HIGHLY PARALLEL MULTI-DIMENSIONAL FAST FOURIER TRANSFORM ON FINE- AND COARSE-GRAINED MANY-CORE APPROACHES

A. Beliz Saybasili
Laboratory of Computational Biology, NIH/NHLBI,
Bethesda, Maryland 20892, USA
and Istanbul Technical University, Maslak, Istanbul, Turkey
email: saybasiliab@mail.nih.gov

Alexandros Tzannes
University of Maryland
Computer Science Department
College Park, Maryland, 20742, USA
email: tzannes@umd.edu

Bernard R. Brooks
Laboratory of Computational Biology, NIH/NHLBI,
Bethesda, Maryland 20892, USA
email: brb@mail.nih.gov

Uzi Vishkin
University of Maryland,
Institute for Advanced Computer Studies,
College Park, Maryland, 20742, USA
email: vishkin@umd.edu

ABSTRACT
Multi-dimensional fixed-point fast Fourier transform (FFT) methods were developed and tested on two general-purpose many-core architecture platforms. One is the highly-parallel fine-grained eXplicit Multi-Threaded (XMT) single-chip parallel architecture that targets reducing single task completion time. The second is 8xDual Core AMD Opteron 8220. The results show that the former outperforms the latter not only in speedup and ease of programming, but also with small data sets (small-scale parallelism). One of our results on XMT was a super-linear speedup (by a factor larger than the number of processors) observed under some rather unique circumstances.

KEY WORDS
Many-cores, parallel algorithmic computer architecture, eXplicit Multi-Threading (XMT), parallel multi-dimensional fixed-point FFT.

1 Introduction

The fast Fourier transform (FFT) is an essential primitive for many scientific and engineering applications including signal processing, image processing, particle simulation, telecommunication, sound engineering and other related fields. Parallelization of FFT has been widely investigated. In distributed systems, the most difficult part of the parallel multi-dimensional FFT implementation is data transpose, whereas in shared memory systems the bottleneck is the high number of access to the memory [1].

In the days when general-purpose computing meant serial computing, high-performance requirements on parallel kernels such as FFT mandated going to either application specific architectures, or to large and expensive parallel machines. However, with the advent of the many-core era it is worthwhile studying the opportunity for tapping these new parallel processing resources that are becoming available on general-purpose platforms. The study reported in this paper reveals significant performance gaps between two many-core approaches.

Franchetti et al. studied FFT performance on Intel Core Duo, Intel Pentium D, AMD Opteron Dual Core, and Intel Xeon MP emphasizing understanding the algorithm before optimizing the implementation [2]. There have been numerous single- or multi-dimensional parallel FFT implementations on Cell [3–7], Cyclops-64 [8] and Graphics Processing Unit (GPU) [9–11]. These implementations achieve good performance via considerable amount of work on synchronization, locality and careful programming. FFT on GPU gained widespread usage, however, GPU is not a general purpose architecture and its computing model is not flexible and requires careful programming [11].

The eXplicit Multi Threading (XMT) is a *general purpose* single-chip parallel architecture aimed at *reducing single task completion time*. XMT provides good performance for parallel programs based on *Parallel Random Access Model (PRAM)* thinking. PRAM is a well-studied parallel computation model, essentially there is a PRAM algorithm for every problem with a serial algorithm. The XMT programming model enables easy programming unlike most other contemporary many-core computers. A serial algorithm prescribes which single operation to perform next, assuming a fast memory of unlimited size and other hardware that can execute it immediately (henceforth, referred as *serial abstraction*). A PRAM algorithm, on the other hand, prescribes all operations that could be performed next, assuming unlimited hardware that can execute these operations immediately (*the PRAM abstraction*). By 1990, PRAM algorithms were on their way to becoming standard computer science know-how that every computer science professional must command, and the basis for standard parallel programming. However, 1990s fabrication technology was not yet capable of supporting parallel computers on which PRAM algorithms would perform well. Recently, advances in silicon integration technology led to an FPGA prototype that fully implements the PRAM approach. In 2007, Wen and Vishkin introduced the first FPGA prototype of XMT [12]. At present, an XMT FPGA implementation is available for testing [13]. The XMT programming environment including a cycle-accurate simulator and a compiler can be freely downloaded from XMT's software release website [14]. The environment allows experimenting with XMT, and also teaching and studying parallelism.

Contribution. In this paper, we present an efficient fixed-point multi-dimensional parallel radix-2 FFT implementation on eXplicit Multi-Threading. We evaluated the same implementation on a state-of-the-art existing processor, 8xDual Core AMD Opteron 8220 for comparison purposes.

XMT supports *short* threads (*not* operating system threads), thus a fine-grained approach. The AMD platform can accommodate coarse and fine-grained threads. For tasks where a coarse-grained approach does not provide the best performance, an OpenMP-based implicit fine-grained approach can be used. Although this approach makes programming easier, it requires the existence of a suitable algorithm that allows clear functional decomposition [15].

The results show that XMT outperforms the AMD platform in speedup, and also achieves speedups with small data sets (small-scale parallelism). On XMT, super-linear speedups (by a factor larger than the number of processors) were observed with some large data. We attribute this phenomenon to prefetching allowing high utilization of the interconnection network and the parallel memory architecture of XMT.

This paper is organized as follows. Section 2 briefly reviews the XMT environment and the FFT algorithm. Fixed-point implementation details and parallelization of the multi-dimensional FFT is presented in section 3. The results of the experiments and discussion are given in section 4 and section 5, respectively. Section 6 provides the summary of the whole study giving emphasis to future work.

2 Background

2.1 eXplicit Multi-Threading

XMT offers a complete parallel computing framework aiming high performance by easy programming. It provides good performance for PRAM-like algorithms. A PRAM algorithm is advanced to an XMT program with the work-depth methodology [16]. This methodology introduces the concept of *work*, the total number of operations that the PRAM algorithm perform, and *depth*, the parallel computation time assuming unlimited hardware resources. Such an algorithm is coded in XMTC, a *Single Program Multiple Data (SPMD)* extension to C programming language, and compiled with the XMTC compiler that takes advantage of the full capabilities of the XMT hardware.

2.2 XMT Architecture

The XMT architecture consists of a Global Register File (GRF), a serial processor, called Master Thread Control Unit (MTCU), clusters ($C_0,\dots,C_{n-1}$) of Thread Control Units (TCUs), a prefix-sum unit, multiple memory modules ($MM_0,\dots,MM_{m-1}$) and a high bandwidth interconnection network between all modules. Scheduling of individual virtual threads onto TCUs is done efficiently in hardware by XMT's unique prefix-sum unit, in a completely transparent manner. A high level representation of the architecture is given in Figure 1.

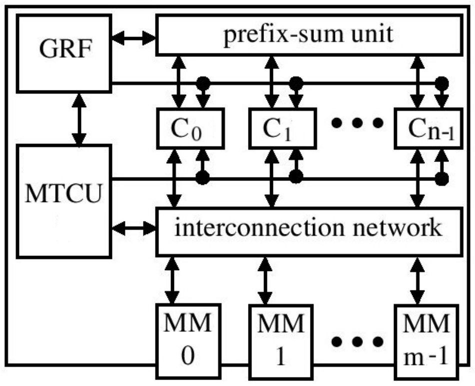

Figure 1. XMT architecture

Cache coherence is a challenging and yet unsolved problem on a large number of cores. To mitigate this problem, XMT has local cache only at the MTCU and has read-only caches and prefetch buffers at TCUs. More information on the current XMT architecture and FPGA prototype specifications can be found in Wen and Vishkin [13, 17].

2.3 XMTC Programming Language

XMTC is a super-set of the C programming language with additional statements: *spawn*, *join* and *ps/psm*. XMT has both serial and parallel modes. The execution starts in serial mode during which code is executed on the MTCU. Parallel mode, executed on the TCUs, is initiated by the statement *spawn*: $spawn(thread\_min, thread\_max)\{\dots\}$ creates threads with virtual ids between $(thread\_min, thread\_max)$. There is no need for an explicit *join* statement because it is called implicitly at the end of a spawn block. Subsequent to the execution of a spawn block, MTCU resumes the remaining part in serial mode until a new spawn statement is encountered. This behavior is represented in Figure 2. The instruction $ps(local, psRegister)$ performs an atomic fetch-and-add on a prefix-sum register. The $psm(local, variable)$ instruction performs the same operation but on a memory location. Further details of the XMTC language usage can be found in XMTC tutorial and manual [18, 19].

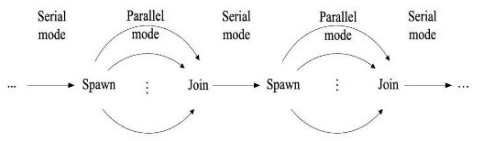

Figure 2. XMT serial and parallel execution flow

2.4 Fast Fourier Transform

FFT is a fast and efficient way to compute computationally expensive discrete Fourier transform (DFT) which requires $O(N^2)$ complex multiplications for N complex points. Although there are numerous FFT algorithms in the literature, we focus on divide-and-conquer type radix-2 Cooley-Tukey FFT, which requires $O(NlogN)$ complex multiplications. In this method, N-point data are divided into two $N/2$ sub-sequences; even-indexed elements and odd-indexed elements, as follows:

$$X_n = \sum_{k=0}^{N/2-1} x_{2k} \times e^{2\pi i n(2k)/N} + \sum_{k=0}^{N/2-1} x_{2k+1} \times e^{2\pi i n(2k+1)/N}. \qquad (1)$$

Every sub-sequence in Equation 1 is then divided into half-sized sub-sequences until the smallest sequence is reached. By defining the root of the unity $\omega = e^{2\pi i/N}$, also called the *twiddle factor*, we obtain the following expansion for the FFT equation.

$$X_n = X_n^{even} + \omega^n X_n^{odd}. \qquad (2)$$

The algorithm runs in stages and the twiddle factor is used to combine the output of a previous stage with the input of the following stage. To access the data, the sequence needs to be reversed at the beginning of the stages, which is a *bit reversal* in radix-2 FFT. Because of the radix-2 FFT stage execution flow structure, this algorithm is also called butterfly algorithm (see Figure 3). Butterfly FFT is well suited for parallelism because every butterfly stage is performed in parallel.

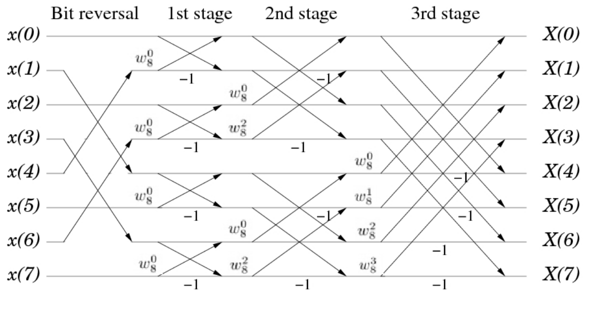

Figure 3. 16-point FFT butterfly

The FFT computes the transform of a *single-dimensional* sequence. Multi-dimensional FFT is defined for *multi-dimensional array* and consists of many single-dimensional transforms along dimensions. 2D FFT is defined on two-dimensional grid. In the parallelization of 2D FFT, two solutions can be considered. Every row-FFT can be performed in parallel or every row-FFT can be assigned to different thread and every thread can perform an independent FFT on it. The later one is a more coarse-grained approach.

3 Implementation

Our FFT implementation on XMT was based on the radix-2 Cooley-Tukey algorithm. This algorithm can be problematic for many architectures because of a false sharing causing unnecessary cache coherence traffic [1,2]. XMT is free from cache coherence and we show that we obtained good performance with our radix-2 Cooley-Tukey FFT implementation on XMT. The same algorithm is implemented in several studies on many-cores including IBM Cell Broadband Engine [5,7] and IBM Cyclops-64 [8]. Because of current physical limitations of XMT, such as small memory size, we used in-place FFT: we directly changed the input data and converted it to its transform, therefore removed the need to allocate space for the output array.

The current XMT FPGA prototype supports only integer arithmetic, and incorporating floating-point to XMT is a subject of research. We implemented FFT using fixed-point arithmetic. In fixed-point arithmetic, numbers are represented with fixed number of digits after the decimal point. Assignment of a limited number of bits before and after the decimal point can cause precision loss and overflow. Precision loss is caused by the need of rounding the result after the multiplication of two numbers, and overflow is caused by the adding of two numbers. There is a tradeoff between rounding and overflow handling. By using accurate twiddle factors that are pre-computed by Octave, we focus on overflow handling [20]. There are several methods to prevent overflow including input scaling and scaling intermediate data where the latter one gives gives a better error bound [21]. Numbers are scaled by a factor of $1/2$ in each stage to prevent overflows. The overall scaling is then $1/N$, N being input data size. XMT is a 32-bit processor, we adjusted our scaling factor (SF) to be between 10 and 15, according the data size.

Given input size N, we implemented complex data type in two arrays *xr[N]* and *xi[N]* representing complex and imaginary parts respectively. Twiddle factors were kept in a single array *w[N]*. 1D parallel fixed-point FFT butterfly algorithm with input size N is given as follows.

```
1: $step = N$
2: **for** $level = 1$ **to** $level < N$ **do**
3:     $level2 = level << 1$
4:     $step >>= 1$
5:     **for** $tid = 0$ **to** $tid = N/2 - 1$ **pardo**
```

```
6:          $k = (step) * (tid\%level)$                      {twiddle index}
7:          $i = (tid/level) * level2 + (tid\%level)$        {src. index}
8:          $j = i + level$                                  {dest. index}
9:          $wr = w[k + N/4] >> 1$                           {twiddle factor real part}
10:         $wi = -w[k] >> 1$                                {twiddle factor imaginary part}
            {complex multiplications:}
11:         $xr[j] = (xr[i] >> 1) - [(wr * xr[j]) >> SF$
                    $-(wi * xi[j]) >> SF]$
12:         $xi[j] = (xi[i] >> 1) - [(wr * xi[j]) >> SF$
                    $+(wi * xr[j]) >> SF]$
13:         $xr[i] = (xr[i] >> 1) + [(wr * xr[j]) >> SF$
                    $-(wi * xi[j]) >> SF]$
14:         $xi[i] = (xi[i] >> 1) + [(wr * xi[j]) >> SF$
                    $+(wi * xr[j]) >> SF]$
15:     **end pardo**
16:     $level <<= 1$
17: **end do**
```

We extended 1D FFT to 2D by applying separate 1D parallel FFTs to every dimension. Because XMT supports fine-grained approach, this implementation gave better performance than applying serial FFTs in parallel, which is a coarse-grained approach. We kept the matrix input in single-dimensional arrays ($xr[N]$ and $xi[N]$), which fits perfectly *shuffle transpose* scheme [22]. Perfect shuffle introduces the notion of a *shuffle interconnection pattern*. In this scheme, the matrix data of size $2^n \times 2^n$ is stored in a row such that row index is major. Row elements are interchanged during the process. The data passes n times from that interconnection and yields the result, as illustrated in the Figure 4.

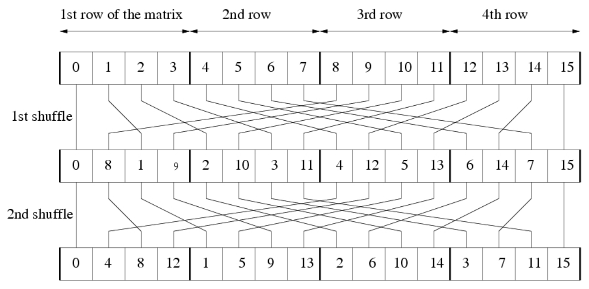

Figure 4. Shuffle transpose with $n = 2$

Perfect shuffle algorithm is as follows.

```
1: **for** $i = 1$ to $n$ **do**
2:     **for** $k = 1$ to $2^{2n} - 2$ **pardo**
3:         $xr[(2k)mod(2^{2n} - 1)] = xr[k]$
4:         $xi[(2k)mod(2^{2n} - 1)] = xi[k]$
5:     **end pardo**
6: **end do**
```

We implemented fixed-point parallel 1D and 2D FFT first on XMT, and then on a Custom Linux workstation with 8xDual Core AMD Opteron 8220 and with 16 GB of RAM. Following a suggestion of an Intel engineer, we collected clock cycle counts for our measurements [23]. XMT FPGA is configured to give cycle counts at the end of execution. We used *Time Stamp Counter* on AMD platform.

The 1D FFT input data was a combination of sinusoidal functions. We chose the data size to be 2^N with N varying between 3 and 25. Maximum level of optimization (-O3) in our homegrown GCC-based XMTC compiler was activated. We executed both serial and parallel fixed-point FFT for each data set. We computed 64-TCU speedups with respect to the MTCU. For the 2D FFT implementation input, we used grayscale images with resolution between 8×8 and 4096×4096. Due to current limitations of XMT hardware, such as small storage, it was not possible to make comparisons with larger data sets.

4 Results

4.1 FFT on XMT

Our serial and parallel programs were executed for each data set, and cycle counts were collected for performance analysis. Some of the results for 1D FFT are given in Table 1.

Table 1. 1D FFT cycle counts on XMT

N	2^3	2^{10}	2^{15}	2^{16}
serial	3974	858872	86304774	226328070
parallel	3173	28384	1268302	4707640
speedup	1.2	30.2	68.0	48.0

An example of 128×128 grayscale input image (Lena) and the resulting image of inverse-FFT applied to XMT-FFT output is shown in Figure 5. The figure also shows a fixed-point 2D Gaussian low-pass and high-pass filter applications to the resulting image on XMT. Parallel filtering of the 128×128 image on XMT was 30 times faster than the serial one. Cycle counts of 2D FFT for different resolution images were collected. The results for some data sets are given in Table 2.

Table 2. 2D FFT cycle counts on XMT

resolution	16×16	128×128	512×512
serial	182297	35168274	758806964
parallel	75145	1720174	63439303
speedup	2.4	20.4	11.9

The performance gain was nearly linear in both 1D and 2D FFT. The speedup decreased with data size larger than 2^{15} in 1D and 2D FFT. XMT has a shared-cache size $256KB$, which can hold $32K(2^{15})$ complex type data. If the data is larger than shared cache size, access to main memory is required. At that point, memory access latency is augmented by access to main memory. Consequently, the

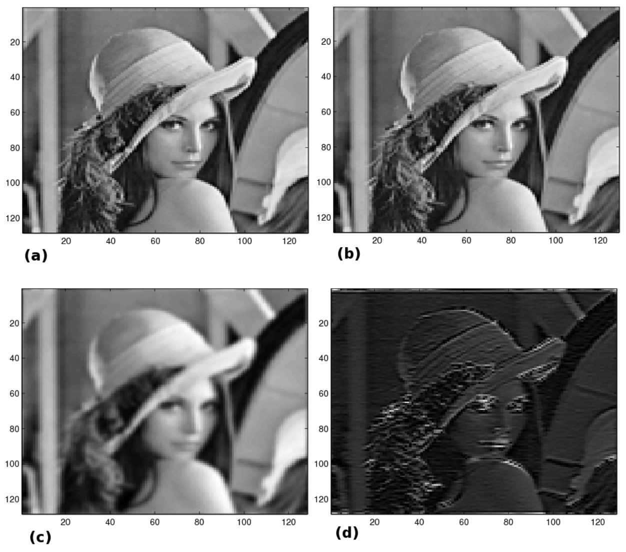

Figure 5. (a) 128×128 input "Lena" image. (b) XMT output of FFT restored back using inverse-FFT function of Octave (c) XMT output of the image filtered by 2D Gaussian low-pass filter restored back using inverse-FFT function of Octave (d) XMT output of the image filtered by 2D Gaussian high-pass filter restored back using inverse-FFT function of Octave

performance decreases. Curiously, with data set ($N = 2^{15}$) in 1D FFT, XMT gave a speedup of 68, higher than theoretical expectation of linear speedup of 64. This speedup was related to TCU prefetching together with low-latency high-bandwidth interconnection network, and the efficient parallel memory architecture [24]. TCUs have prefetching, whereas MTCU has a cache but no prefetching capability. In parallel mode, TCU prefetching allows XMT to provide more uniform memory access latencies, and exploitation of the underlying hardware. The speedup became sub-linear (45 for $N = 2^{15}$) when the input complex data were stored in a one 2D array instead of two 1D arrays. The use of a single array did not show the advantage of prefetching. The comparison of these two cases can be seen in Figure 6. Similarly, when we disabled TCU prefetching property, the speedup became sub-linear.

4.2 Comparison with Many-Core AMD

XMTC programs were converted to OpenMP and tested with the same data sets on a 8xDual Core AMD Opteron 8220.

On the AMD platform, our 1D FFT implementation did not get any speedup until a data size of $N = 2^{15}$ was reached (which gave speedup of 2). Similarly, in our 2D FFT implementation on AMD platform, no speedup was observed until a data size of $N \times N = 2048 \times 2048$ was reached (which gave speedup of 3). A graphical representation of 1D FFT and 2D FFT speedup comparisons of

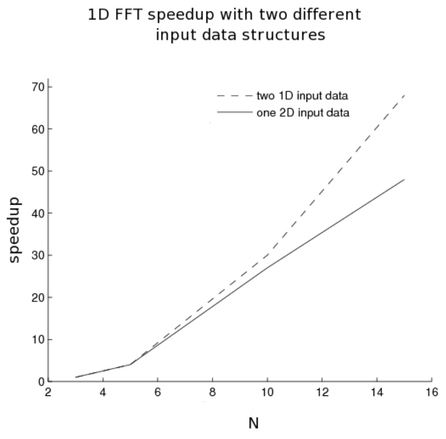

Figure 6. Speedup comparison of 1D FFT in two cases: two 1D input data vs one 2D input data. The speedup becomes sub-linear with 2D input data structure. Data size is 2^N

both XMT and AMD is shown in Figure 7 and Figure 8, respectively. We implemented both 1D and 2D FFT on a 16-TCU XMT configuration as well. As mentioned previously, when the data became larger than the shared cache module size, which was 2^{15}, the performance on XMT decreased because of the off-chip memory access latency.

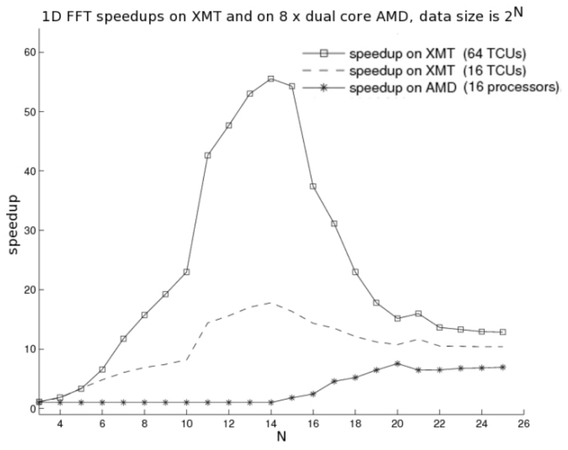

Figure 7. Speedup of 1D FFT on XMT with 64 TCU configuration, on XMT with 16 TCU configuration, and on AMD where there are 16 processors. Our 1D FFT implementation with data sizes less than 2^{15} does not get any speedup on AMD.

5 Discussion

A 64-TCU XMT ASIC chip needs about the same silicon area as a single state-of-the-art commodity core. However, we did not restrict our comparison, based on silicon area, to only comparing a 64-TCU XMT with a single Opteron. But, even comparison of a 16-core Opteron and a 16-TCU

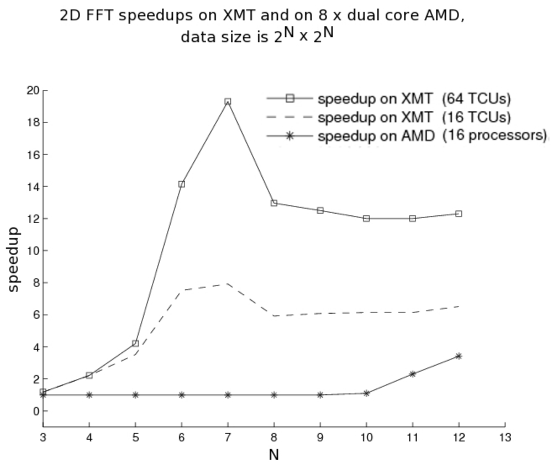

Figure 8. Speedup of 2D FFT on XMT with 64 TCU configuration, on XMT with 16 TCU configuration, and on AMD where there are 16 processors. Our 2D FFT implementation on AMD gets speedup with 2048×2048 input data.

XMT, suggests that XMT has some advantage.

Our paper evaluates FFT only by fixed-point implementations. This work does not claim a role for predicting floating-point performance. Several decision parameters such as the number of floating-point unit (FPU) per XMT chip or per TCU, and design complexity of the FPU would affect floating-point performance. The lack of floating-point support would be insignificant for applications that are already based on fixed-point FFTs, including real-time fixed-point image reconstruction for spiral MRI [25], some mobile TV and some video codec applications, as long as they do not reflect a much larger investment in total silicon area. On the other hand, our FFT implementation on XMT provides speedup also for small data sizes, which was not possible for AMD, and most modern multi-core architectures. XMT would be advantageous for applications using large amounts of small size FFTs, done one after the other, such as sound engineering and edge detection applications.

Contemporary application-specific many-cores have strong performance but offer a limited programming model that is not flexible enough for general purpose computing. As an example, GPU does not allow thread interaction which prevents many parallel graph algorithms, such as breadth first search, from having an efficient implementation, whereas XMT offers an easy and elegant solution [16]. When it comes to high-throughput oriented many-cores, the programmer has to figure out lots of specialized work, like locality, load balancing and synchronization, to get good speedup. Franchetti et al suggest that in the future, concurrency will pose a major burden on compiler developers and programmers [2], which seems to be the case for the aforementioned platforms. On the other hand, XMT gives good performance with any amount and type (grain or regularity) of parallelism provided by the algorithm; up- and down-scalability including backwards compatibility on serial code, which make it much easier to program. Our FFT implementation on XMT did not require any explicit synchronization, data partitioning or load balancing opposed to FFT on state-of-the-art many-cores [3–11]. It has been previously demonstrated that XMT outperforms Intel Core 2 Duo for some algorithm implementations including matrix-vector multiplication and quicksort [23].

XMTC extends the C programming language, but other programming languages, including Fortran and Java, could be extended in a similar way as well. In general, only knowledge of a basic programming language and interest in algorithms would be enough to develop efficient parallel XMT programs. A parallel programming course was given to high school students in late 2007 at the University of Maryland. They only had a 5-hour introductory tutorial and weekly practice sessions with the guidance of an undergraduate teaching assistant. All of the students were able to comprehend PRAM thinking and some have even successfully performed graduate-level programming assignments. This experience reinforces our claim that XMT is easy to program.

6 Conclusion

In this study, we demonstrated that XMT can be competitive with 8xDual Core AMD Opteron 8220 in FFT-based applications. Unlike many other contemporary many-core computers, XMT is well suited for small-scale parallelism as well. In some FFT implementations with a large data size, XMT provided speedup higher than theoretical expectations as can be seen in Figure 6. This speedup was provided by TCU prefetching and XMT's highly optimized interconnection network and memory architecture that allows multiple memory requests from the same or different TCUs to be routed simultaneously.

A highly parallel multi-dimensional fixed-point FFT implementation for the XMT architecture was developed and demonstrated. A future extension of this work would involve comparing our FFT implementation performance on XMT to other popular many-cores including GPU and Cell. This is an ongoing effort as new chips and architects emerge. Good performance results encourage us to develop a basic linear algebra library for XMT. FFT is tolerant to low precision and is well suited for fixed-point implementation. However, there are many implementations requiring higher precision. Currently, XMT floating-point support is in its final stage of development. Floating-point FFT will be evaluated once the hardware support is available and further performance studies will be realized. With the addition of that support, XMT will be available for high-precision applications as well. Two other challenges include using our FFT implementation in a complete scientific application and optimization of the algorithm for large data sizes exceeding on-chip memory size.

References

[1] M. Frigo and S.G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005.

[2] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. FFT program generation for shared memory: SMP and multicore. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 115, New York, NY, USA, 2006. ACM.

[3] A.C. Chow, G.C. Fossum, and D.A. Brokenshire. A programming example: Large FFT on the Cell Broadband Engine. In *Global Signal Processing Expo (GSPx)*, 2005.

[4] J. Greene and R. Cooper. A parallel 64k complex FFT algorithm for the IBM/Sony/Toshiba Cell Broadband Engine Processor. In *Global Signal Processing Expo (GSPx)*, 2005.

[5] D.A. Bader and V. Agarwal. FFTC: Fastest Fourier Transform on the IBM Cell Broadband Engine. In *14th IEEE Int. Conf. on High Performance Computing (HiPC'07)*, 2007.

[6] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *ACM Proc. of the 3rd Conf. on Comput. Frontiers (CF'06)*, pages 9–20, 2006.

[7] D. A. Bader, V. Agarwal, and S. Kang. Computing discrete transforms on the cell broadband engine. *Parallel Comput.*, 35(3):119–137, 2009.

[8] L. Chen, Z. Hu, and J. Lin G.R. Gao. Optimizing the fast fourier transform on a multi-core architecture. In *IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[9] K. Moreland and E. Angel. The FFT on a GPU. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware (HWWS'03)*, pages 112–119, 2003.

[10] O. Fialka and M. Cadik. FFT and convolution performance in image filtering on GPU. In *10th International Conference on Information Visualization (IV'06)*, pages 609–614, 2006.

[11] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *Proc. of the IEEE/ACM Conf. on Supercomputing (SC'08)*, 2008.

[12] X. Wen and U. Vishkin. PRAM-on-chip: first commitment to silicon. In *Proc. 19th ACM Symp. on Parallel Algorithms and Archit. (SPAA'07)*, 2007.

[13] X. Wen and U. Vishkin. FPGA-based prototype of a PRAM-on-chip processor. In *ACM Comput. Frontiers*, 2008.

[14] Software release of the explicit multi-threading (XMT) programming environment. `http://www.umiacs.umd.edu/users/vishkin/XMT/sw-release.html`, Aug 2008.

[15] A. Zeichick. Coarse-grained vs. fine-grained threading for native applications, part I and part II. AMD Developer central. `http://developer.amd.com/documentation/articles/Pages/default.aspx`, Feb 2006.

[16] U. Vishkin, G. Caragea, and B. Lee. *Parallel Computing: Models, Algorithms and Applications*, chapter Models for advancing PRAM and other algorithms into parallel programs for a PRAM-On-Chip platform. CRC press, 2008.

[17] X. Wen and U. Vishkin. The XMT FPGA prototype/cycle-accurate-simulator hybrid. In *The 3rd Workshop on Architectural Research Prototyping (WARP'08)*, 2008.

[18] A. Tzannes, C. Caragea, A.O. Balkan, and U. Vishkin. XMT-C tutorial. `http://www.umiacs.umd.edu/users/vishkin/XMT/tutorial4xmtc2out-of2.pdf`, Nov 2008.

[19] A. Tzannes, C. Caragea, A.O. Balkan, and U. Vishkin. XMT-C manual. `http://www.umiacs.umd.edu/users/vishkin/XMT/manual4xmtc1out-of2.pdf`, Jan 2009.

[20] W. M. Gentleman and G. Sande. Fast fourier transforms - for fun and profit. *Proc. of the AFIPS*, 29:563–578, 1966.

[21] A.V. Oppenheim, R.W. Schafer, and J.R. Buck. *Discrete-Time Signal Processing*. Prentice Hall, 2nd edition, 1999.

[22] H. S. Stone. Parallel processing with the perfect shuffle. In *IEEE Trans. Comput.*, volume C20, pages 153–161, 1971.

[23] G.C. Caragea, A.B. Saybasili, X. Wen, and U. Vishkin. Performance potential of an easy-to-program PRAM-On-Chip prototype versus state-of-the-art processor. In *Proc. 21st ACM Symp. on Parallel Algorithms and Archit. (SPAA'09)*, 2009.

[24] A. Balkan, G. Qu, , and U. Vishkin. An area-efficient high-throughput hybrid interconnection network for single-chip parallel processing. In *45th Design Automation Conference*, 2008.

[25] J. R. Liao. Real-time image reconstruction for spiral MRI using fixed-point calculation. *IEEE Trans. Med. Imaging*, 19(7):690 – 698, Jul 2000.