorithm for 8 SPEs is easily obtained by factoring $F_{n_2}$ as in Eqn. (1), setting $q = 8$ and $p = n_2/8$: $F_{n_2} = \left(F_8 \otimes I_{n_2/8}\right) D_{8,n_2/8} \left(I_8 \otimes F_{n_2/8}\right) P_{n_2,8}$, and decomposing $F_8$ as $P_{8,4} \left(I_2 \otimes F_4\right) D_{2,4} \left(F_2 \otimes I_4\right)$. The resulting algorithm is analogous to the one shown in Alg. 1, with the addition of one extra butterfly-twiddling stage. In Alg. 2 we present a distributed algorithm for 8 SPEs for the computation of 3D FFTs. The only difference with respect to the 2D counterpart is at Stage 2, when 3D FFTs, instead of 2D FFTs, are computed.

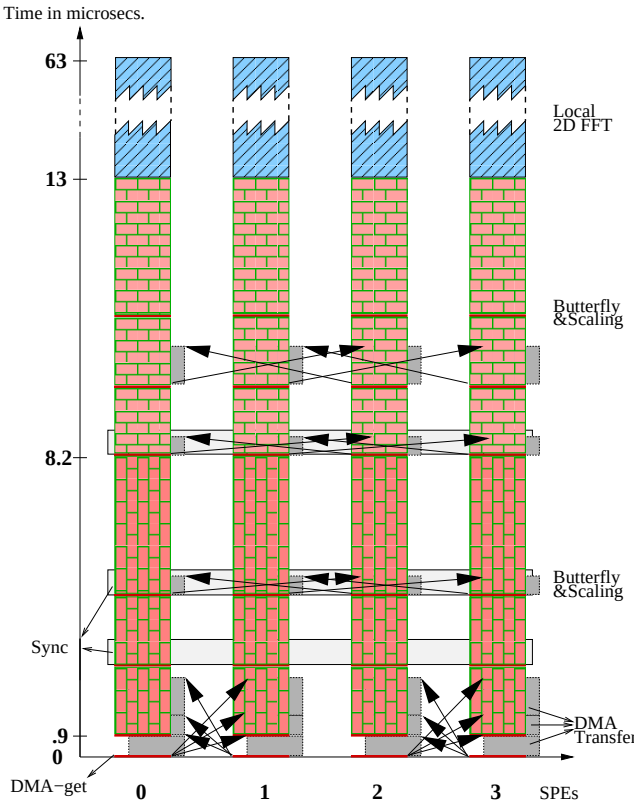


**Fig. 1.** The processor-time diagram for the algorithm described in Alg. 1, executed on 4 SPEs. Different operations are shown in different colors and patterns: the local 2D FFTs are in blue with diagonal grid, the butterfly-twiddle operations are in pink with brick pattern, DMA commands are the red thin bars, DMA data transfers are the dark gray boxes, and the synchronizing actions appear as light gray boxes.

### 3.3 Experimental Results

We present performance for two different scenarios: 1) computation of a single 2D or 3D FFT, 2) computation of a stream of FFTs. In both cases we assume that the input array is stored in main memory, and the output array is written back to main memory. In the considered test cases, our implementation outperforms FFTW (scenario 1) and attains performance as high as 72 GFlops (scenario 2).

The experiments are carried out on an IBM BladeCenter QS20 (3.2GHz) using the eight SPEs of one of the two Cell processors. The local FFT routines are written in C and compiled with the GNU `spe-gcc` compiler (4.1.1) and flag `-O3`. All the reported results are averages over at least 20 iterations and refer to single precision complex data.

**Algorithm 2** Distributed algorithm for 3D FFTs on 8 SPEs.

**Distributed Algorithm** for a $n_1 \times n_2 \times n_3$ FFT with 8 SPEs.

**0.** Data array partition and placement. $P_{\text{in}} = P_{8,4}(I_2 \otimes P_{4,2})$.

$Y_i := X\left(1{:}n_1,\ 1{:}n_2,\ P_{\text{in}}(i) \times \frac{n_3}{8} + [1{:}\frac{n_3}{8}]\right)$, $\quad i = 0, \ldots, 7.$

**1.** Data exchange, butterfly and twiddle operations.

$Y_i := Y_i + Y_{i+1}$; $\quad i \in \{0, 2, 4, 6\}.$

$Y_i := (Y_{i-1} - Y_i)d_i$; $\quad i \in \{1, 3, 5, 7\}.$

$Y_i := (Y_i + Y_{i+2})w_i$; $\quad i \in \{0, 1, 4, 5\}.$

$Y_i := (Y_{i-2} - Y_i)w_i$; $\quad i \in \{2, 3, 6, 7\}.$

$Y_i := (Y_i + Y_{i+4})W_i$; $\quad i \in \{0, 1, 2, 3\}.$

$Y_i := (Y_{i-4} - Y_i)W_i$; $\quad i \in \{4, 5, 6, 7\}.$

**2.** Local FFTs.

$Y_i := \text{FFT}_3(Y_i)$, $\quad i = 0, \ldots, 7.$

**3.** Data permutation and write out. $P_{\text{out}} = P_{n_2,8}$

$(P_{\text{out}}^{-1} Y)\left(1{:}n_1,\ 1{:}n_2,\ i \times \frac{n_3}{8} + [1{:}\frac{n_3}{8}]\right) := Y_i.$ $\quad i = 0, \ldots, 7.$

For the computation of 2D and 3D FFTs local to each SPE (Stage 2 in Alg. 1 and 2), we developed a set of routines that take advantage of the dimensionality of SPEs' local memory. In particular, a 2D scalar data array is folded in the first dimension into a 2D array of vectors of QUADs. The FFT is therefore decomposed along the and across the QUAD dimension, according to the factorization shown in 5. Four FFTs of QUADs and a local transposition across four consecutive QUADs are then performed (this operation is shown as a shuffle example in [8]). No other transposition is needed. The absence of a global transposition of the entire array results in a significant performance improvement.

| Array Size | Parallel on 8 SPEs | | | |
|---|---|---|---|---|
| | Total GFLOPs | GFLOPs/SPE | cycles$\times 10^3$ | $\mu$s |
| $256 \times 256$ | 42.1 | 5.26 | 388 | 124.3 |
| $32 \times 32 \times 64$ | 44.1 | 5.5 | 371 | 118.8 |

**Table 1.** Performance and execution time (in $10^3$ cycles and $\mu$-seconds), for the computation of one FFT with the input array read from and written to main memory.

Tables 1 and 2 show the performance of FFTs on 8 SPEs, up to local memory capacity. Table 1 refers to the computation on a single FFT, while the figures for Table 2 are based on the execution time to complete each FFT while computing a streaming sequence of FFTs. The number of arithmetic operations for an array of volume $n$ is counted as $5n \log_2(n)$. The calculation of the twiddle factors is performed once for the entire sequence and it is not included in the reported time.

| Array Size | Parallel on 8 SPEs | | | |
|---|---|---|---|---|
| | Total GFLOPs | GFLOPs/SPE | cycles$\times 10^3$ | $\mu$s |
| $128\times256$ | 46 | 5.75 | 167 | 53.4 |
| $256\times128$ | 46.5 | 5.81 | 165 | 52.8 |
| $256\times256$ | **67.9** | 8.48 | 241 | 77.2 |
| $32\times32\times32$ | 55.9 | 6.98 | 137 | 43.9 |
| $32\times16\times64$ | 57.1 | 7.13 | 134 | 43.0 |
| $32\times32\times64$ | **72.1** | 9.01 | 227 | 72.7 |

**Table 2.** Performance and execution time for the computation of a stream of FFTs. In bold the performance for the problems of maximum volume.

*Acknowledgments:* This work is supported by the DESA and MTO programs at DARPA, and by grant GSC 111 of the *Deutsche Forschungsgemeinschaft* (German Research Association). We wish to thank the Texas Advanced Computing Center and the HPC Consortium for providing us access to a Cell Blade and the referees for their constructive comments.

## References

1. D. A. Bader and V. Agarwal. FFTC: Fastest fourier transform for the IBM cell broadband engine. In *The 14th Annual IEEE International Conference on High Performance Computing (HiPC 2007)*, pages 172–184. Springer-Verlag, 2007.
2. A. Bonelli, F. Franchetti, J. Lorenz, M. Püschel, and C. W. Ueberhuber. Automatic performance optimization of the discrete Fourier transform on distributed memory computers. In *In International Symposium on Parallel and Distributed Processing and Application (ISPA)*, pages 818–832, 2006.
3. A. C. Chow, G. C. Fossum, and D. A. Brokenshire. *A Programming Example: Large FFT on the Cell Broadband Engine.* Presented at GSPx 2005 - Pervasive Signal Processing Conference and Expo, Santa Clara, CA., 2005.
4. F. Franchetti, Y. Voronenko, and M. Püschel. FFT program generation for shared memory: SMP and multicore. In *Supercomputing (SC)*, 2006.
5. J. Greene and R. Cooper. A parallel 64k complex FFT algorithm for the IBM/Sony/Toshiba Cell Broadband Engine processor. In Tech. Conf. Proc. of the Global Signal Processing Expo (GSPx), 2005.
6. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.
7. M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.
8. IBM Systems and Technology Group. *Cell Broadband Engine Programming Handbook, Version 1.1.* Hopewell Junction, NY 12533, `http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs`,, April 2007.
9. S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.