Parallel Computing 35 (2009) 119–137

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Computing discrete transforms on the Cell Broadband Engine

David A. Bader *, Virat Agarwal, Seunghwa Kang

Georgia Institute of Technology, College of Computing, Atlanta, GA 30332, USA

ARTICLE INFO

Article history:
Received 17 June 2008
Received in revised form 14 October 2008
Accepted 16 December 2008
Available online 27 December 2008

Keywords:
Hardware accelerators
Parallel algorithms
Scientific computing

ABSTRACT

Discrete transforms are of primary importance and fundamental kernels in many computationally intensive scientific applications. In this paper, we investigate the performance of two such algorithms; Fast Fourier Transform (FFT) and Discrete Wavelet Transform (DWT), on the Sony–Toshiba–IBM Cell Broadband Engine (Cell/B.E.), a heterogeneous multicore chip architected for intensive gaming applications and high performance computing.

We design an efficient parallel implementation of Fast Fourier Transform (FFT) to fully exploit the architectural features of the Cell/B.E. Our FFT algorithm uses an iterative out-of-place approach and for 1K to 16K complex input samples outperforms all other parallel implementations of FFT on the Cell/B.E. including FFTW. Our FFT implementation obtains a single-precision performance of 18.6 GFLOP/s on the Cell/B.E., outperforming Intel Duo Core (Woodcrest) for inputs of greater than 2K samples. We also optimize Discrete Wavelet Transform (DWT) in the context of JPEG2000 for the Cell/B.E. DWT has an abundant parallelism, however, due to the low temporal locality of the algorithm, memory bandwidth becomes a significant bottleneck in achieving high performance. We introduce a novel data decomposition scheme to achieve highly efficient DMA data transfer and vectorization with low programming complexity. Also, we merge the multiple steps in the algorithm to reduce the bandwidth requirement. This leads to a significant enhancement in the scalability of the implementation. Our optimized implementation of DWT demonstrates 34 and 56 times speedup using one Cell/B.E. chip to the baseline code for the lossless and lossy transforms, respectively. We also provide the performance comparison with the AMD Barcelona (Quad-core Opteron) processor, and the Cell/B.E. excels the AMD Barcelona processor. This highlights the advantage of the Cell/B.E. over general purpose multicore processors in processing regular and bandwidth intensive scientific applications.

1. Introduction

The Cell Broadband Engine (or the Cell/B.E.) [20,12,13,31] is a novel high-performance architecture designed by Sony, Toshiba, and IBM (STI), primarily targeting multimedia and gaming applications. The Cell/B.E. is used in Sony's PlayStation 3 gaming console, Mercury Computer System's blade servers, and IBM's QS20, QS21, and QS22 Cell Blades. In this paper we design efficient parallel implementations of two discrete transform routines, fast Fourier transform (FFT) and discrete wavelet transform (DWT) on the Cell/B.E.

FFT is of primary importance and a fundamental kernel in many computationally intensive scientific applications such as computer tomography, data filtering, and fluid dynamics. Another important application area of FFT is in spectral analysis of

* Corresponding author. Tel.: +1 404 385 0004.
URL: http://www.cc.gatech.edu/~bader (D.A. Bader).

0167-8191/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2008.12.007

speech, sonar, radar, seismic and vibration detection. FFT is also used in digital filtering, signal decomposition, and in solver of partial differential equations. The performance of these applications rely heavily on the availability of a fast routine for computing forward and inverse Fourier transforms. The literature contains several publications related to FFTs on the Cell/B.E. processor. Williams *et al.* [37] analyze the Cell/B.E.'s peak performance for FFT of various types (1D, 2D), accuracy (single-, double-precision) and input sizes. Cico et al. [11] estimate the performance of 22.1 GFLOP/s for a single FFT that is resident in the local store of one SPE, or 176.8 GFLOP/s for computing 8 independent FFTs with 8K complex input samples. Note that all the other computation rates given in this paper – except for Cico et al. – consider the performance of a single FFT and include the off-loading overheads assuming that the source and the output of the FFT are both stored in main memory. In another work, Chow et al. [10] achieve 46.8 GFLOP/s for a large FFT (16 million complex samples) on the Cell/B.E. that is highly-specialized for this particular input size. FFTW on the Cell/B.E. [15] is a highly-portable FFT library of various types, precisions, and input sizes.

In our design of FFT routine for the Cell/B.E. (FFTC) we use an iterative out-of-place approach to solve 1D FFTs with 1K to 16K complex input samples. We describe our methodology to partition the work among the SPEs to efficiently parallelize a *single FFT computation* where the source and the output of the FFT are both stored in main memory. This differentiates our work from the prior literature and better represents the performance that one realistically sees in practice. The algorithm requires a synchronization among the SPEs after each stage of FFT computation. Especially, if input size is relatively small, the synchronization overhead becomes a significant bottleneck in performance. We introduce an optimized tree based barrier synchronization primitive for the Cell/B.E. Our synchronization barrier requires only $2 \log p$ stages (p: number of SPEs) of inter-SPE communication by using a tree-based approach. Also, our implementation achieves synchronization without any PPE intervention. This significantly improves the performance, as the PPE intervention not only results in a high communication latency but also leads to a sequentialization of the synchronization step. We achieve a performance improvement of over 4 as we vary the number of SPEs from 1 to 8. Considering that synchronization overhead becomes higher as the number of SPEs increases, the parallel speedup for relatively small input size demonstrates the efficiency of our barrier implementation. We obtain a performance of 18.6 GFLOP/s for a single-precision FFT with 8K complex input samples and also show significant speedup in comparison with various architectures. Our implementation is generic for this range of complex inputs.

DWT [7,36] is an important kernel in various application areas such as data analysis, numerical analysis, and image processing [28]. Also, JPEG2000 [18] employs DWT as its key algorithm, and prior analysis of JPEG2000 execution time [25,1,26] shows that DWT is one of the most computationally expensive algorithmic kernels. Previous parallelization strategies for the DWT [38,28,8] suggest the trade-offs between computation and communication overheads for the different underlying architectures. Lifting based DWT is also proposed in [33], by which, in-place DWT can be performed faster than the previous convolution based DWT [23]. Still, efficient implementation of DWT is challenging due to the distinct memory access patterns in horizontal and vertical filtering. In a naive implementation, poor cache behavior in a column-major traversal for vertical filtering becomes a performance bottleneck. Initially, a matrix transpose, which converts a column-major traversal to a row-major traversal, is adopted to improve the cache behavior. Column grouping, which interleaves the processing of adjacent columns, is introduced in [8] to improve cache behavior without a matrix transpose. For the Cell/B.E., Muta et al. optimize DWT in Motion JPEG2000 encoder using *pseudo-tile* approach [27]. However, their DWT implementation does not scale beyond a single SPE despite having high single SPE performance. This suggests that in order to achieve high performance, we need to take different approaches.

We adopt Jasper [1], a reference JPEG2000 implementation, as a baseline program and optimize lifting based forward DWT for the Cell/B.E. First, we introduce a novel data decomposition scheme to achieve highly efficient DMA data transfer and vectorization with low programming complexity. Then, we tune the column grouping strategy based on our data decomposition scheme to increase spatial locality and also design a new loop merging approach to increase temporal locality. In addition, we investigate the relative performance of the floating point operations and its fixed point approximation [1] on the Cell/B.E. Lastly, we fine tune the DMA data transfer to show how the Cell/B.E.'s software controlled data transfer mechanism can further improve the performance. These optimization approaches contribute to superior scalability in addition to high single SPE performance. Using one Cell/B.E. chip, our optimized code demonstrates 34 and 56 times speedup to the baseline code running on the PPE for the lossless and lossy DWT, respectively. Also, our Cell/B.E. optimized code runs 4.7 and 3.7 times faster than the code optimized for the AMD Barcelona (Quad-core Opteron) processor. Further, our implementation scales up to two Cell/B.E. chips in a single blade, especially for the lossy case.

This paper extends our previous FFT paper [5] and JPEG2000 paper [21]. We include the detailed explanation about FFT and DWT with the performance issues and the optimization approaches. We enhance the DWT performance from our previous conference paper by further tuning data transfer in the algorithm. This leads to additional 16 % and 7 % speedup for the lossless and lossy transform to the previous implementation. Also, we append the performance comparison with the AMD Barcelona (Quad-core Opteron) processor for DWT. We apply various optimization techniques using PGI C compiler for the Barcelona processor, which enables head to head comparison of the Cell/B.E. with the current general purpose multicore processor. The source code of our FFT and DWT implementations is freely available as open source from our CellBuzz project in SourceForge (http://sourceforge.net/projects/cellbuzz/).

This paper is organized as following. We first describe the fast Fourier and discrete wavelet transforms in Section 2. The novel architectural features of the Cell/B.E. are reviewed in Section 3. We then present our design to parallelize these transforms on the Cell/B.E. and optimize for the SPEs in Section 4. The performance analysis of our implementation is presented in Section 5, and we give conclusions in Section 6.

2. Fast Fourier and discrete wavelet transform

2.1. Fast Fourier transform

Fast Fourier transform (FFT) is an efficient algorithm that is used for computing discrete Fourier transform (DFT). The DFT of an m-element vector v is defined as,

$$F * v$$
$$\text{where}, F[j,k] = \omega^{j*k}$$
$$\text{and}, \omega = e^{2\pi i/m} = \cos(2\pi/m) + i * sin(2\pi/m)$$

The jth element of the FFT of a vector v can be represented by polynomial V,

$$(F * v)[j] = V(\omega^j)$$
$$\text{where}, V(x) = \Sigma_{k=0}^{m-1} x^k * v(k)$$

Using a divide-and-conquer approach V is reduced into V_{odd} and V_{even}.

$$V(x) = V_{even}(x^2) + x * V_{odd}(x^2)$$

These polynomials are evaluated at points $(\omega^j)^2$, which essentially are $m/2$ different points, as $(\omega^{j+m/2})^2 = (\omega^j)^2$. This reduces the original problem (V) at m-points to two half-size problems (V_{even}, V_{odd}) at $m/2$ points. Extending from these definitions, Algorithm 1 gives the pseudo-code of a naive Cooley-Tukey radix-2 Decimate in Frequency (DIF) algorithm. The algorithm runs in $\log N$ stages and each stage requires $O(N)$ computation, where N is the input size. This algorithm uses the out-of-place approach, where at each stage two complex-element arrays (a&b) are used for computation, one input and one output that are swapped at every stage. The array ω_v contains the *twiddle factors* required for FFT computation. At each stage the

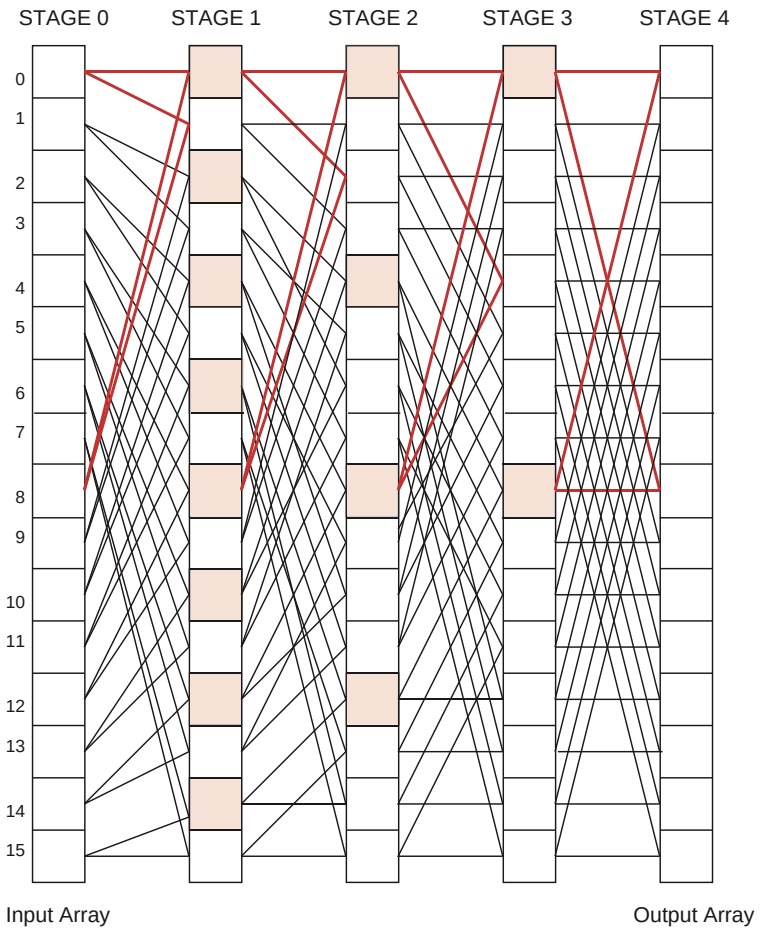

Fig. 1. Butterflies of the ordered DIF FFT algorithm.

computed complex samples are stored at their respective locations thus saving a bit-reversal stage for output data. This is an iterative algorithm which runs until the parameter *problemSize* reduces to 1.

Fig. 1 shows the butterfly stages of this algorithm for an input of 16 sample points. The FFT computation for this input size runs into 4 stages. The input in this figure is the left-most array and the output of the FFT algorithm is the array on the right. At every stage the lines represent the input elements used and location where the computed elements are stored in the output array. The computation involved is given in steps 14 and 15 of Algorithm 1. The colored boxes indicate that the offset between the output elements doubles with every stage.

Algorithm 1: Sequential FFT algorithm.

Input: array $A[0]$ of size N

```
1  NP ⟵ 1 ;
2  problemSize ⟵ N;
3  dist ⟵ 1;
4  i1 ⟵ 0;
5  i2 ⟵ 1;
6  while problemSize > 1 do
7  |  Begin Stage;
8  |  a ⟵ A[i1];
9  |  b ⟵ A[i2];
10 |  k = 0, jtwiddle = 0;
11 |  for j ← 0 to N − 1 step 2 ∗ NP do
12 |  |  W ⟵ ω_v[jtwiddle];
13 |  |  for jfirst ← 0 to NP do
14 |  |  |  b[j + jfirst] ← a[k + jfirst] + a[k + jfirst + N/2];
15 |  |  |  b[j + jfirst + Dist] ← (a[k + jfirst] − a[k + jfirst + N/2]) ∗ W;
16 |  |  k ← k + NP;
17 |  |  jtwiddle ← jtwiddle + NP;
18 |  swap(i1, i2);
19 |  NP ← NP ∗ 2;
20 |  problemSize ← problemSize/2;
21 |  dist ← dist ∗ 2;
22 |  End Stage;
```

Output: array $A[i1]$ of size N

There are several algorithmic variants of the FFTs that have been well studied for parallel processors and vector architectures [2–4,6].

Apart from the theoretical complexity, another common performance metric used for the FFT algorithm is the floating point operation (FLOP) count. On analyzing the sequential algorithm, we see that during each iteration of the innermost *for* loop there is one complex addition for the computation of first output sample, which accounts for two FLOPs. The second output sample requires one complex subtraction and multiplication that account for eight FLOPs. Thus, for the computation of two output samples during each innermost iteration we require 10 FLOPs, which suggests that we require five FLOPs for the computation of a complex sample at each stage. The total computations in all stages are $N \log N$ which makes the total FLOP count for the algorithm as $5N \log N$. The canonical FLOP count is used in most performance studies of the FFT, such as the benchFFT [15].

2.2. Discrete wavelet transform in JPEG2000

Fourier transform translates the time domain representation of a signal to the frequency domain representation. However, in the translation, Fourier transform does not preserve the signal's time varying nature. Or, in terms of image processing, Fourier transform does not preserve spatial information of the image. Wavelet transform, in contrast, preserves spatial information in the transform, and this enhances the image quality especially for the low bit rate representation. Wavelet transform, in general, represents a signal based on the basis wavelets $\psi_{a,b}(t)$ created by scaling and shifting a single function $\psi(t)$, called mother wavelet. In mathematical form, $\psi_{a,b}(t) = \frac{1}{\sqrt{a}}\psi(\frac{t-b}{a})$ represents the relationship between a mother wavelet and the basis wavelets. The narrower wavelets capture the detail of a signal while the wider wavelets contribute in building

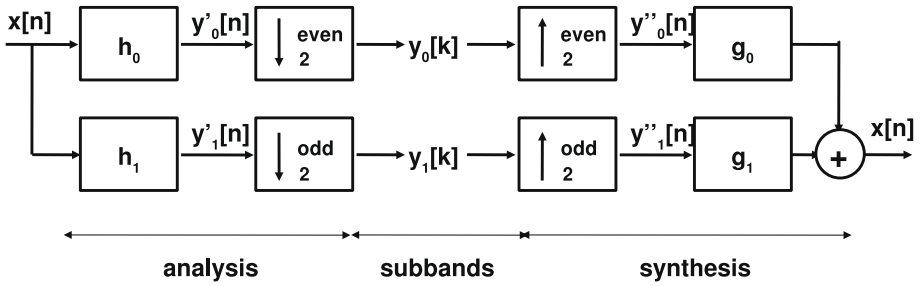

Fig. 2. 1-D, two channel subband transform [35].

coarse representation of a signal. The wavelets with different shift amounts capture the time (or spatial) information of a signal.

JPEG2000 adopts discrete wavelet transform (DWT) and applies the algorithm to an entire tile (note that a single image can be composed of a single tile or multiple tiles) in contrast to applying discrete cosine transform (DCT) to independent small blocks as the case of JPEG. Taubman and Marcellin well explains the DWT in the context of JPEG2000 in [35]. The DWT in JPEG2000 applies 1-D DWT for every line in both vertical and horizontal directions to achieve 2-D decomposition of an image. In the 1-D DWT, an input sequence $x[n]$ is filtered through the low pass and high pass filters with impulse response $h_0[n]$ and $h_1[n]$. This filtering generates two subband data for the coarse and detail part of the input. Then, the low pass and high pass filter outputs are subsampled by the factor of 2. After subsampling, the input sequence $x[n]$ and output sequence $y[n]$ (constructed by interleaving two filter outputs $y_0[k]$ and $y_1[k]$) have same number of samples. There exist filters $h_0[n]$ and $h_1[n]$ and their inverse filters ($g_0[n]$ and $g_1[n]$) that enable perfect reconstruction. This feature enables lossless

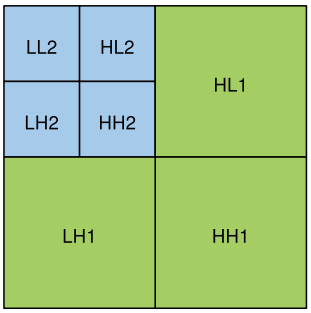

Fig. 3. The DWT output for a 2-D image array with two levels.

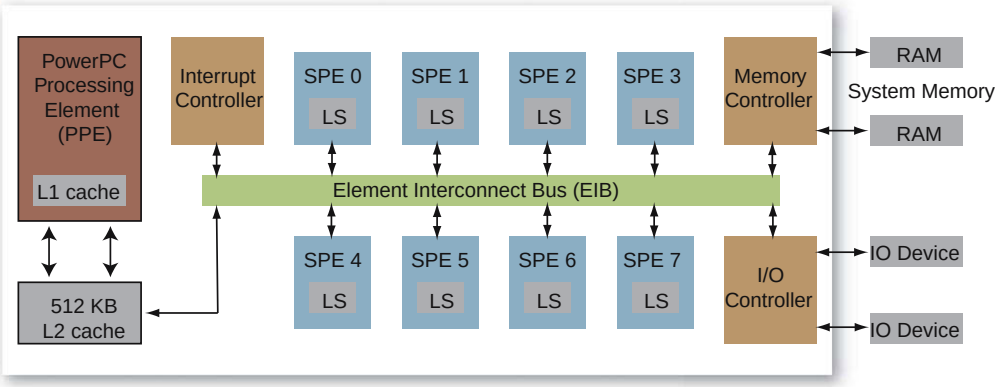

Fig. 4. Cell Broadband Engine architecture.

transform. Fig. 2 summarizes the 1-D DWT in JPEG2000. By applying 1-D DWT in both horizontal and vertical direction, DWT generates four subbands (LL, HL, LH and HH). The LL subband represents the low pass part in both vertical and horizontal directions and the HH subband represents the high pass part in both directions. The HL and LH subbands represent the mix of the low and high pass part in vertical and horizontal directions. This process can be repeated for multiple levels by recursively applying DWT to the LL subband of the previous level as portrayed in Fig. 3.

Basically, the low pass or high pass filtering in the 1-D DWT is a convolution of two sequences $y[n] = \Sigma_i x[i] \times h_{n \bmod 2}[n-i]$ ([34]), and DWT is initially implemented using convolution based approach [23]. Later, Sweldens [32] introduces lifting based approach. Lifting based approach requires (maximum factor of two times) fewer operations and runs faster than convolution based DWT. Sweldens describes the algorithm as following. Lifting based approach consists of multiple steps: splitting, predicting, and updating steps. Splitting step splits an input sequence to two subsequences with high correlation. This is achieved by constructing two subsequences using even and odd elements of the input sequence. Then, the second subsequence is predicted based on the first sequence, and only the differences between real and predicted subsequences are recorded. In the last updating step, the first subsequence is updated to maintain certain scalar quantity (e.g. the average of the sequence). Lifting based approach requires one splitting step and two lifting steps (one predict and one update step) for the lossless mode and one splitting step and four lifting steps (two predict and two update steps) with one optional scaling step for the lossy mode [35].

3. Cell broadband engine architecture

The Cell Broadband Engine (Cell/B.E.) processor is a heterogeneous multi-core chip that is significantly different from conventional multiprocessor or multi-core architectures. The Cell/B.E. consists of a traditional microprocessor (called the PPE), eight SIMD co-processing units named as synergistic processor elements (SPEs), a high speed memory controller, and a high bandwidth bus interface (termed the element interconnect bus, or EIB), all integrated on a single chip. Fig. 4 gives an architectural overview of the Cell/B.E. processor. We refer the reader to [29,14,22,16,9] for additional details.

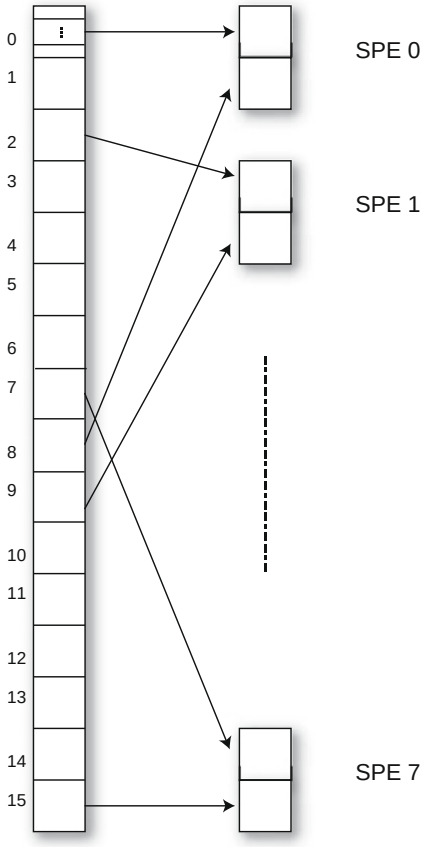

Fig. 5. Partition of the input array among the SPEs (e.g. eight SPEs in this illustration).

The PPE runs the operating system and coordinates the SPEs. It is a 64-bit PowerPC core with a vector multimedia extension (VMX) unit, 32 KB L1 instruction and data caches, and a 512 KB L2 cache. The PPE is a dual issue, in-order execution design, with two way simultaneous multithreading.

Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPEs and the PPE. The SPE is a micro-architecture designed for high performance data streaming and data intensive computations. It includes a 256 KB *local store* (LS) memory to hold SPE program's instructions and data. The SPE cannot access main memory directly, but it can issue DMA commands to the MFC, with 16 outstanding DMAs per SPE, to bring data into the local store or write computation results back to main memory. DMA is non-blocking so that the SPE can continue program execution while DMA transactions are performed. Peak DMA performance is achievable when both the effective address and the local store address are 128 bytes aligned and the transfer size is an even multiple of 128 bytes.

The SPE is an in-order dual-issue statically scheduled architecture. Two SIMD [19] instructions can be issued per cycle for even and odd pipelines. The SPE does not support dynamic branch prediction, but instead relies on user-provided or compiler-generated branch hints using *prepare-to-branch* instructions to redirect instruction prefetch to branch targets. Thus branches should be minimized on the SPE as far as possible.

With a clock speed of 3.2 GHz, the Cell processor has a theoretical peak performance of 204.8 GFLOP/s (single-precision). The EIB supports a peak bandwidth of 204.8 GB/s for intra-chip transfers among the PPE, the SPEs, and the memory and I/O interface controllers. The memory interface controller (MIC) provides a peak bandwidth of 25.6 GB/s to main memory. The I/O controller provides peak bandwidths of 25 GB/s inbound and 35 GB/s outbound.

IBM has recently announced the PowerXCell 8i second-generation Cell/B.E. processor available in the IBM QS22 blade that performs native double-precision floating point operations at over 100 GFLOP/s [17].

4. Cell/B.E.-optimized algorithms for fast Fourier and discrete wavelet transform

4.1. FFT algorithm for the Cell/B.E. processor

There are several architectural features that make it difficult to parallelize and optimize the Cooley-Tukey FFT algorithm on the Cell/B.E. The algorithm is branchy due to the presence of a doubly nested *for* loop within the outer *while* loop. This results in a compromise on the performance due to the absence of a dynamic branch predictor on the Cell/B.E. The algorithm requires an array that consists of the $N/2$ complex twiddle factors. Since each SPE has a limited local store of 256 KB, this array cannot be stored entirely on the SPEs for a large input size. The limit in the size of the local store memory also restricts the maximum input data that can be transferred to the SPEs.

Parallelization of a single FFT computation involves synchronization between the SPEs after every stage of the algorithm, as the input data of a stage is the output data of the previous stage. To achieve high performance it is necessary to divide the work equally among the SPEs so that no SPE waits at the synchronization barrier. Also, the algorithm requires $\log N$ synchronization stages which impacts the performance.

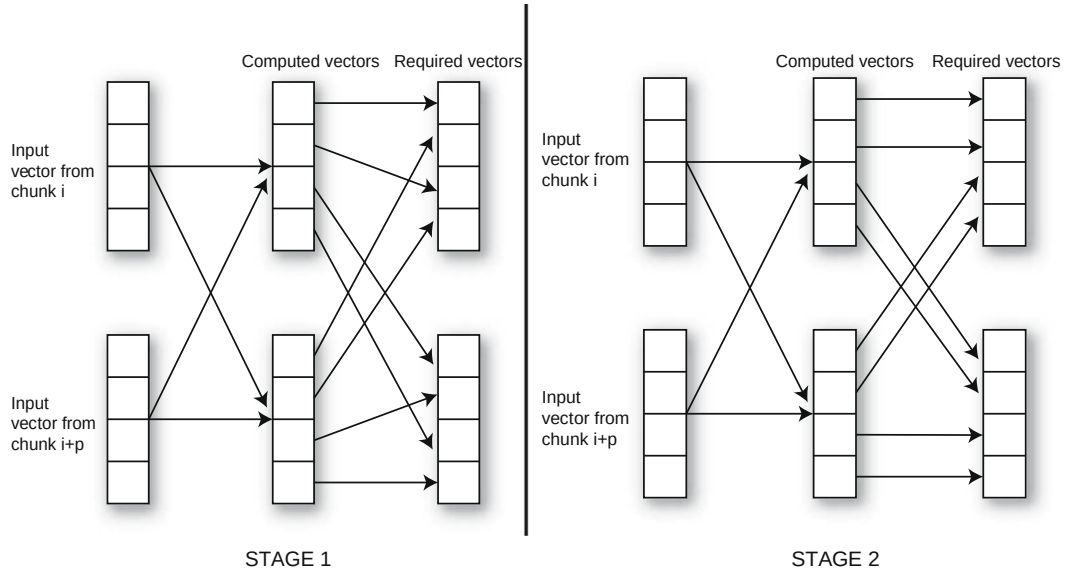

Fig. 6. Vectorization of the first two stages of the FFT algorithm. These stages require a shuffle operation over the output vector to generate the desired output.

It is difficult to vectorize every stage of the FFT computation. For vectorization of the first two stages of the FFT computation it is necessary to shuffle the output data vector, which is not an efficient operation in the SPE instruction set architecture. Also, the computationally intensive loops in the algorithm need to be unrolled for the best pipeline utilization. This becomes a challenge given a limited local store space on the SPEs.

4.1.1. Parallelizing FFT for the Cell/B.E.

As mentioned in the previous section, for the best performance it is important to partition work among the SPEs to achieve load balancing. We parallelize by dividing the input array held in main memory into $2p$ chunks, each of size $\frac{N}{2p}$, where p is the number of SPEs.

During every stage, SPE i is allocated chunk i and $i+p$ from the input array. The basis for choosing these chunks for an SPE lies in the fact that these chunks are placed at an offset of $N/2$ input elements. For the computation of an output complex sample we need to perform complex arithmetic operation between input elements that are separated by this offset. Fig. 5 gives an illustration of this approach for work partitioning among 8 SPEs.

The PPE does not intervene in the FFT computation after this initial work allocation. After spawning the SPE threads it waits for the SPEs to finish execution.

4.1.2. Optimizing FFT for the SPEs

After dividing the input array among the SPEs, each SPE is allocated two chunks each of size $\frac{N}{2p}$. Each SPE, fetches this chunk from main memory using DMA transfers and uses double-buffering to overlap memory transfers with computation. Within each SPE, after computation of each buffer, the computed buffer is written back into main memory at the correct offset using DMA transfers.

The detailed pseudo-code is given in Algorithm 2. The first two stages of the FFT algorithm are duplicated, that correspond to the first two iterations of the outer *while* loop in the sequential algorithm. This is necessary as the vectorization of these stages requires a shuffle operation (*spu_shuffle()*) over the output to re-arrange the output elements to their correct locations. Please refer to Fig. 6 for an illustration of this technique for stages 1 and 2 of the FFT computation.

The innermost *for* loop (in the sequential algorithm) can be easily vectorized for $NP > 4$, that correspond to the stages 3 through $\log N$. However, it is important to duplicate the outer *while* loop to handle stages where $NP <$ *buffersize*, and otherwise. The global parameter *buffersize* is the size of a single DMA get buffer. This duplication is required as we need to stall for a DMA transfer to complete, at different places within the loop for these two cases. We also unroll the loops to achieve better pipeline utilization. This significantly increases the size of the code thus limiting the unrolling factor.

For relatively small inputs and as the number of SPEs increases, the synchronization cost becomes a significant issue since the time per stage decreases but the cost per synchronization increases. The SPEs are synchronized after each stage, using *inter-SPE communication*. This is achieved by constructing a binary synchronization tree, so that synchronization is achieved in $2\log p$ stages. The synchronization involves the use of inter-SPE mailbox communication without any intervention from the PPE. Please refer to Fig. 7 for an illustration of the technique.

This technique performs significantly better than other synchronization techniques that either use chain-like inter-SPE communication or require the PPE to synchronize between the SPEs. The chain-like technique requires $2p$ stages of inter-SPE communication whereas with the intervention of the PPE latency of communication reduces the performance of this barrier. With instruction level profiling we determine that the time required per synchronization stage using our tree-synchronization barrier is about 1 microsecond (3200 clock cycles) as opposed to 20 microseconds in the PPE-coordinated synchronization.

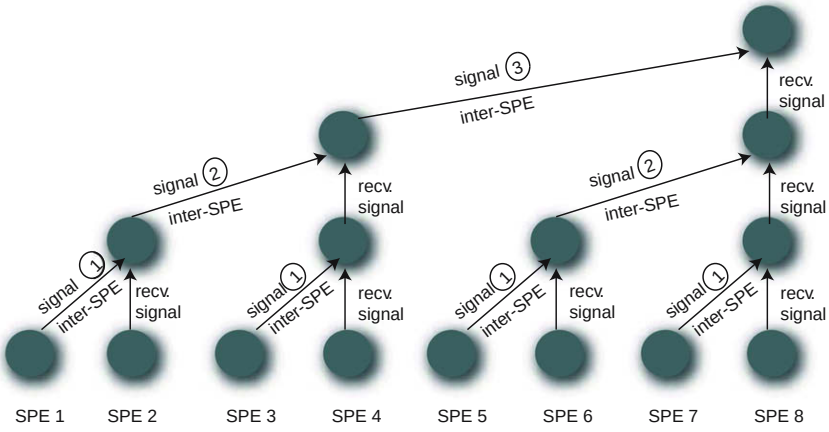

Fig. 7. Stages of the synchronization barrier using inter-SPE communication. The synchronization involves sending inter-SPE mailbox messages up to the root of the tree and then sending back acknowledgment messages down to the leaves in the same topology.

Algorithm 2: Parallel FFTC algorithm: view within SPE.

Input: array in PPE of size N
Output: array in PPE of size N

```
1  NP ⟵ 1 ;
2  problemSize ⟵ N;
3  dist ⟵ 1;
4  fetchAddr ⟵ PPE input array;
5  putAddr ⟵ PPE output array;
6  chunkSize ⟵ N/(2*p);
7  Stage 0 (SIMDization achieved with shuffling of output vector);
8  Stage 1 ;
9  while NP < buffersize && problemSize > 1 do
10 |  Begin Stage;
11 |  Initiate all DMA transfers to get data;
12 |  Initialize variables;
13 |  for j ← 0 to 2 * chunkSize do
14 |  |  Stall for DMA buffer;
15 |  |  for i ← 0 to buffersize/NP do
16 |  |  |  for jfirst ← 0 to NP do
17 |  |  |  |  SIMDize computation as NP > 4;
18 |  |  |  Update j, k, jtwiddle;
19 |  |  Initiate DMA put for the computed results
20 |  swap(fetchAddr, putAddr);
21 |  NP ← NP * 2;
22 |  problemSize ← problemSize/2;
23 |  dist ← dist * 2;
24 |  End Stage;
25 |  Synchronize using Inter-SPE communication;
26 while problemSize > 1 do
27 |  Begin Stage;
28 |  Initiate all DMA transfers to get data;
29 |  Initialize variables;
30 |  for k ← 0 to chunkSize do
31 |  |  for jfirst ← 0 to min(NP, chunkSize − k) step buffersize do
32 |  |  |  Stall for DMA buffer;
33 |  |  |  for i ← 0 to buffersize do
34 |  |  |  |  SIMDize computation as buffersize > 4;
35 |  |  |  Initiate DMA put for the computed results;
36 |  |  Update j, k, jtwiddle;
37 |  swap(fetchAddr, putAddr);
38 |  NP ← NP * 2;
39 |  problemSize ← problemSize/2;
40 |  dist ← dist * 2;
41 |  End Stage;
42 |  Synchronize using Inter-SPE communication;
```

4.2. Cell/B.E. optimized DWT algorithm

4.2.1. Data decomposition scheme

Data layout and partitioning is an important design issue for the Cell/B.E. due to the alignment and size requirements for DMA data transfer and SIMD load/store. Our data decomposition scheme, shown in Fig. 8, satisfies the above requirements for the two dimensional array with an arbitrary width and height, assuming that every row can be arbitrarily partitioned into multiple chunks for independent processing. First, we pad every row to force the start address of every row to be cache line

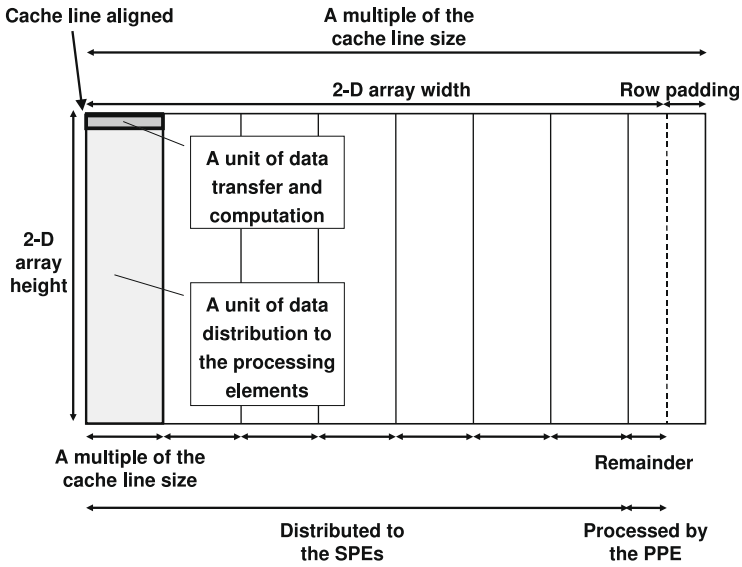

Fig. 8. Data decomposition scheme for two dimensional array.

aligned. Then, we partition the data array to multiple chunks, and every chunk except for the last has the width a multiple of the cache line size. These data chunks become a unit of data distribution to the PPE and the SPEs. The constant width chunks are distributed to the SPEs and the PPE processes the last remainder chunk. The SPE traverses the assigned chunks by processing every single row in the chunk as a unit of data transfer and computation.

This data decomposition scheme has several impacts on the performance and the programmability. First, every DMA transfer in the SPE becomes cache line aligned, and the transfer size becomes a multiple of the cache line size. This results in efficient DMA transfers and reduced programming complexity. Reduced programming complexity, or in other words, shorter code size also saves the local store space, and this is important for the Cell/B.E. since the local store size (256 KB) is relatively small. The remainder chunk with an arbitrary width is processed by the PPE to enhance the overall chip utilization. Our data decomposition scheme also satisfies the alignment requirement for SIMD load/store.

Second, the local store space requirement becomes constant independent of the data array size. As mentioned above, a single row in the chunk, which has a constant width, becomes a unit of data transfer and computation in the SPE. This leads to a constant memory requirement, and we can easily adopt the optimization techniques that require additional local store space. For example, double buffering or multi-level buffering (Fig. 9) is an efficient technique for hiding latency but increases

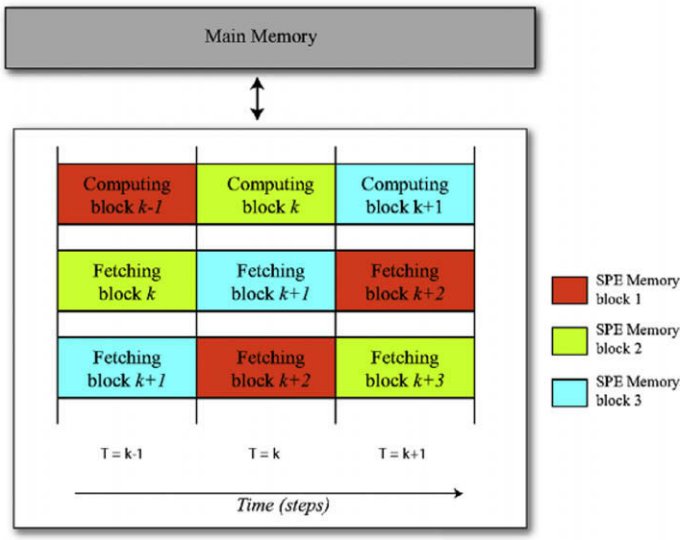

Fig. 9. Illustration of multi-level buffering within a single SPE.

the local store space requirement at the same time. However, owing to the constant memory requirement in our data decomposition scheme, we can increase the level of buffering to a higher value that fits within the local store.

In addition, fixed data size leads to a constant loop count, which enables the compiler to better predict the program's runtime behavior. This helps the compiler to use optimization techniques such as loop unrolling, instruction rescheduling, and compile time branch prediction. Compile time instruction rescheduling and branch prediction compensate for the lack of runtime out-of-order execution and dynamic branch prediction support in the SPE.

4.2.2. Parallelization and vectorization of DWT based on the data decomposition scheme

For the horizontal filtering, we assign an identical number of rows to each SPE, and a single row becomes a unit of data transfer and computation. We parallelize and vectorize vertical filtering based on our data decomposition scheme, and parallelization and vectorization of vertical filtering are straightforward based on our data decomposition scheme. Still, there are additional performance issues need to be discussed related to the bandwidth requirement and the real number representation. This will be described in the remainder of this section (see Fig. 10).

4.2.3. Loop merging algorithm

Algorithm 3: Pseudo code for original implementation.

Input: an image data array $array\_data$, number of rows $number\_of\_rows$, row width including padding $stride$
Output: an image data array $array\_data$

$high\_start \leftarrow number\_of\_rows/2$;

/* 1st lifting step for the vertical filtering */

$p\_low \leftarrow array\_data[0]$;
$p\_high \leftarrow array\_data[high\_start * stride]$;

$n \leftarrow high\_start - 1$;
for $i \leftarrow 0$ ***to*** $n - 1$ **do**
  $*p\_high \leftarrow *p\_high - ((*p\_low + *(p\_low + stride))/2)$;
  $p\_low \leftarrow p\_low + stride$;
  $p\_high \leftarrow p\_high + stride$;

$*p\_high \leftarrow *p\_high - *p\_low$;

/* 2nd lifting step for the vertical filtering */

$p\_low \leftarrow array\_data[0]$;
$p\_high \leftarrow array\_data[high\_start * stride]$;

$*p\_low \leftarrow *p\_low + ((*p\_high + 1)/2)$;
$p\_low \leftarrow p\_low + stride$;

$n \leftarrow high\_start - 1$;
for $i \leftarrow 0$ ***to*** $n - 1$ **do**
  $*p\_low \leftarrow *p\_low + ((*p\_high + *(p\_high + stride) + 2)/4)$;
  $p\_low \leftarrow p\_low + stride$;
  $p\_high \leftarrow p\_high + stride$;

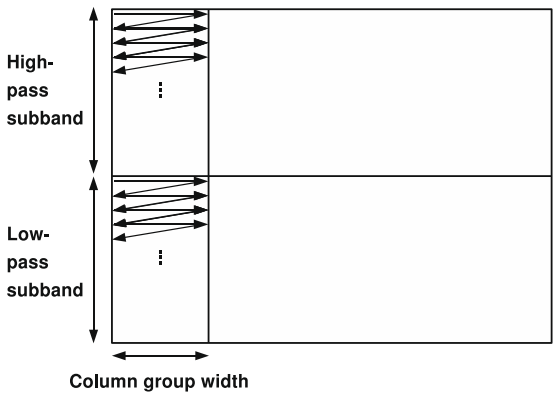

Fig. 10. Illustration of column-major traversal with column grouping.

Algorithm 4: Pseudo code for our merged implementation.

Input: an image data array $array\_data$, number of rows $number\_of\_rows$, row width including padding $stride$
Output: an image data array $array\_data$

$high\_start \leftarrow number\_of\_rows/2$;

/* *merges the 1st and 2nd lifting steps for the vertical filtering* */

$p\_low \leftarrow array\_data[0]$;
$p\_high \leftarrow array\_data[high\_start * stride]$;

$*p\_high = *p\_high - ((*p\_low + *(p\_low + stride))/2)$;
$*p\_low = *p\_low + ((*p\_high + 1)/2)$;
$p\_low \leftarrow p\_low + stride$;
$p\_high \leftarrow p\_high + stride$;

$n \leftarrow high\_start - 2$;
for $i \leftarrow 0$ ***to*** $n - 1$ **do**
  $*p\_high \leftarrow *p\_high - ((*p\_low + *(p\_low + stride))/2)$;
  $*p\_low \leftarrow *p\_low + ((*(p\_high - stride) + *p\_high + 2)/4)$;
  $p\_low \leftarrow p\_low + stride$;
  $p\_high \leftarrow p\_high + stride$;

$*p\_high \leftarrow *p\_high - *p\_low$;
$*p\_low \leftarrow *p\_low + ((*(p\_high - stride) + *p\_high + 2)/4)$;

The DWT algorithm consists of multiple steps: one splitting step and two lifting steps in the lossless mode and one splitting step, four lifting steps, and one optional scaling step in the lossy mode. Considering that the entire column group data for a large image does not fit into the local store, 3 or 6 steps in the vertical filtering involve 3 or 6 DMA data transfers of the entire column group data. As the number of SPEs increases, the limited off-chip memory bandwidth becomes a bottleneck. To reduce the amount of DMA data transfer, we merge the lifting steps. Algorithm 3 illustrates the original algorithm for the lossless mode assuming that the number of rows is even and larger than four. Algorithm 4 depicts our merged algorithm. The splitting step can also be merged with the next two lifting steps. Fig. 11 illustrates the splitting step and the merged lifting step in the lossless vertical filtering. In the splitting step, the even and odd elements of the input array are split to two subsequences with high correlation for the following lifting steps (prediction and updating). This splitting step involves the DMA data transfer of the whole column group data. If we adjust the pointer addresses for the two subsequences (to compute low and high pass subbands) and increase the increment size in the following merged lifting step, the splitting step can be merged with the merged lifting step. This merges three steps in the lossless mode to a single step and reduces the amount of data transfer. However, updating the high frequency part in the merged single step will overwrite the input data before it is read, and this leaves us with a problem. In Fig. 11, *high0** and *low0** are updated first in the merged lifting step, and *high1** and *low1** are updated next. If we adjust the input data pointer and skip the splitting step, updating *high0** overwrites *low2* in the input data array before it is read. To remedy this problem, we use an auxiliary buffer (in main memory), and the updated high pass data are written to the buffer first and copied to the original data array later. As this buffer needs to hold a upper half of the column group data, it cannot be placed in the local store. Still, the amount of data transfer related to the auxiliary buffer is half of the entire column group, and this halves the amount of data transfer for the splitting step. We find that a similar idea is published in [24] for the lossy case. Loop fusion for the four lifting steps is described in the paper. By combining this idea with our approach, we merge one splitting step, four lifting steps, and the optional scaling step in the lossy mode into a single loop to further reduce the DMA bandwidth requirement.

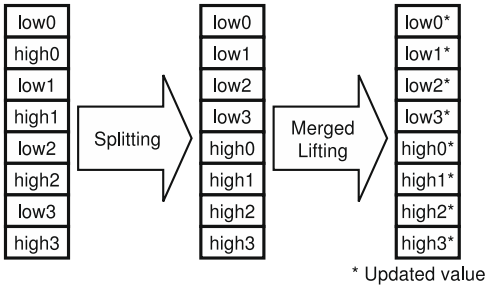

Fig. 11. The splitting step and the merged lifting step for the vertical filtering.

Table 1
Latency for the SPE instructions.

Instruction	Description	Latency
mpyh	Two byte integer multiply high	Seven cycles
mpyu	Two byte integer multiply unsigned	Seven cycles
a	Add word	Two cycles
fm	Single-precision floating point multiply	Six cycles

The Cell/B.E.'s software controlled DMA data transfer mechanism gives an opportunity for further optimization. We note that the data transfer from the auxiliary buffer to the original data array can be started before the merged lifting step is finished. For the above example, we can update $high0^*$ in the final output after *low2* is read. This, in combination with the Cell/B.E.'s data transfer mechanism, enables the overlap of the last data copy with the merged lifting step. This fine tuning of data transfer based on the application specific requirements leads to additional 16% and 7% speedup for the lossless and lossy case to the already highly optimized implementation.

4.2.4. Real number representation and its performance impact on the Cell/B.E.

In [1], Adams and Kossentini suggest the fixed point representation for the real numbers in the JPEG2000 lossy encoding process to enhance the performance and the portability. The authors assume that fixed point instructions are generally faster than floating point instructions. However, the current Cell/B.E. processor is optimized for (single-precision) floating point operations, and the floating point instructions have comparable speed to the fixed point instructions. Moreover, the SPE instruction set architecture does not support four-byte integer multiplication; thus four-byte integer multiplication needs to be emulated by two-byte integer multiplications and additions. Table 1 summarizes the latency for the instructions used in the integer and floating point multiplications. Therefore, replacing floating point arithmetic with fixed point arithmetic loses its benefit on the Cell/B.E. in both performance and programmability. Jasper adopts the fixed point approximation, and we re-implement the routine to use floating point operations before vectorization.

5. Performance analysis

In this section we present the performance analysis of our Cell/B.E. optimized FFT and DWT implementations. In each of the sections below we give details on the compilers used, platform description and optimization levels. We also present extensive performance comparisons with other optimized implementations and leading architectures.

5.1. Cell/B.E. optimized FFT

For compiling, instruction level profiling, and performance analysis we use the IBM Cell Broadband Engine SDK 3.0.0-1.0, gcc 4.1.1 with level 3 optimization. From '*/proc/cpuinfo*' we determine the clock frequency as 3.2 GHz with revision 5.0. We use *gettimeofday()* on the PPE before computation on the SPE starts, and after it finishes. For profiling measurements we iterate the computation 10000 times to eliminate the noise of the timer. For parallelizing a single 1D FFT on the Cell/B.E., it is important to divide the work among the SPEs. Fig. 12 shows the performance of our algorithm with varying the number of

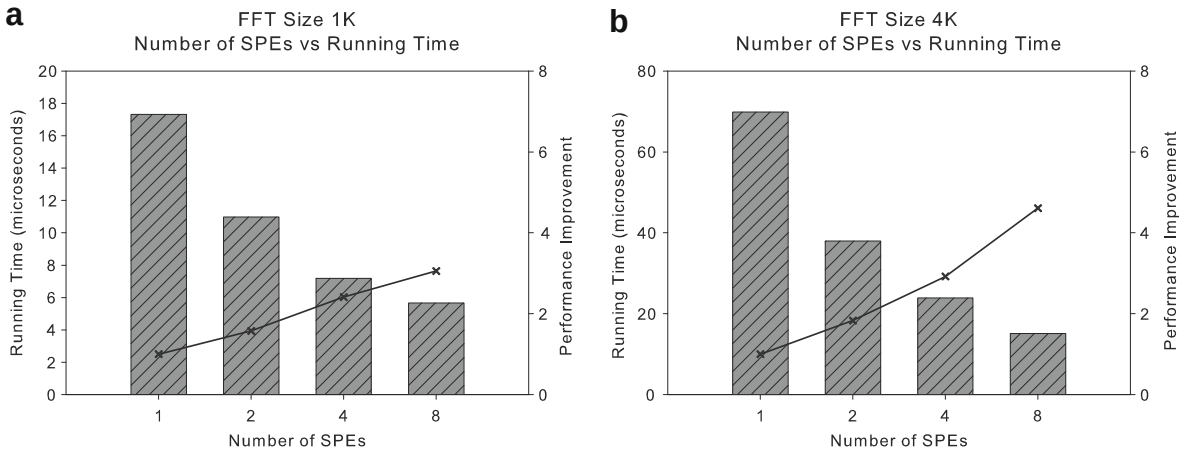

Fig. 12. Running time of our FFTC code on 1K and 4K inputs as we increase the number of SPEs.

SPEs for 1K and 4K complex input samples. The performance scales well with the number of SPEs which suggests that load is balanced among the SPEs.

We focus on FFTs that have from 1K to 16K complex input samples. Fig. 13 shows the single-precision performance for complex inputs of FFTC, our optimized FFT, as compared with the following architectures:

- *IBM Power 5:* IBM OpenPower 720, Two dual-core 1.65 GHz POWER5 processors.
- *AMD Opteron:* 2.2 GHz Dual Core AMD Opteron Processor 275.
- *Intel Duo Core:* 3.0 GHz Intel Xeon Core Duo (Woodcrest), 4MB L2 cache.
- *Intel Pentium 4:* Four-processor 3.06 GHz Intel Pentium 4, 512 KB L2.

We use the performance numbers from benchFFT [15] for the comparison with the above architectures. We consider the FFT implementation that gives best performance on these architectures for comparison.

The Cell/B.E. has a two instruction pipelines, and for achieving high performance it is important to optimize the code so that the processor can issue two instructions per clock cycle. This level of optimization requires inspecting the assembly dump of the SPE code. We use the *IBM Assembly Visualizer for the Cell/B.E.* tool to analyze this optimization. The tool highlights the stalls in the instruction pipelines and helps the user to reorganize the code execution while maintaining correctness. In many cases the tool helped us analyze the stalls and dependencies, that helped us reorder instructions and increase pipeline utilization. It also helped in determining the level of unrolling for the loops to fully utilize the available registers on the machine.

5.2. Cell/B.E. optimized DWT

We use *gcc* compiler with *–O5* optimization flag for the performance analysis. We use *gcc* instead of *xlc* as *gcc* produces the executable with smaller code size. Based on our data decomposition scheme, the output code is highly static and *gcc* provides comparable performance to *xlc* with smaller code size. As we optimize the DWT routine in the context of JPEG2000, the SPE program image includes all the optimized routines for JPEG2000 encoding. Thus, the SPE program image compiled with *xlc*, which attempts more aggressive optimization, leaves only a small fraction of the local store for dynamic memory allocation. Our baseline code is the open source Jasper 1.900.1, and the test file is a 28.3 MB 3800 × 2600 color image (http://www.jpeg.org/jpeg2000guide/testimages/WalthamWatch/Original/waltham_dial.bmp). The default option and *-O mode=real -O rate=0.1* are applied for lossless and lossy encoding, respectively. Our experiments use real hardware, the IBM QS20 Cell blade server with dual Cell/B.E. 3.2 GHz chips (rev. 3) and 1 GB main memory. We use *gettimeofday()* function for timing and do not iterate the experiments multiple times owing to the low variance in execution time.

5.2.1. The execution time and the scalability

Figs. 14 and 15 display the execution time and the speedup for forward DWT for the lossless and lossy cases, respectively. Single SPE performance outperforms single PPE performance by far, and we demonstrate a remarkable speedup with additional SPEs. Vectorization contributes to the superb single SPE performance, and in the lossy encoding case, execution time is further reduced by replacing the fixed point representation with the floating point representation. Loop merging, applied to vertical filtering, removes the execution time for the splitting step and also reduces the bandwidth requirement. The reduced off-chip memory traffic, in combination with our data decomposition scheme, leads to higher scalability. Figs. 16 and 17 summarizes the impact of the adopted optimization schemes. The best execution time is achieved with 11 SPEs for the loss-

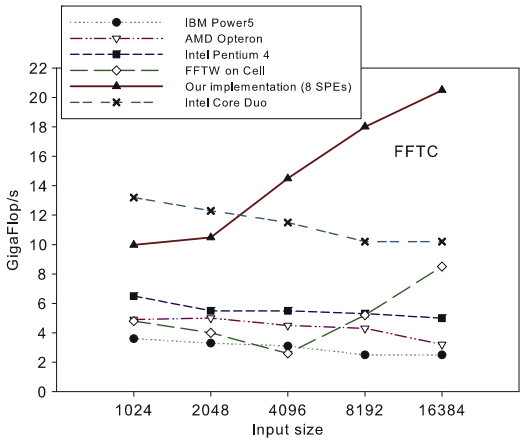

Fig. 13. Performance comparison of FFTC with other architectures for various input sizes of FFT. The performance numbers are from benchFFT from the FFTW website.

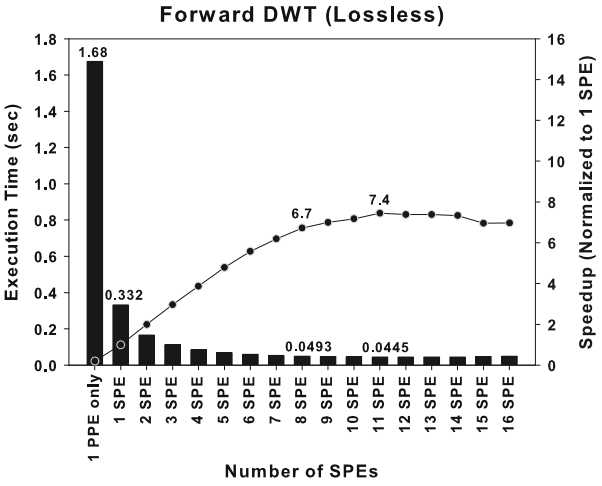

Fig. 14. The execution time and the speedup for lossless DWT.

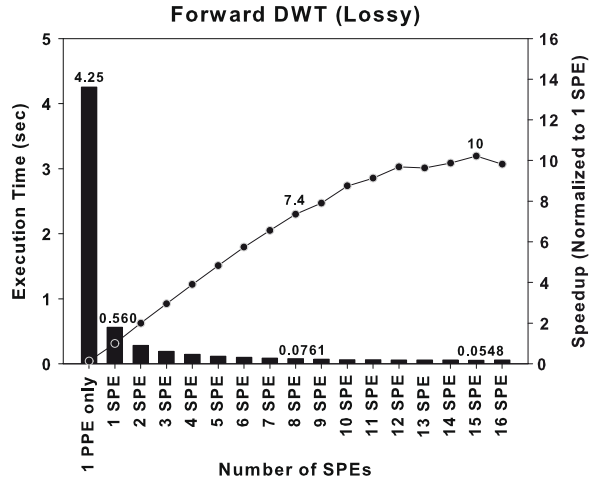

Fig. 15. The execution time and the speedup for lossy DWT.

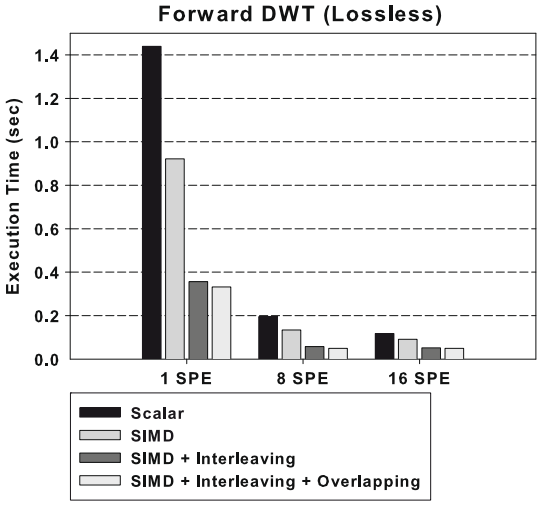

Fig. 16. The execution time for lossless DWT with the different levels of optimization.

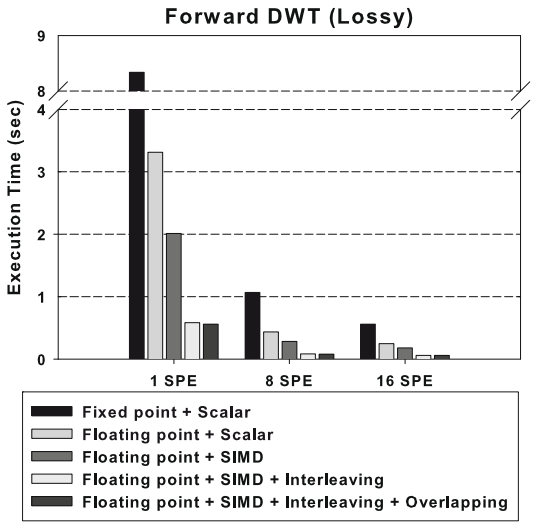

Fig. 17. The execution time for lossy DWT with the different levels of optimization.

less case and 15 SPEs for the lossy case as off-chip memory bandwidth becomes a performance bottleneck. The lossy case shows higher scalability than the lossless case as the lossy case has more computation (four lifting steps instead of two, one additional scaling step, and more computation in a single lifting step) per unit data transfer.

As vertical filtering and horizontal filtering have distinct data access patterns, and this affects both DMA data transfer and vectorization efficiency. In our case, the horizontal filtering takes longer than vertical filtering (1.07 and 1.28 times for the lossless and lossy cases) in a single SPE case owing to more efficient vectorization. Also, as a result of loop merging, vertical filtering does not involve the splitting step while horizontal filtering does. In 16 SPEs case, in contrast, vertical filtering takes 2.04 and 1.43 times longer than horizontal filtering, respectively. As the vertical filtering requires larger amount of data transfer, the horizontal filtering achieves higher scalability than vertical filtering and this leads to the performance gap.

5.2.2. Comparison with the previous implementation

Fig. 18 summarizes the performance comparison. Muta0 and Muta1 in the figure denote the implementations in [27], where Motion JPEG2000 encoding algorithm is optimized for the Cell/B.E. In Muta0, two encoding threads are executed on two Cell/B.E. chips while one encoding thread is executed using two Cell/B.E. chips in Muta1. The authors measure the

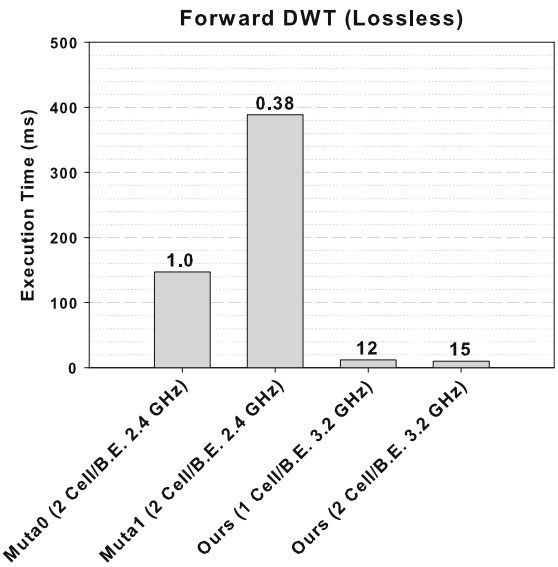

Fig. 18. The DWT performance comparison with the previous implementations for the Cell/B.E. The numbers above the bars denote the speedup relative to the Muta0 (2 Cell/B.E.)

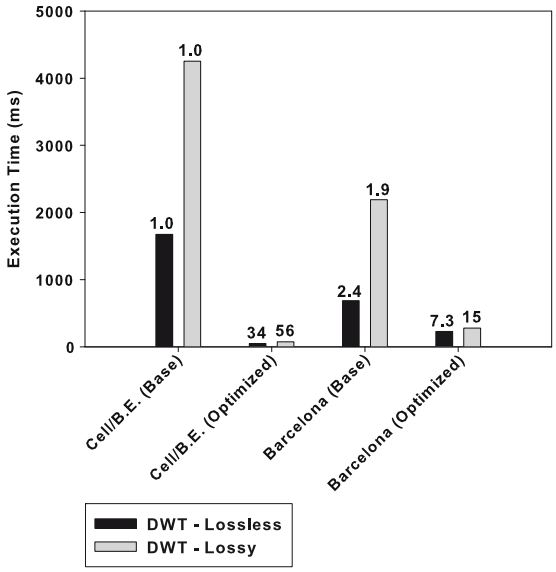

Fig. 19. The encoding performance comparison of the Cell/B.E. to that of the AMD Barcelona (Quad-core Opteron) processor. The numbers above the bars denote the speedup relative to the baseline implementation on the Cell/B.E.

encoding time for 24 frames with the size of 2048 × 1080. Total execution time is divided by the number of encoded frames to compute the per frame encoding time. In the first implementation (Muta0), the encoding time for one frame can be up to two times higher than the reported number. We use the performance numbers reported by the authors for the comparison. We also scale down the test image size to 2048 × 1080 for fair comparison. Note that the timings in [27] use the slower pre-production Cell/B.E. 2.4 GHz processor.

Our implementation with one Cell/B.E. processor and two Cell/B.E. processors demonstrate superior overall performance than the previous implementations with the two Cell/B.E. processors. This is mainly due to the following reasons. First, we run the experiments on the Cell/B.E. 3.2 GHz processor, which is faster than the Cell/B.E. 2.4 GHz used in [27]. Second, we adopt a lifting scheme, which is known to be faster than a convolution based scheme [30]. Third, we achieve 6.7 and 7.4 times speedup with 8 SPEs and additional speedup using two chips based on our data decomposition scheme and loop merging while in [27] performance does not scale above one SPE.

5.2.3. Comparison with the AMD Barcelona (Quad-core Opteron) processor

Fig. 19 summarizes the performance comparison between the Cell/B.E. and the AMD Barcelona 2.0 GHz (Quad-Core Opteron Processor 8350). We optimize the code for Barcelona using PGI C compiler, a parallel compiler for multicore optimization, and user provided compiler directives. Following summarizes the optimization approaches.

- *Parallelization:* OpenMP based parallelization.
- *Vectorization:* Auto-vectorization with compiler directives for pointer disambiguation.
- *Real number representation:* Identical to the Cell/B.E. case.
- *Loop merging:* Identical to the Cell/B.E. case.
- *Run-time profile feedback:* Compile with the run-time profile feedback (–Mpfo).

The above result shows that the baseline implementation for the Cell/B.E. (running on the PPE only) has lower performance, but the Cell/B.E. optimized code runs significantly faster than the AMD Barcelona optimized code. This demonstrates the Cell/B.E. processor's performance potential in processing regular and bandwidth intensive applications.

6. Conclusions

In summary, we present our design for computing two important discrete transform routines; fast Fourier and discrete wavelet transforms on the Cell Broadband Engine processor. For FFT, we use an iterative out-of-place approach and focus on inputs with 1K to 16K complex samples. We describe our methodology to partition the work among the SPEs to efficiently parallelize a single FFT computation. Our synchronization barrier is designed to use inter-SPE communication only without any intervention from the PPE. The synchronization barrier requires only $2 \log p$ stages (p:number of SPEs) of inter-SPE communication by using a tree-based approach. This significantly improves the performance, as the PPE intervention not only results in a high communication latency but also results in sequentializing the synchronization step. The computation on

the SPEs is fully vectorized with other optimization techniques such as loop unrolling and double buffering. Our implementation outperforms the Intel Duo Core (Woodcrest) for input sizes greater than 2K and to our knowledge, we have the fastest Fourier transform for the Cell/B.E. For DWT, we provide the novel data decomposition scheme and analyze its performance impact. Then, our data decomposition scheme is applied to DWT implementation to achieve efficient DMA data transfer and vectorization. We also introduce loop merging approach to reduce the bandwidth requirement. Our data decomposition scheme enhances the performance and reduces the programming complexity at the same time. The efficiency of the approach is demonstrated by the experimental results. In addition to the superior performance within a single Cell/B.E., our implementation shows the scalable performance on a single blade with two Cell/B.E. processors. This suggests that our approach will work efficiently even in the future Cell/B.E. processors with more SPEs. In conclusion, to unveil the Cell/B.E.'s full performance, judicious implementation strategy is required. Yet, under the proper implementation strategy, the Cell/B.E. can unleash its superb performance, especially for the floating point based loop intensive algorithms such as FFT and DWT. Our future work investigates novel strategies for computing transforms on the PowerXCell 8i double-precision Cell/B.E. processor.

Acknowledgements

This work was supported in part by an IBM Shared University Research (SUR) award and NSF Grants CNS-0614915, CAREER CCF-0611589, and DBI-0420513. We would also like to thank Michael Perrone, Bob Szabo, and Sidney Manning (IBM Corporation) for providing valuable inputs during the course of our research. We acknowledge our Sony–Toshiba–IBM Center of Competence for the use of Cell Broadband Engine resources that have contributed to this research.

References

[1] M.D. Adams, F. Kossentini, Jasper: a software-based JPEG-2000 codec implementation, in: Proceedings, 2000 Int'l Conf. on Image Processing, vol. 2, Vancouver, BC, Canada, September 2000, pp. 53–56.
[2] R.C. Agarwal, J.W. Cooley, Vectorized mixed radix discrete Fourier transform algorithms, Proc. IEEE 75 (9) (1987) 1283–1292.
[3] M. Ashworth, A.G. Lyne, A segmented FFT algorithm for vector computers, Parallel Computing 6 (2) (1988) 217–224.
[4] A. Averbuch, E. Gabber, B. Gordissky, Y. Medan, A parallel FFT on an MIMD machine, Parallel Computing 15 (1990) 61–74.
[5] D.A. Bader, V. Agarwal, FFTC: Fastest Fourier transform for the IBM Cell Broadband Engine, in: Proceedings 14th Int'l Conf. on High Performance Computing (HiPC 2007), Springer-Verlag, Goa, India, December 2007.
[6] D.H. Bailey, A high-performance FFT algorithm for vector supercomputers, Int'l J. Supercomputer Applications 2 (1) (1988) 82–87.
[7] P.J. Burt, E.H. Anderson, The Laplacian pyramid as a compact image code, IEEE Trans. Commun. 31 (4) (1983) 532–540.
[8] D. Chaver, M. Prieto, L. Pinuel, F. Tirado, Parallel wavelet transform for large scale image processing, in: Proceedings of the Int'l Parallel and Distributed Processing Symposium (IPDPS 2002), Ft. Lauderdale, FL, USA, April 2002, pp. 4–9.
[9] T. Chen, R. Raghavan, J. Dale, E. Iwata, Cell Broadband Engine Architecture and its first Implementation, Technical Report, November 2005.
[10] A.C. Chow, G.C. Fossum, D.A. Brokenshire, A programming example: large FFT on the Cell Broadband Engine, in: Technical Conference Proceedings of the Global Signal Processing Expo (GSPx), 2005.
[11] L. Cico, R. Cooper, J. Greene, Performance and Programmability of the IBM/Sony/Toshiba Cell Broadband Engine Processor, White Paper, 2006.
[12] IBM Corporation, Cell Broadband Engine Technology. <http://www.alphaworks.ibm.com/topics/cell>.
[13] IBM Corporation, The Cell Project at IBM Research. <http://www.research.ibm.com/cell/home.html>.
[14] B. Flachs et al., A streaming processor unit for a Cell processor, in: International Solid State Circuits Conference, vol. 1, San Fransisco, CA, USA, February 2005, pp. 134–135.
[15] M. Frigo, S.G. Johnson, FFTW on the Cell Processor, 2007. <http://www.fftw.org/cell/index.html>.
[16] H.P. Hofstee, Cell Broadband Engine Architecture from 20,000 feet, Technical Report, August 2005.
[17] IBM, PowerXCell 8i Processor, White Paper, 2008.
[18] ISO and IEC, ISO/IEC 15444-1: Information Technology – JPEG2000 Image Coding System – Part 1: Core Coding System, 2000.
[19] C. Jacobi, H.-J. Oh, K.D. Tran, S.R. Cottier, B.W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, The vector floating-point unit in a synergistic processor element of a Cell processor, in: Proceedings of the 17th IEEE Symposium on Computer Arithmetic, Washington, DC, USA, IEEE (ARITH'05) Computer Society, 2005, pp. 59–67.
[20] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, D. Shippy, Introduction to the Cell multiprocessor, IBM J. Res. Dev. 49 (4/5) (2005) 589–604.
[21] S. Kang, D.A. Bader, Optimizing JPEG2000 still image encoding on the cell broadband engine, in: Proceedings of the 37th Int'l Conf. on Parallel Processing (ICPP), Portland, OR, September 2008.
[22] M. Kistler, M. Perrone, F. Petrini, Cell multiprocessor communication network: built for speed, IEEE Micro 26 (3) (2006) 10–23.
[23] D. Krishnaswamy, M. Orchard, Parallel algorithms for the two-dimensional discrete wavelet transform, in: Procedings of the Int'l Conf. Parallel Processing, vol. 3, 1994, pp. 47–54.
[24] R. Kutil, A single-loop approach to SIMD parallelization of 2-D wavelet lifting, in: Proceedings of the Euromicro Int'l Conf. on Parallel, Distributed, and Network-based Processing (PDP 2006), Montbeliard, France, February 2006.
[25] C.-J. Lian, K.-F. Chen, H.-H. Chen, L.-G. Chen, Analysis and architecture design of block-coding engine for EBCOT in JPEG 2000, Circuits and Systems 13 (3) (2003) 219–230.
[26] P. Meerwald, R. Norcen, A. Uhl, Parallel JPEG2000 image coding on multiprocessors, in: Proceedings of the Int'l Parallel and Distributed Processing Symposium (IPDPS 2002), Ft. Lauderdale, FL, USA, April 2002, pp. 2–7.
[27] H. Muta, M. Doi, H. Nakano, and Y. Mori. Multilevel Parallelization on the Cell/B.E. for a Motion JPEG 2000 Encoding Server. In Proc. ACM Multimedia Conf. (ACM-MM 2007), Augsburg, Germany, Sep. 2007.
[28] O.M. Nielsen, M. Hegland, Parallel performance of fast wavelet transform, Int'l J. of High Speed Computing 11 (1) (2000) 55–73.
[29] D. Pham et al., The design and implementation of a first-generation Cell processor, in: International Solid State Circuits Conference, vol. 1, San Fransisco, CA, USA, February 2005, pp. 184–185.
[30] A. Shahbahrami, B. Juurlink, S. Vassiliadis, Performance comparison of SIMD implementations of the discrete wavelet transform, in: Proceedings of the Int'l Conf. on Application Specific Systems, Architecture and Processors, 2005, pp. 393–398.
[31] Sony Corporation, Sony Release: Cell Architecture. <http://www.scei.co.jp/>.
[32] W. Sweldens, The lifting scheme: a new philosophy in biorthogonal wavelet constructions, in: Proceedings of the SPIE, Wavelet Applications in Signal and Image Processing III, vol. 2569, September 1995, pp. 68–79.
[33] W. Sweldens, The lifting scheme: a construction of second generation wavelets, SIAM J. Math. Anal. 29 (2) (1998) 511–546.

[34] D.S. Taubman, M.W. Marcellin (Eds.), JPEG2000 Image compression fundamentals, standards, and practice, Kluwer Academic Publishers, 2002.
[35] D.S. Taubman, M.W. Marcellin, JPEG2000: standard for interactive imaging, Proc. IEEE 90 (8) (2002) 1336–1357.
[36] P.P. Vaidyanathan, Quadrature mirror filter banks, m-band extensions, and perfect reconstruction techniques, IEEE ASSP Magazine 4 (7) (1987) 4–20.
[37] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, K. Yelick, The potential of the Cell processor for scientific computing, in: Proceedings of the 3rd Conference on Computing Frontiers (CF'06), ACM Press, New York, USA, 2006, pp. 9–20.
[38] L. Yang, M. Misra, Coarse-grained parallel algorithms for multi-dimensional wavelet transforms, J. Supercomput. 12 (1–2) (1998) 99–118.