euronoise

Elimination of corner-turning in FFT-based sonar array beamforming

Jacob Barhen[a], Travis Humble[a] and Michael Traweek[b]

[a]Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge, TN 37831-6015, USA
[b]Office of Naval Research, 875 North Randolph Street, Arlington, VA 22203, USA
barhenj@ornl.gov

Abstract. The emergence of ultra-low power multicore processors will open unprecedented opportunities for implementing sophisticated signal processing algorithms faster and within a much lower energy budget. In that context, the concept of "corner turning" has been, for many decades, at the heart of array beamforming via Fourier transforms. As widely reported in the open literature (both for sonars and radars), the computational sequence involving corner turning operations, i.e., the sequence: temporal Fourier transforms → data cube corner turning → spatial Fourier transforms, constitutes a major obstacle to achieving high-performance and lower power dissipation (by reducing the number memory accesses). To date, leading industry providers still include explicit corner turning stages in their computational flow architectures for multidimensional array processing. The primary innovation reported in this paper addresses the development of a computational scheme that avoids altogether the corner turning stage. We discuss its implementation on currently available IBM CELL multicore technology and provide initial timing results.

1 Introduction

With the emergence of ever stealthier underwater targets, there is a continuing need to develop innovative approaches for near–real–time remote detection, classification, localization, and tracking. The *complexity of calculations* required to perform these tasks increases dramatically with the rapidly growing sophistication of the required algorithms, as well as with capabilities of the deployed sensor arrays. This results in substantial processing power requirements that cannot readily be met even with leading-edge conventional computing hardware.

To illustrate this challenge, consider phased acoustic arrays (PAA). For instance, two-dimensional PAA provide an attractive means for precise and reliable close-in tracking and avoidance of quiet targets. From a computational perspective, one of the most demanding aspects of PAA data processing involves the beamforming (BF) algorithms.

As widely reported in the open literature, the computational sequence involving corner turning operations, i.e., the sequence: temporal Fourier transforms → data cube corner turning → spatial Fourier transforms, constitutes one of the primary obstacles to achieving high-performance (and lower power dissipation) in sensor arrays. Even with the emergence of novel multicore processors, leading system providers still include explicit corner turning stages in their computational flow architectures for array processing.

The *primary innovation* reported in the present article is the development, implementation, and demonstration of a *massively parallel computational scheme that avoids altogether the corner turning* stage. We are presenting the details of this scheme in the context of the IBM Cell multicore processor. However, the underlying paradigm is general, and could readily be specialized to other multicore architectures, ranging from existing quad-core AMD and Intel processors, to upcoming computational platforms with a much larger number of cores.

2 Corner turning in FFT BF

The concept of "corner turning" has been, for many decades, at the heart of beamforming via Fourier transforms. Early articles (see, e.g., [1]) refer to the concept of corner turning "memory" (CTM). Digitally filtered samples from each hydrophone in a 2D PAA are initially stored sequentially in a memory block. These data sequences are then Fourier transformed into the frequency domain, and stored into the CTM, a second auxiliary block. The CTM storage is organized in such a manner as to enable efficient spatial processing, where data need to be invoked in terms of spatial (i.e., hydrophone) order rather than in sequential (temporal or temporal frequency) order. For a one dimensional sensor array, this corresponds simply to transposing the space vs temporal frequency matrix. For 2D arrays, the operation can be envisioned as a set of rotations and mirror reflections.

A substantial number of papers have been published in this area, both for sonars (see, e.g., [2, 3]) as well as radars (see, e.g., [4, 5]). There has also been growing interest in implementing "multidimensional" FFTs on parallel processors [6]. Often, it is convenient to envision a PAA beamforming process as such a "multidimensional" FFT, where, for a 2D sensor array, one of the dimensions is temporal and two dimensions are spatial. There have also been proposals for new software languages [7] to address the parallelization challenge. In terms of hardware implementation, there has been much emphasis on energy-efficient FPGAs [8] or other dedicated systems [5].

The sequel of this paper is organized as follows. Section 3 provides a brief background on the multicore computational platform. Then, the new *no-corner-turning* BF paradigm is proposed in Section 4. Practical implementation and timing results are addressed in Section 5. Finally, conclusions reached so far are given in Section 6.

3 Computational platform

Four parameters are of paramount importance when evaluating the relevance of emerging computational platforms for time-critical, embedded applications. They are: (1) the computational speed, measured in GFLOPS or in GOPS (giga floating point or integer operations per second), (2) the communication speed, i.e., the I/O and inter-processor data transfer rates measured in GB (giga bytes) per second, (3) the power dissipated, measured either in Watts or in pico-Joules per operation, and (4) the processor footprint, expressed either in area or volume units, as appropriate. For each of these parameters, one could compare the BF performance for different hardware platforms and software (e.g., compilers) tools. Results reported here refer to the IBM CELL processor.

In 2000, IBM, Sony, and Toshiba formed an alliance to develop a revolutionary computational platform for game and video applications and for numerically intensive tasks

in science and engineering. The Cell Broadband Engine™ architecture [9] is the extraordinary resulting product of 5 years of sustained, intensive R&D effort (involving over $ 400 M investment). It is comprised of one multithreaded 64-bit PowerPC processor element (PPE) with two levels of globally coherent cache, and 8 synergistic processor elements (SPEs). Each SPE consists of a processor (SPU) designed for streaming workloads, local memory, and a globally coherent DMA engine. Computations are performed in 128-bit wide SIMD. An integrated high-bandwidth element interconnect bus (EIB) connects the nine processors and their ports to external memory and to system I/O.

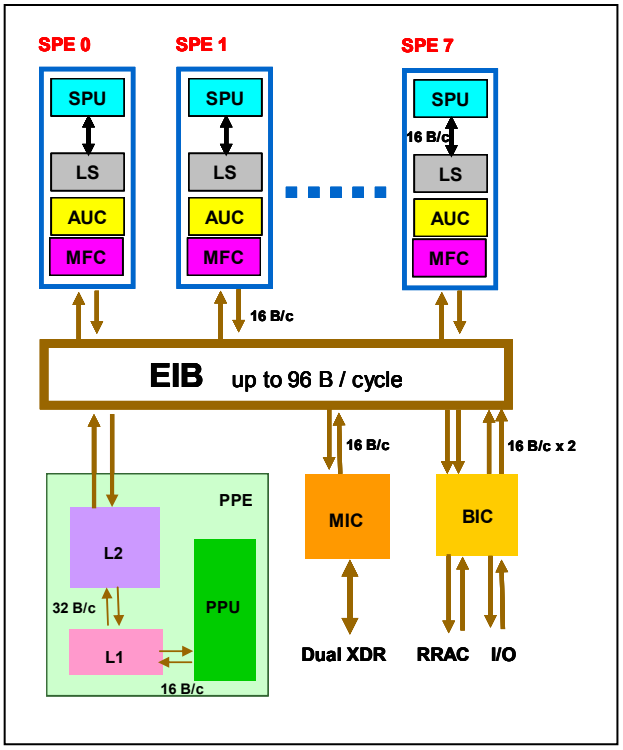

Fig 1. CELL processor architecture.

The key CELL design parameters are as follows. Each SPE comprises a 25.6 GFLOP (single precision) synergistic processing unit (SPU), a 256 KB local store memory, a 2×25.6 GB/s DMA engine, and 128 128-bit registers, which enable SIMD-type exploitation of data parallelism. It is designed to dissipate 4W at a 4 GHz clock rate. The EIB provides 2 × 25.6 GB/s memory bandwidth to each SPE, and allows external communication (I/O) up to 35 GB/s (out), 25 GB/s (in). The PPE has a 64-bit RISC PowerPC architecture, a 32KB L1 cache, a 512 KB L2 cache, and can operate at clock rates in excess of 3 GHz. The total power dissipation is estimated nominally around 100W per node (not including system memory and NIC). The total peak throughput exceeds 200 GFLOPS for a total communication bandwidth above 200 GB/s.

Because of their highly competitive costs, we initially acquired several Sony PS3 platforms. Each such platform includes a CELL processor, but only 6 out of 8 SPE cores are available to the user. Thus, the PS3s can be employed for algorithm development, debugging and testing, prior to porting code to 9-cores (PPE + SPEs) CELL hardware.

4 The no-corner-turning paradigm

We will consider an acoustic sensor array comprising N_V vertical and N_H horizontal nodes (omni directional hydrophones). Over a time interval of interest, N_S data samples are generated at each node. We assume that the properly conditioned data for each hydrophone are stored in an increasing temporal order, producing a data structure representation illustrated in Fig 2.

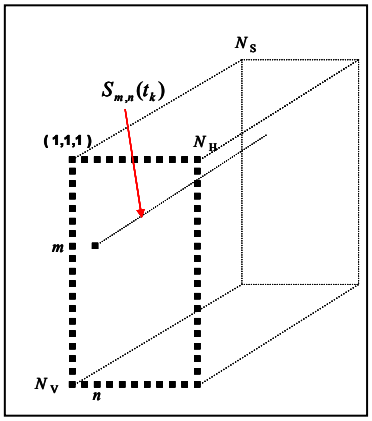

Fig 2. PAA a structure. In the figure, the symbol ■ denotes the physical sensors. Only the array boundaries and one interior sensor are shown. The dotted lines (…) refer to the data samples generated at each sensor.

To achieve beamforming, we consider a frequency domain spatial Fourier transform paradigm. In order to enable a more precise discussion, let us denote by $S_{m,n}(t_k)$ the k-th sample measured at time t_k, associated with a hydrophone located at position (x_n, y_m) in the array. It is clear that the indices m and n vary from 1 to N_V and 1 to N_H, respectively, while k spans the range 1 to N_S. The frequency domain representation (see Fig 3) is obtained, conventionally, through the following algorithm

$$\begin{aligned}&\text{DO } n = 1 : N_H\\ &\quad \text{DO } m = 1 : N_V\\ &\qquad \{\Phi_{m,n}(f_\nu) \mid \nu = 1 : N_F\} = F_t[\{S_{m,n}(t_k) \mid k = 1 : N_S\}]\\ &\quad \text{ENDDO}(m)\\ &\text{ENDDO}(n)\end{aligned} \tag{1}$$

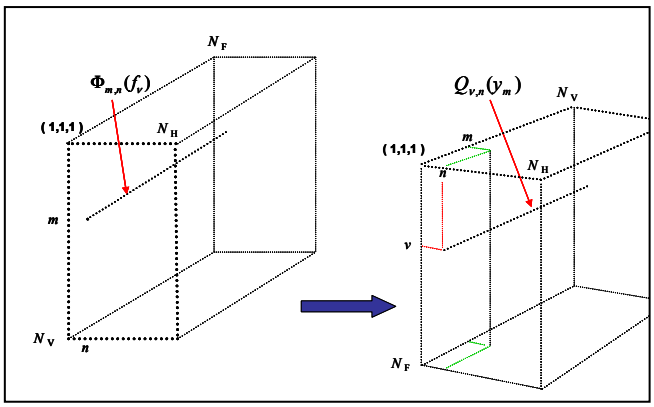

Fig 3. Illustration of corner turning

In Eq (1), Φ_t denotes a one-dimensional Fourier transform (typically an FFT) in the time domain, and N_F refers to the number of frequencies f. The data in cube **S** are real, while the data in cube □ are complex (and $N_F = N_S / 2$). To proceed with the spatial Fourier transforms, data need to be "corner turned". This is illustrated in Fig 3.

On the left hand side, is the sample data cube in the frequency domain, □. In the "corner turned" data cube **Q**, the frequency samples are stored in contiguously increasing hydrophone order along the N_V direction. Specifically, $Q_{\nu,n,m} = \Phi_{m,n,\nu}$. Clearly, "corner turning" is a misnomer, as the transformation is geometrically more involved. The primary benefit is, however, that the spatial Fourier transforms along the N_V direction can now be efficiently implemented.

$$\begin{array}{l}\text{DO } n = 1 : N_H \\ \quad \text{DO } \nu = 1 : N_F \\ \quad\quad \{P_{\nu,n}(\kappa_\sigma^y) \mid \sigma = 1 : N_V\} = F_y[\{Q_{\nu,n}(y_m) \mid m = 1 : N_V\}] \\ \quad \text{ENDDO}(\nu) \\ \text{ENDDO}(n)\end{array} \quad (2)$$

In Eq (2), κ^y denotes the spatial frequency along the y (i.e., the N_V) direction. A similar procedure (corner turning of **P**), is applied in performing the spatial Fourier transform Φ_x along the N_H direction.

The main computational and energy (because of memory access) BF costs arise from the sequence of alternate directions FFTs with corner turning. With the emergence of novel multicore processors, it is vital to design algorithms that carry out such a sequence of computations in a near optimal way. Before proceeding, we make the following assumptions. We consider a processing system with N_P homogeneous computational cores. Thus, N_P refers to 8 SPE cores on the standard IBM CELL, or 6 SPE cores on the Sony PS3.

Basic Idea. The entire computational sequence (temporal Fourier transforms → corner turning → spatial Fourier transforms) can be implemented most efficiently by properly combining sequential processing, parallel processing, and concomitant data transfers. The high level computational flow is as follows.

DO $n = 1 : N_H$

- distribute n–th data plane of **S** across cores $\{C_1..C_{N_P}\}$
- carry out temporal Fourier transforms
  - in parallel, across all cores
  - sequentially, but in vectorized form, within each core
- generate synthetic templates for spatial Fourier transform (no corner turning needed!)
- carry out spatial Fourier transforms across all cores
  - in parallel, across all cores
  - sequentially, but in vectorized form, within each core
- collect and combine resulting data from all cores (data transfer carried out in parallel with computation)

ENDDO (n)

This involves sequential processing of data planes extracted from the data cube **S**, segmentation of these planes and distribution of the resulting data blocks across the homogeneous cores, vectorized processing of each block within its respective core, parallel processing of all cores, and overlapping within each core of computation and data transfer. We assume that, after conditioning, the data cube **S** is stored in RAM from which it can be accessed via a DMA engine by each core.

Data Plane Segmentation. As shown in Figure 4, each data plane extracted from cube **S** contains N_V rows of data samples. Each row is a vector of length N_S. We distribute these data, grouped by sets of rows, to the available N_P cores.

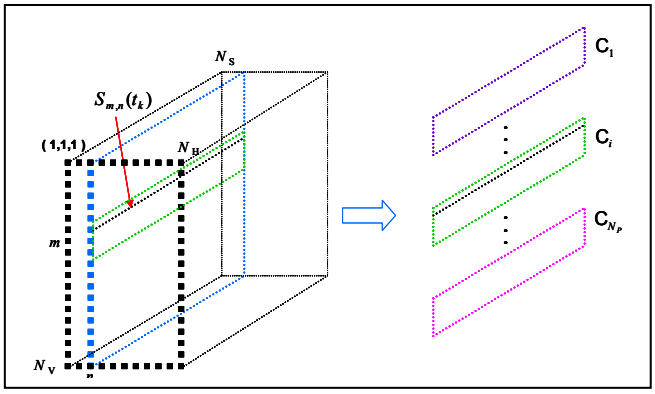

Fig 4. Data Partitioning.

Note that if $\text{MOD}(N_V, N_P) = 0$, where MOD designates the "modulo" function, then perfect load balance can be achieved, and each core will store $N_R = N_V / N_P$ rows. In the current version of the IBM CELL processor, only 256KB of memory are available at each core. This memory needs to cover code, data, and stack. For large arrays, the algorithms outlined below will still apply, but each data plane would have to be processed in more than one pass through the cores.

Temporal Fourier Transforms. Within each core, we now need to carry out a set of one dimensional FFTs. Since for each FFT the computational complexity is $N_S Log_2(N_S)$, the per-core complexity (assuming N_R rows per core and no FFT transverse set-vectorization) is $N_R N_S Log_2(N_S)$. At each core C_i ($i = 1 : N_P$), when processing data plane n

$$\begin{array}{l}\text{DO } m = (i-1)N_R + 1 : iN_R \\ \quad \{\Phi_{m,n}(f_\nu) \mid \nu = 1 : N_F\} = F_t[\{S_{m,n}(t_k) \mid k = 1 : N_S\}] \\ \text{ENDDO}(m)\end{array} \quad (3)$$

Since the cores are homogeneous, computations of equal complexity are expected to take similar times on each core. Thus, if the sets of temporal Fourier transforms are processed in parallel, i.e., if each core works on its own set of FFTs while all cores work in parallel, an overall complexity of $N_R N_S Log_2(N_S)$ per data plane will be achieved.

No Corner Turning. The primary innovation of our proposed scheme is the *avoidance* of corner turning. To achieve this capability, we exploit the linearity of the Fourier transform. Assume that the n-th data plane is being processed. After the temporal Fourier transforms stage, core C_i stores the N_R vectors of N_F complex frequency samples $\{\Phi_{m,n}(f_\nu) \mid \nu = 1 : N_F\}$, with $m = (i-1)N_R + 1$ to iN_R.

Conventionally, vectors □$_{m,n}$(:) would be sent to the corner turning memory, and properly stored within the matrix **Q**, from which new vectors would subsequently be retrieved for the spatial transforms.

In contradistinction, we proceed as follows. Recall that, at a given temporal frequency, complex samples from all hydrophones in an array column have to be (spatially) Fourier transformed. In order to avoid aliasing and enable easy incorporation of array design constraints, we embed the corresponding vectors of length N_V (nominally extracted from **Q**) into a spatial *template* vector (which we denote **q**). This vector has a dimension N_κ that is substantially larger than N_V. The template specifies the exact positions into which the contributions (i.e., complex frequency samples) from each hydrophone in a particular column must be stored. We exploit this spatial design and the linearity of the Fourier transform to develop the following parallel computational scheme.

Initially, the spatial template is provided to each core. More specifically, an array $\boldsymbol{\pi}^i$ is assigned to each core C_i such that, for $j = (i-1)N_R + \mu$, with $\mu = 1 : N_R$

$$q[\pi^i(\mu)] = \Phi(j)\,|_{\nu,n} \,. \tag{4}$$

All other positions in **q** for core *i* will be padded with zeros. In other words, when processing data plane *n* (i.e., the *n*-th column of the hydrophone array), we are embedding in each core only the complex frequency samples corresponding to the rows of data samples assigned to that core. Such assignments are done in parallel for all cores.

Overlapping Communications and Computation. The high-level architecture of the CELL processor was shown in Fig 1. As can be seen, each SPE core can communicate directly (using a DMA engine via the EIB) not only with all other cores, including the PPE, but also with a dual XDR storage and with external I/O. Within each core, notice a memory flow control unit (MFC). The MFC enables a user to send data from the core's local store (LS) to any destination of interest.

Hence, while computation is being performed in the SPU unit, data transfers can occur concomitantly. In our case, the computation of interest is a spatial Fourier transform of a vector **q** containing N_κ complex data. The resulting vector **p** will also be of length N_κ, and will be stored in LS. We need to carry out N_F such computations per data plane. Thus, while vector **q** for frequency ν is being processed in the SPU, MFC can dispatch vector **q** for the just-processed frequency ν-1 to the destination (e.g., the PPE), where the **q** vectors from all cores are combined (see below).

Note that it may be even more efficient to combine several such vectors in a single data packet before dispatch. How many depends on the speed of computation versus the latency and cost of data transfer.

Spatial Fourier Transforms. Let $\mathbf{q}^i(\nu)$ denote the spatial template vector for core C_i when processing temporal frequency ν. By definition,

$$\mathbf{q}(\nu) = \sum_{i=1}^{i=N_P} \mathbf{q}^i(\nu) \tag{5}$$

If $\mathbf{p}^i(\nu)$ represents the spatial Fourier transform of $\mathbf{q}^i(\nu)$, that is if

$$\mathbf{p}^i(\nu) = \mathrm{F}_y\{\mathbf{q}^i(\nu)\} \tag{6}$$

Then, by linearity of the Fourier transform,

$$\begin{aligned}\mathbf{p}(\nu) = \sum_{i=1}^{i=N_P} \mathbf{p}^i(\nu) = \sum_{i=1}^{i=N_P} \mathrm{F}_y\{\mathbf{q}^i(\nu)\} = \\ = \mathrm{F}_y\{\sum_{i=1}^{i=N_P} \mathbf{q}^i(\nu)\} = \mathrm{F}_y\{\mathbf{q}(\nu)\}\end{aligned} \tag{7}$$

Consequently, performing a distributed computation of the Fourier transform across all cores will produce the required result. We can now formulate the corresponding computational scheme. Note that in absence of a vectorized FFT, each core will loop over its assigned rows, as well as all temporal frequencies. Thus, for core C_i, we get

set : $\dim(\mathbf{q}^i) = \dim(\mathbf{p}^i) = N_\kappa > N_V$ (index *i* is implicit)
DO $\nu = 1 : N_F$
+ $\mathbf{q}^i = 0$
+ $q^i_\mu = q[\pi^i(\mu)] = \varphi^i(\mu)\,|_\nu = \Phi(j)\,|_{\nu,n}$ $\quad j = (i-1)N_R + \mu\,,\ \mu = 1 : N_R$
+ $\mathbf{p}^i = \mathrm{F}_y\{\mathbf{q}^i\}$
+ $\mathbf{p}^i(\nu)$ in LS(i)

ENDDO(ν)
+ **in parallel with computation in SPU(i), use MFC(i) to:**
% check availability of $\mathbf{p}^i(\nu)$ in LS(i)
% either send $\mathbf{p}^i(\nu)$ while $\mathbf{p}^i(\nu+1)$ is being computed in SPU(i)
% or concatenate: $\mathbf{p}^i(\nu)$, $\mathbf{p}^i(\nu+1)$,... in LS(i)
while computing: $\mathbf{p}^i(\nu+1)$, $\mathbf{p}^i(\nu+2)$,... in SPU(i)
prior to sending to PPE (while still computing) (8)

In the above formulation, the vector $\boldsymbol{\varphi}^i$ denotes data local to C_i. The correspondence with the frequency samples cube **Φ** has also been indicated. The vectors $\mathbf{p}^i$ are accumulated in the destination core (e.g., the PPE or an SPE on the CELL), combined according to Eq (7), and stored for further processing. We can now proceed with the actual implementation on a CELL computational platform.

5 Implementation and Timing Results

For developing highly complex, computationally-intensive programs, IBM recommends using the XLF compiler for multicore acceleration on Linux operated systems. It is part of the SDK 3.0 release, and provides, under Fedora 7, an advanced, high-performance cross-compilation capability that is tuned for the Cell Broadband Engine™ architecture. XLF fully supports the FORTRAN 95 standard, and also incorporates many features from the latest leading-edge FORTRAN 2003 standard. Thus, it provides not only unique array language capabilities, but also includes an optimizer capable of performing interprocedure analysis, loop optimizations, and *true* SIMD optimizations. Hence, the compiler back-end generates highly-optimized code for the PPU and SPU architectures, a condition *sine qua non* for achieving high computational performance. Since this compiler was only very recently released, the results reported herein refer to a software environment consisting of the Fedora Core 6 OS along with version 2.1 of IBM's CELL SDK, which included the IBM XLC C compiler enhanced for the CELL.

The written code is separated into PPE and SPE components. The PPE component consists of a PPE `main()` routine that performs initialization of the 3D data, spawns the SPE threads, and adds together the partial spatial FFT results generated by the SPEs. The SPE component consists of the SPE `main()` routine that

performs the temporal and spatial FFTs on the designed data strips, and copies the results back to main memory (MM). The SPE program makes use of three high-level functions: a load function to retrieve data from MM, a store function to copy data to MM, and the 1D FFT function.

Parameters for the code include the length of the temporal data samples (`tSIZE`), the number of SPEs to use in the processing (`nSPES`, limited by the number of available SPEs), the number of vertical elements per SPE (`vSIZE`), the number of horizontal elements (`hSIZE`), and the size of the padded vector used in performing the spatial FFT (`PADDED_vSIZE`).

For collecting our timing measurements, we have used the `gettimofday()` command, which is prototyped in `sys/time.h`. The command can be used to report time differences with microsecond accuracy. We have made use of an artificial looping procedure to gather better timing statistics. The latter number of iterations is labelled as `myTRIALS`.

The code was run using the parameters shown in Table 1. We have found that a single execution of the code with a data block given by `tSIZE` × `vSIZE` × `nSPES` × `hSIZE` is significantly faster than the time required for spawning the threads on the individual SPEs. Results are shown in Tables 2 and 3.

Number of data points per row	`tSIZE`	`2048`
Number of available SPEs	`nSPES`	`6`
Number of *m* values per SPE	`vSIZE`	`8`
Total number of *m* index values	`nSPES×vSIZE`	`48`
Padded size of *m* domain	`PADDED_vSIZE`	`256`
Number of *n* index values	`hSIZE`	`8`

Table 1. Parameters used for testing the distributed FFTs on the CELL.

`myTRIALS`	Total time	Per data block
100	`4.48 s`	`44.80 ms`
1000	`33.33 s`	`33.33 ms`
10000	`321.92 s`	`32.30 ms`

Table 2. Timing results. The number of trials denotes the effective number of data blocks processed within the stated time.

`myTRIALS`	Total time (s)	On the SPEs (percent)	On the PPEs (percent)
100	`4.51`	`2.71 (60)`	`1.80 (40)`
1000	`33.50`	`16.22 (48)`	`17.28 (52)`
10000	`322.72`	`150.48 (47)`	`172.23 (53)`

Table 3. Aggregate time spent on the SPEs and the PPE during execution of the FFTs sequence.

6 Conclusions

The results in Table 3 show that the PPE summation requires more time than the actual computation on the SPEs. In the future, we will use inter SPE communication procedures, in conjunction with a full implementation of the possible computation-communication overlap.

The *primary innovation* resulting from the present effort is the development of a *massively parallel computational scheme that avoids altogether the BF corner turning* stage. We presented details of this scheme in the context of the IBM CELL multicore processor. However, the underlying paradigm is general, and could readily be specialized to other multicore architectures. The algorithms were programmed in enhanced C for multicore processing. The recent release by IBM of the XLF high-performance compiler for multicore acceleration opens, via mixed language (C and FORTRAN) programming (using inter-language calls allowed by IBM's XLC and XLF compilers) an optimal framework for ultra fast future implementations. Such a framework would fully exploit the intrinsic array language, compiler optimization and numerical capabilities of FORTRAN in conjunction with the DMA and system capabilities of C.

Acknowledgments

This work was supported by the United States Office of Naval Research and by the Naval Sea Systems Command. Oak Ridge National Laboratory is managed for the US Department of Energy by UT-Battelle, LLC under contract DE-AC05-00OR22725.

References

[1] R. Trider, "A fast Fourier transform (FFT) based sonar signal processor", *IEEE Tran Acoustics, Speech, and Signal Processing*, **26**(1), 15-20 (1978).

[2] R. F. Follett and J. P. Donohoe, "A wideband, high-resolution, low probability of detection FFT beamformer", *IEEE Journal of Ocean Engineering*, **19**(2), 175-182 (1994).

[3] A. Chiang, S. Broadstone, and J. Gilbert, "Mul-tidimensional beamforming device", *US Patent* 6,111,816 (August 29, 2000).

[4] J. O. McMahon, "Space-time adaptive processing on the Mesh Synchronous Processor", *Lincoln Laboratory Journal*, **9**(2), 131-152 (1996).

[5] J. Meyer, "Multifunction radar systems for the dep-loyed warrior using VPX-REDI and RapidIO", *Military Embedded Systems* (Fall 2006).

[6] N. S. Sundar *et al*, "Hybrid algorithms for complete exchange in 2D meshes", *IEEE Transactions on Parallel and Distributed Systems*, **12**(12), 1201-1218 (2001).

[7] M. Mitchell and J. Oldham, "VSIPL++: parallel performance", unpublished report, CodeSourcery, LLC (May 2004).

[8] V. K. Prasana, "Energy efficient computation on FPGAs", *Journal of Supercomputing*, **32**(2), 139-162 (2005).

[9] J. A. Kahle *et al*, "Introduction to the Cell multi-processor", *IBM Journal of Research and Development*, **49**(4-5), 589-604 (2005).