Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Implementation and evaluation of parallel FFT on Engineering and Scientific Computation Accelerator (ESCA) architecture*

Dan WU†, Xue-cheng ZOU, Kui DAI†‡, Jin-li RAO, Pan CHEN, Zhao-xia ZHENG

(*Department of Electronic Science & Technology, Huazhong University of Science and Technology, Wuhan 430074, China*)

†E-mail: dandan58wu@gmail.com; josh.maxview@gmail.com

Received Jan. 26, 2011; Revision accepted July 26, 2011; Crosschecked Nov. 4, 2011

Abstract: The fast Fourier transform (FFT) is a fundamental kernel of many computation-intensive scientific applications. This paper deals with an implementation of the FFT on the accelerator system, a heterogeneous multi-core architecture to accelerate computation-intensive parallel computing in scientific and engineering applications. The Engineering and Scientific Computation Accelerator (ESCA) consists of a control unit and a single instruction multiple data (SIMD) processing element (PE) array, in which PEs communicate with each other via a hierarchical two-level network-on-chip (NoC) with high bandwidth and low latency. We exploit the architecture features of ESCA to implement a parallel FFT algorithm efficiently. Experimental results show that both the proposed parallel FFT algorithm and the ESCA architecture are scalable. The 16-bit fixed-point parallel FFT performance of ESCA is compared with a published work to prove the superiority of the mapping algorithm and the hardware architecture. The floating-point parallel FFT performances of ESCA are evaluated and compared with those of the IBM Cell processor and GPU to demonstrate the computing power of the ESCA system for high performance applications.

Key words: Fast Fourier transform (FFT), Multi-core, Parallel computing, SIMD
doi:10.1631/jzus.C1100027 **Document code:** A **CLC number:** TP302.7

1 Introduction

The high performance computing (HPC) system has long been an infrastructure of scientific research and engineering computation. One way to construct an HPC system is to employ the 'host-processor(s)+co-processor(s)' structure, in which the host-processor takes charge of task partitioning and scheduling while the co-processor deals with numerous parallel computing. The on-chip multiprocessor or CMP, as a strong candidate to continue Moore's law, is the mainstream to build the co-processor, such as IBM PowerXcell in Roadrunner (Barker *et al.*, 2008), ClearSpeed CSX600 (Nishikawa *et al.*, 2007), and GPU (Owens *et al.*, 2008).

In this paper, we propose the Engineering and Scientific Computation Accelerator (ESCA) architecture to accelerate computation-intensive parallel programs in scientific and engineering fields and to act as a co-processor in the HPC system. ESCA relies on the trend toward hierarchical design styles with replicated components, and contains a control unit (CU) that controls a set of processing elements (PEs). To simplify the control logic and achieve higher efficiency gains, we choose the single instruction multiple data (SIMD) architecture to implement the PE array, in which PEs are connected by a hierarchical network-on-chip (NoC) with low

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (Nos. 60973035 and 60976027) and the Natural Science Foundation of Hubei Province, China (No. 2010CDB02705)

communication latency and high bandwidth. Different levels of parallelism can be exploited in the ESCA-based system: (1) At the system level, ESCA chips can execute individual application programs in a multiple instruction multiple data (MIMD) mode; (2) PEs of the ESCA chip execute at the same step in SIMD mode; (3) The function units in each PE are all pipelined and support vector instructions, and the subword-parallel processing is also supported to provide effective extensions for multimedia applications.

As one of the most widely used algorithms in scientific computing, the fast Fourier transform (FFT) can reduce the time complexity of an N-point complex sequence Fourier transform from $O(N^2)$ to $O(N\log_2 N)$. FFT has a large degree of parallelism in each stage of the computation and its implementation has been well studied in various parallel architectures (Swarztrauber, 1984; Agarwal *et al.*, 1994; Cvetanovic, 1987; Takahashi, 2000; 2002). However, the best sequential and parallel FFT implementations vary significantly on different platform architectures. A high efficient FFT algorithm mapping on a specific architecture may perform poorly on other architectures.

The purpose of this paper is therefore not to map an existing FFT algorithm onto the ESCA architecture, but to propose a parallel FFT algorithm by fully using the inherent parallelism of SIMD and the high efficient broadcast and permutation network (BPN) (Wu *et al.*, 2010) within it. With the system architecture described in Section 2, the host processor pre-computes the twiddle factors and preprocesses (bit-reverses) the input data. Then the data is offloaded and $\log_2 N$ stages of butterfly transform are computed using the ESCA co-processor. Not all twiddle factors needed in each stage are stored in each PE. Instead, the twiddle factors are placed regularly and each PE holds only a small portion to save the limited local memory space. During various stages of transform, twiddle factors are broadcasted through the BPN and the communication among PEs is flexible with low hop latency. After the operated data has been prepared, the PEs can execute the multiplications and additions in SIMD to achieve the ultra high parallel computing performance.

Both fixed- and floating-point FFT computation can be executed by ESCA. With elaborately distributing the input data and fully exploiting parallelism, the 16-bit fixed-point parallel FFT performance of ESCA is compared with that of a referenced multi-core NoC to prove the superiority of the mapping algorithm and the hardware architecture. And the floating-point FFT implementation on the ESCA system with 16 and 256 PEs integrated are compared with the IBM Cell processor and GPU, respectively, to show the efficiency of the proposed parallel FFT algorithm and the computing power of the ESCA system for high performance applications.

2 ESCA architecture

The ESCA-based HPC system is constructed as clusters by a collection of computing nodes (CNs). As shown in Fig. 1, each CN consists of a host processor and one or more ESCA co-processor chips. The whole system works in a hybrid computing mode. The host processor can be one of the commercial general-purpose CPUs, executing control-intensive programs and manipulating task partitioning and scheduling. The co-processor ESCA, which integrates multiple PEs on the chip to form a tiled architecture (Taylor *et al.*, 2004) and operates in SIMD mode, accelerates application-specific computing tasks.

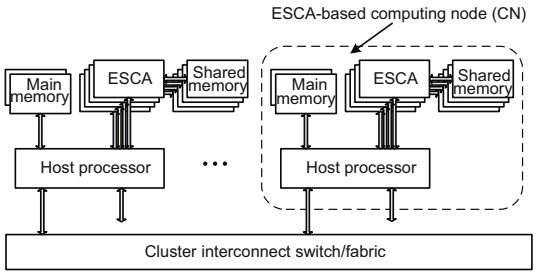


Fig. 1 Conceptual architecture of the ESCA-based high performance computing system

Memory space of the CN is independent and in accord with the distributed shared memory (DSM) model. The host processor is equipped with a large main memory and communicates with ESCA, which acts as an attached memory device, through a memory-like interface. Moreover, several memory chips are tightly-coupled to the ESCA co-processor as the shared memory to alleviate the memory bandwidth bottleneck between the host and ESCA.

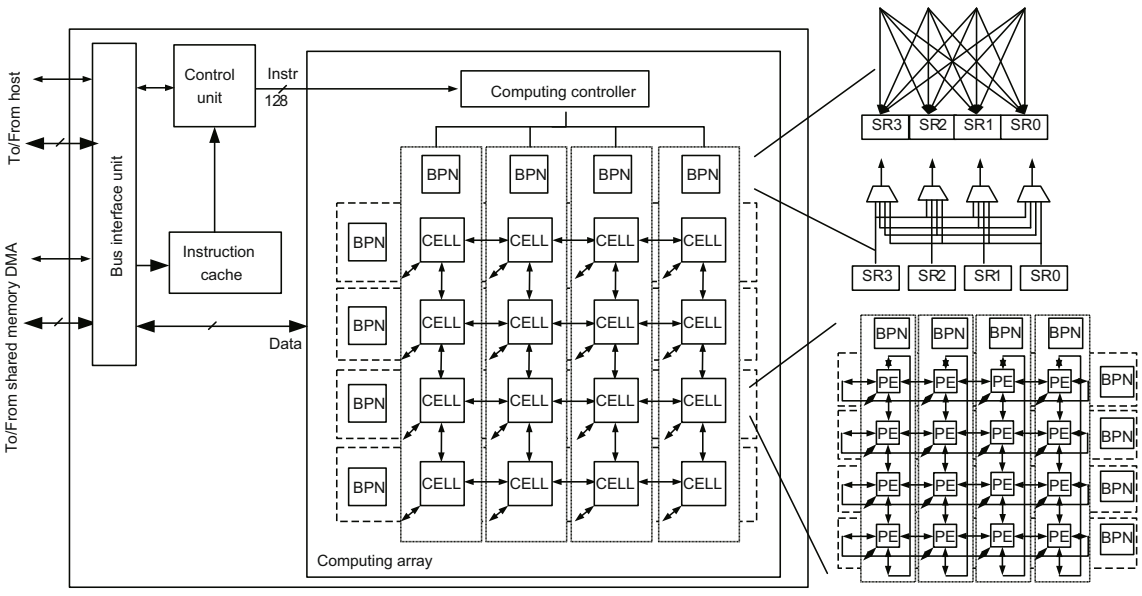
The host can access each ESCA's on-chip

memory space as if accessing an SRAM device, and access the shared memory space via the ESCA chip interface. In addition, the data transfer between main memory and ESCA is accomplished by the host-initiated direct memory access (DMA). And the data transfer between ESCA's on-chip memory and off-chip shared memory or from one ESCA's shared memory to another ESCA's shared memory (within the same CN) is accomplished by the ESCA-initiated DMA.

The programming model of the entire ESCA-based system is the message passing interface (MPI). While the programming of the ESCA-based CN is similar to the Cell broadband engine (Cell BE) (Kahle *et al.*, 2005), the programming model can be based on OpenMP or CellSs (Bellens *et al.*, 2006). The ESCA-based system works in a function-offload mode. The host processor executes complex control-intensive programs, and is typically implemented by equipping a general-purpose programming language, for example, C. The ESCA co-processor executes the computation-intensive programs, in which PE operates in SIMD mode to simplify control logic and exploit data-level parallelism; thus, it can be implemented by equipping with extensions for parallel programming. Two different source files are compiled with the host compiler and ESCA compiler, respectively, and finally linked together to generate an executable program. An application programming interface is provided to implement the communication between the host processor and the ESCA co-processor.

For an application running on this HPC system, the main body of the application data still resides on the main memory of the host processor, and only the computation-intensive accelerating tasks (program's instructions and data) are allocated and offloaded to the memory space of ESCA. Then the host processor will activate the ESCA co-processor to execute those computing tasks, and wait for its completion status, by accessing ESCA's control status register or through the feedback hardwired signals from ESCA. When the ESCA has completed the computing tasks, the host processor will initiate a DMA to transfer result data from the ESCA shared memory to the main memory and continue the thread. Moreover, if the host processor is a multiprocessing system or supports multithreading, it is capable of executing another thread or other threads while waiting for the feedback of the ESCA co-processor.

The dedicated architecture of the ESCA processor (Fig. 2) consists of the bus interface unit (BIU), the control unit (CU), the instruction cache (I-Cache), and the computing array (CA). The BIU module receives and responds to the memory access requests from the host processor or other ESCA co-processors, and generates control signals to access off-chip shared memory. The CU module, the



DMA: direct memory access; BPN: broadcast and permutation network; SR: shared register; PE: processing element

Fig. 2 Overview of ESCA architecture and its hierarchical communication networks

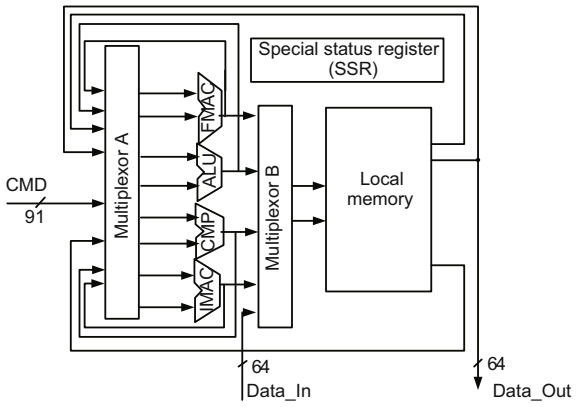
control center of ESCA, can manipulate computation instructions to execute a sequential program or generate the control flow signals and coordinate all parts of ESCA to operate correctly. In addition, the configuration register and the control status registers are located in the CU and can be accessed and set by the host processor. The CA module contains PEs to perform parallel computing. It is constituted by 4×4 CELLs (which are grouped by 4×4 PEs), and a hierarchical two-level BP-Mesh architecture (Wu *et al.*, 2010) serves as the communication fabric of all PEs. Thus, at most 256 PEs can be integrated into one single chip.

Pipelined BPN network in the BP-Mesh architecture offers a low hop latency and high throughput. Its architecture is shown in the upper-right of Fig. 2. Shared registers (SRs) in the BPN are used to pipe the communication data. By setting the control bits of the BPN, data in each PE can be swapped synchronously and stored into the correlated SRs like a crossbar, and then be broadcasted or transferred to destination PEs through a set of multiplexers (MUXs). For a detailed introduction of the BPN and BP-Mesh architecture, one can refer to Wu *et al.* (2010).

PE provides the main computing power of ESCA and offers effective subword-parallel extensions to further support different word-length operations in multimedia fields. PE includes four function units: the floating-point multiply-accumulate, FMAC; the integer multiply-accumulate, IMAC; the arithmetic logic unit, ALU; and the comparison unit, CMP (Fig. 3). The local memory stores the data and provides rapid access to the operands for each computation. All data transfers are explicitly controlled by data transfer instructions, and the active/inactive status of each PE can be flexibly set according to computation instructions. In addition, a dual-buffering mechanism is implemented to overlap the communication and computation.

To sum up, below are listed some characteristics of the ESCA architecture that help to implement the parallel FFT algorithm proposed in Section 3. Table 1 shows its theoretical specification.

1. Vector mode. A vector instruction set (ISA) has been defined for ESCA. By enabling the vector mode, a single instruction can be executed multiple times on multiple data by PEs with a specified vector length and stride. This eliminates the need for instruction scheduling and further alleviates the external shared memory bandwidth pressure for instruction stream fetching.



FMAC: floating-point multiply-accumulate; ALU: arithmetic logic unit; CMP: comparison unit; IMAC: integer multiply-accumulate

Fig. 3 Processing element (PE) structure

Table 1 Specification of the ESCA architecture

Parameter	Value/Description
Instruction width	128-bit
Maximum number of PEs	256
Clock frequency	1 GHz@65 nm
Instruction cache	16–64 KB
Floating-point peak-performance	1024 Gflop/s (single), 512 Gflop/s (double)
NoC bandwidth	2048 GB/s
Maximum local memory of PE	128 KB
Shared memory size	4 GB
Bus width between ESCA and shared memory	64–256 bits (SiP), 512–1024 bits (3D-stacked)
Theoretical peak shared-memory bandwidth	128 GB/s@1 GHz
System memory space	8 TB

2. Subword-parallelism. Subword-parallelism implemented in ESCA offers a micro-SIMD concurrency for each PE. That is, two single-precision floating-point operations or 2/4/8 integer operations with 32/16/8-bits, respectively, can be performed simultaneously in one instruction.

3. Multi-level data broadcast and permutation. ESCA supports multiple levels of data broadcast and flexible data permutation, which improves the performance of the memory system and the bandwidth of on-chip communication. First, a data read from the external shared memory can be broadcasted to all local memories of PEs under the control of one instruction. Second, data in PEs or CELLs can be flexibly transferred through the BPN network in a

swap or broadcast mode. Data exchange and broadcast operations among sub-words are also supported.

3 Implementation of the FFT algorithm on ESCA

In this section, an efficient parallel FFT algorithm implemented on the ESCA based HPC system is described. The parallelism of the algorithm is identified to fully utilize parallel computing capability of the ESCA processor. Thereafter, the effective mapping scheme is presented.

3.1 FFT algorithm background

Let x_1, x_2, ..., x_{N-1} be the sequence of N complex numbers. The discrete Fourier transform (DFT) is defined as

$$X_k = \sum_{n=0}^{N-1} x_n \exp(-\frac{2\pi \mathrm{i}}{N} kn),\ k = 0, 1, \ldots, N-1, \tag{1}$$

where $\exp(2\pi \mathrm{i}/N)$ is a primitive Nth root of unity.

FFT is an efficient algorithm to compute the DFT and its inverse. The most common FFT algorithm is the Cooley-Tukey algorithm (Cooley and Tukey, 1965), which is a divide-and-conquer algorithm that recursively breaks down a DFT of any composite size N ($= N_1 N_2$) into smaller DFTs of sizes N_1 and N_2, along with $O(N)$ multiplications by complex twiddle factors (roots of unity).

Generally speaking, a parallel FFT algorithm for the multi-core processor can be divided into three parts of operations: the pre-processing input data or post-processing results, sequential transform computations on each core's local data with no communication, and the Fourier transform computations with data communication among cores.

Two kinds of parallelism can be observed: the parallel computing of all cores after all operated data has already been stored in core's local memory, and the parallel butterfly pattern of communication among cores.

3.2 Overview of the FFT algorithm implementation on the ESCA

The basic radix-2 decimation-in-time (DIT) FFT of data point N (which is a power of 2) is chosen, since it can be simply implemented on the ESCA. Though fewer data exchange stages are needed, the communication overhead for data exchanging among PEs is much higher for a high radix FFT algorithm.

To take full advantages of ESCA architectural features and reduce the off-chip communication as far as possible, the data distribution among the local memories of all PEs is different from those of other algorithms: data points are divided into N/P groups according to the ratio of the number of data points (N) and the number of PEs (P), and then cyclically distributed into the on-chip local memories. Twiddle factors with a few redundancies are stored in the same way.

With this kind of cyclic distribution, a parallel FFT is implemented on ESCA first with $\log_2 P$ stages of the Fourier transform computations with data communication among PEs, and then with $\log_2(N/P)$ stages of transform computations on each PE's local data without communication.

To unify the algorithm architecture for fixed-point data with different word lengths and floating-point data with different precisions, our proposed FFT algorithm needs an optional sub-word operation step. The detailed implementation of the proposed FFT algorithm on the ESCA based HPC system is described below.

3.3 Data preprocessing and distributing

The first step is preprocessing, in which data is rearranged in bit-reversed order and divided by P blocks. The twiddle factors to be used in each butterfly stage are computed at first. For N-point FFT, the number of different twiddle factors is $N/2$ for its periodicity. The number of twiddle factors needed in each butterfly stage is increased from 1, 2, 4, ..., to $N/2$. These $N-1$ twiddle factors can be accumulated in all stages and prefixed with 0 to form another N-input sequence. The arranged data points and twiddle factors are not fed into PEs sequentially, but in a cyclic mode.

The detailed description of the data distribution procedure is shown in Algorithm 1. After data preprocessing and distributing, each PE holds N/P data points and N/P twiddle factors, occupying $4N/(PS)$ local memory items to hold the real part and the imaginary part, respectively (S is the number of sub-words stored in one register; it is defined as the ratio of register width to FFT data point size).

Algorithm 1 Data preprocessing and distributing

Input:

$\boldsymbol{a} = (a_0, a_1, a_2, \ldots, a_{N-1})$,

$\boldsymbol{w} = (0, W_0, W_1, \ldots, W_{\log_2 N-1})$.

// W_i is the combination of twiddle factors in butterfly

// stage i ($i = 0, 1, \ldots, \log_2 N - 1$).

// $W_i = \{W_N^j | j = k * 2^{n-(i+1)}, k = 0, 1, \ldots, 2^i - 1,$

// $n = \log_2 N$ ($N \geq 2$)$\}$.

Output:

$\boldsymbol{c} = (c_0, c_1, c_2, \ldots, c_{N/(PS)-1})$,

$\boldsymbol{w} = (w_0, w_1, w_2, \ldots, w_{N/(PS)-1})$.

$\boldsymbol{b}$ = bit-reverse($\boldsymbol{a}$);
for $i = 0$ to $(N/(PS) - 1)$ **do**
  for $j = 0$ to $(S - 1)$ **do**
    for $k = 0$ to $(P - 1)$ **do**
      $r = S * i * P + j * P + k$;
      PE[k].c[i].real.subword[j] = b_{r}.real;
      PE[k].c[i].img.subword[j] = b_{r}.img;
      PE[k].w[i].real.subword[j] = w_{r}.real;
      PE[k].w[i].img.subword[j] = w_{r}.img;
    end for
  end for
end for

3.4 $\log_2 P$-stage butterfly for data communication between PEs

Since N data points are cyclically distributed, the distance between any two data points in the same PE for the butterfly transform is larger than N/P; therefore, data must be exchanged among PEs before the transform computation in the first $\log_2 P$ stages. Some novel communication methods have been introduced to transfer $N/(2P)$ data points at one time to reduce the communication overhead and achieve load-balance (Barua *et al.*, 2004; Bahn *et al.*, 2008). In our proposed algorithm, traditional swapping of N/P data points is adopted for a good tradeoff between the synchronous data exchange of all PEs, load-balance, and the elimination of transformed results re-ordering.

Fig. 4 provides an example of FFT communication pattern with 16 PEs in ESCA. At each stage, PEs on the same row or column exchange data with the same pattern. The twiddle factors are broadcasted via the BPN network.

The detailed procedure of the $\log_2 P$-stage butterfly is shown in Algorithm 2.

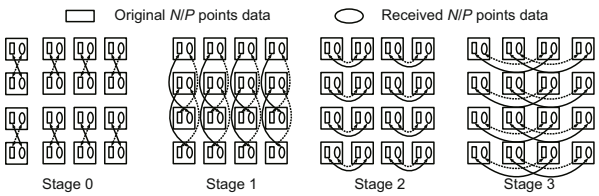


Fig. 4 Communication pattern of FFT with 16 PEs in ESCA. A swapping pair with the corresponding place of original and transferred data is indicated by a solid arrow and a dotted arrow, respectively

Algorithm 2 $\log_2 P$-stage butterfly with data communication between PEs

Input: $\boldsymbol{c} = (c_0, c_1, \ldots, c_{N/(PS)-1})$.
Output: $\boldsymbol{c} = (c_0, c_1, \ldots, c_{N/(PS)-1})$.
for $i = 0$ to $(\log_2 P - 1)$ **do**
  $t = 2^i$;
  for $j = 0$ to $(P - 1)$ **do**
    if $j \bmod t \equiv j \bmod 2t$ **then**
      PE[j] exchanges N/P data points with PE[$j + t$];
    else
      PE[j] exchange N/P data points with PE[$j-t$];
    end if // PE[j] stores these N/P data from
    // $c[N/(PS)]$ to $c[2N/(PS) - 1]$
  end for
  for $k = 0$ to $(t - 1)$ **do**
    if $r \bmod t \equiv k \bmod t$ **then**
      PE[$k + t$] broadcasts twiddle factors to all PE[r];
    end if
  end for // prepare twiddle factors
  for $z = 0$ to $(P - 1)$ **do**
    if $z \bmod t \equiv z \bmod 2t$ **then**
      for $q = 0$ to $(N/(PS) - 1)$ **do**
        PE[z].c[q] = PE[z].c[q]+PE[z].c[q+$N/(PS)$]w;
      end for
    else
      for $q = 0$ to $(N/(PS) - 1)$ **do**
        PE[z].c[q] = PE[z].c[q]–PE[z].c[q+$N/(PS)$]w;
      end for
    end if
  end for // butterfly transform
end for

3.5 $\log_2(N/P)$-stage butterfly in local PE

After the first $\log_2 P$ stages of butterfly transform with data communication among PEs, the operands to be used in the remaining butterfly stages have been stored into the same PE. It can be observed that, with Algorithm 2, all twiddle factors to be exchanged in the following stages have also been stored in the corresponding PE. Hence, the remain-

ing $\log_2(N/P)$ stages of the butterfly transform can be performed by each individual PE in parallel without any NoC communication overhead.

If the width of the internal register of PE is several times the data point size, multiple FFT data points can be packed and stored into one register. With the data distribution strategy described in Section 3.3, the $\log_2 S$ stages of the butterfly are transformed at sub-word level within the registers of each PE first, followed by $\log_2(N/(PS))$ stages of butterfly transform between registers. At the subword level of the butterfly transform, the data points are distributed into different registers and computed in parallel, and then the computation results are recollected. Fig. 5 shows an example of this 'distribute-and-collect' procedure with four sub-words in one register. This procedure efficiently improves the parallelism of all operations.

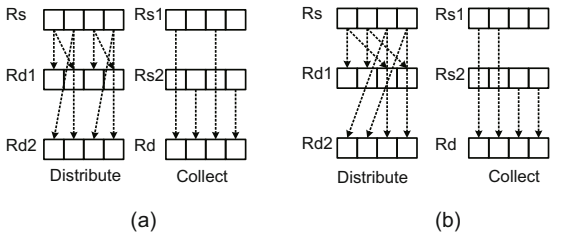


Fig. 5 Distribute-and-collect operation in sub-word mode. (a) ($\log_2 P$)th stage; (b) ($\log_2 P+1$)th stage

Algorithm 3 describes the butterfly computation by each individual PE without any NoC communication overhead. All PEs work in SIMD mode here.

3.6 Implementation of the FFT algorithm on the ESCA based HPC system

In the ESCA-based HPC system, the host processor and the ESCA cooperate to implement the previously proposed FFT algorithm. The host processor performs the bit-reversal operation and computes the twiddle factors. A 'scatter-gather' mechanism is supported by the DMA engine of ESCA, which loads data from the consecutive space of the shared memory and distributes them to the on-chip local memory of PE. It can also gather the computation results from the local memory of PE with a discrete address and store them in the shared memory in a continuous sequence. Thus, the complex data points and twiddle factors can be stored rapidly into the on-chip local memory of ESCA with DMA.

Off-chip communication is much more expensive than on-chip communication, and thus twiddle factors are loaded once. Then the twiddle factors needed in each butterfly stage are transferred to corresponding PEs via the BP-Mesh network. Communication overhead between PEs is a key issue in many parallel FFT algorithms. The flexible permutation mechanism of the BPN supports all PEs in swapping data simultaneously in a pipelined

Algorithm 3 $\log_2(N/P)$-stage butterfly computation in local PE

Input: $\boldsymbol{c}=(c_0,c_1,\ldots,c_{N/(PS)-1})$.
Output: $\boldsymbol{c}=(c_0,c_1,\ldots,c_{N/(PS)-1})$.
// (1) $\log_2 S$ stages butterfly computation within the
// registers of each individual PE
for $i=\log_2 P$ to $(\log_2 P+\log_2 S-1)$ **do**
  for $t=0$ to $(P-1)$ **do**
    for $q=0$ to $(N/(PS)-1)$ **do**
      Distribute the data points from PE[t].c[q] to PE[t].c[q] and PE[t].c[$q+N/(PS)$] in sub-word format, and distribute the twiddle factors to a new memory location;
    end for
  end for // prepare data points and twiddle factors
  for $t=0$ to $(P-1)$ **do**
    for $q=0$ to $(N/(PS)-1)$ **do**
      PE[t].c[q]=PE[t].c[q]+PE[t].c[$q+N/(PS)$]w;
      PE[t].c[$q+N/(PS)$] = PE[t].c[q] − PE[t].c[$q+N/(PS)$]w;
    end for // transform computation
    for $q=0$ to $(N/(PS)-1)$ **do**
      Collect the sub-words from PE[t].c[q] and PE[t].c[$q+N/(PS)$] to PE[t].c[q];
    end for // re-organize results
  end for
end for
// (2) $(\log_2(N/P)-\log_2 S)$ stages butterfly between the
// registers of each individual PE
for $i=(\log_2 P+\log_2 S)$ to $(\log_2 N-1)$ **do**
  $j=i-\log_2 P-\log_2 S$;
  $V=2^j$;
  for $t=0$ to $(P-1)$ **do**
    for $q=0$ to $(N/(PS)-1)$ **do**
      if $q \bmod V \equiv q \bmod 2V$ **then**
        PE[t].c[q] = PE[t].c[q] + PE[t].c[$q+V$]w;
      else
        PE[t].c[q] = PE[t].c[q] − PE[t].c[$q+V$]w;
      end if
    end for
  end for
end for

manner. Thus, the communication overhead here equals only the startup time of the swapping network plus the length of data transferred.

The complex butterfly operation can be decomposed into four multiply-and-add and four multiply-and-subtract operations. And the SIMD and subword parallelism provide a significantly powerful parallel computing capability.

4 Experimental results

This section evaluates the proposed parallel FFT algorithm and the ESCA based HPC system. The experimental environment is first specified. Then the scalability of the proposed parallel algorithm and the ESCA architecture is demonstrated. After that, the performance of the parallel FFT algorithm implemented on the ESCA system is evaluated and then compared with other referenced parallel algorithms and architectures.

4.1 Experimental environment

4.1.1 ESCA prototype chip and platform

The ESCA prototype chip with 16 PEs integrated is taped out with a Chartered 0.13 μm process. The chip works at 500 MHz frequency, and the theoretical peak performance is 32 Gflop/s for single-precision and 16 Gflop/s for double-precision. The BP-Mesh architecture provides a peak bandwidth of 64 GB/s for intra-chip transfers among PEs.

The local memory of PE is made up of register files to provide rapid access. The memory compiler tool is used to generate two-port register file IP and build the 4-read-4-write register file with four banks. Hence, each PE's available local memory is only 4 KB, and an additional arbitration pipeline stage is added between the instruction issue stage and the operands read stage.

For a more detailed introduction of the prototype chip, its chip specification, and the experimental environment, one can refer to Wu *et al.* (2010).

Fig. 6 shows the diagram of the ESCA-based prototype system. The BIU module, CU module, and instruction cache of the ESCA architecture are all implemented on a field-programmable gate array (FPGA) at 250 MHz working frequency. Four CY7C1163V18 4M words 18-bit QDR SRAMs serve as the shared memory at 125 MHz working frequency, and the total capacity is 36 MB with error correcting code (ECC). The shared memory interface provides a theoretical peak bandwidth of 2 GB/s.

4.1.2 ESCA simulator

We also develop a cycle-accurate simulator for the ESCA architecture, which is consistent with the gate-level RTL netlist of ESCA. Its structural diagram is shown in Fig. 7.

The simulator is under the control of a sim_controller module, and simulates the hardware into the abstraction of four parts: the global configuration unit (GCU), the function unit (FCU), the interface unit (IFU), and the memory unit (MU).

The sim_controller executes the simulation environment configurating, simulation scheduling,

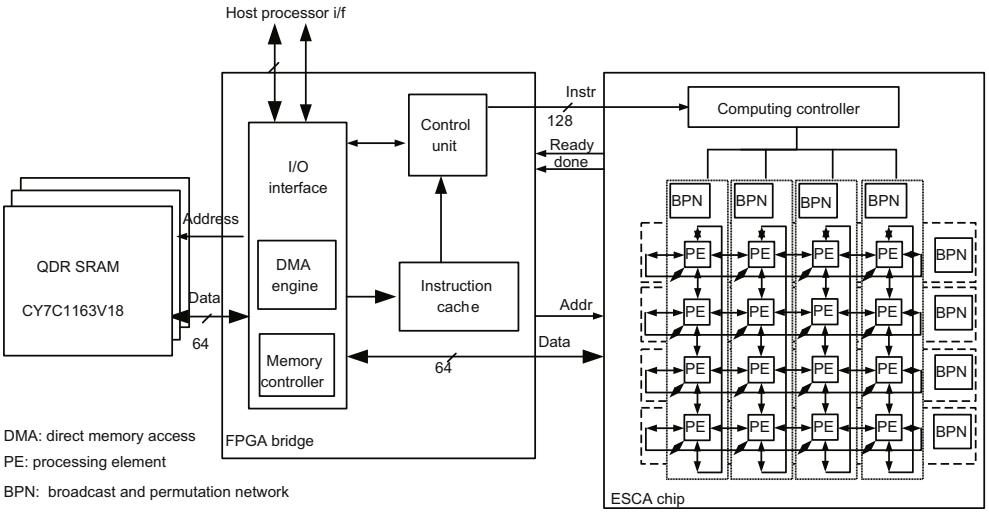


Fig. 6 ESCA-based system prototype test platform (Wu *et al.*, 2010)

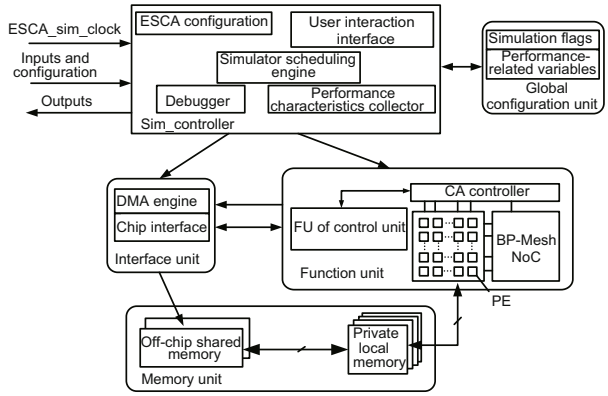


Fig. 7 Simulator structure of the ESCA architecture

debugging, and performance information collecting. Global simulation flags, performance-related simulation variables, and the system configuration table are set and controlled by GCU. FCU simulates all computation parts of the chip and the BP-Mesh network. IFU is responsible for simulating the ESCA interface (with the host processor and off-chip shared memory) and the DMA engine. The off-chip shared memory and on-chip local memory of the PEs are both simulated by the MU module. With an execution-driven mode, the simulator simulates the input binary program codes and generates the computation results, as well as the performance simulation results.

All parts of the simulator have been packed into sub-modules and can be configurated separately. Thereby, the scalability and performance of the system could be flexibly evaluated by configuring the ESCA with various numbers of PEs, memory access latencies, memory bus-widths, and memory capacities.

4.1.3 Programming

The proposed parallel FFT algorithm is coded in the vector instruction set of the ESCA and employed by hand optimization to achieve better performance. As the computation time of twiddle factors is just one time overhead of the host processor, it is excluded from the total time count. Assuming that the source data and programs are located in the shared memory of the ESCA chip before computation, the FFT transform results are transferred back to the shared memory.

4.2 Scalability

Any size of FFT application can be implemented on the ESCA based HPC system using the proposed parallel algorithm. To focus on the FFT transform computation, we exclude the DMA transfer time of the source data input and the final transformed output. Table 2 shows the wall clock time for different sizes of FFT applications implemented on the ESCA system with different PE configurations. The data point of FFT here is double-precision floating-point with a complex value.

Table 2 Computation time for different sizes of FFT

FFT size	Computation time (μs)		
	4×4	8×8	16×16
64	1.032	0.744	0.744
128	1.692	1.016	0.952
256	3.056	1.436	1.088
512	5.956	2.256	1.364
1024	12.184	3.940	1.864
2048	**69.498**	7.480	2.844
4096	**208.558**	14.988	4.848
8192	**552.466**	**128.338**	9.008
16 384	**846.870**	**327.678**	17.816

It can be observed that the proposed parallel FFT algorithm, as well as the ESCA architecture, has excellent scalability with the increased number of PEs for larger FFT applications. By using the unified mapping methodology, an 8×8 PE array requires less time to complete the 64-point parallel FFT computation than a 4×4 array, since the pipeline stage of local-level BPN is shorter than that of the FMAC and the butterfly computation time is the dominant factor in performance at this scale of FFT application.

It can also be seen that different sizes of FFT applications have a lower-bound configuration requirement of the number of PEs with fixed local memory to best exploit data locality. Up to 64 double-precision complex data points and twiddle factors can be handled in each PE with a 4 KB local memory. When the scale of the FFT exceeds the local memory limit, the performance drops rapidly, and the intermediate computing results have to be transferred between on-chip memory and off-chip shared memory, causing a large communication overhead.

This phenomenon is emphasized by the highlighted items in Table 2. When the number of data

points exceeds 1024 for the 4×4 configuration of PEs or 4096 for the 8×8 configuration, the transform time is increased dramatically due to the limited off-chip shared memory bandwidth (2 GB/s). Though the dual-buffering mechanism is adopted, the inter-chip communication overhead is higher than that of the on-chip FFT transform computation and becomes the dominant factor.

4.3 Performance evaluation and comparison

The experimental results of the proposed parallel FFT algorithm implemented on the 16-core ESCA prototype platform are evaluated and analyzed. Then the experimental results regarding the FFT performance evaluated by the ESCA simulator based on 16- and 256-core configurations are compared with those of some referenced architectures.

4.3.1 Performance evaluation of the 16-core ESCA prototype

1. Fixed-point FFT performance

Table 3 compares the performance of fixed-point FFT implemented on the ESCA prototype platform with a referenced 16-core NoC (Bahn *et al.*, 2008) in terms of cycle count. Data points and twiddle factors are represented in 16-bit real part and 16-bit imaginary part in 2's complement format.

Table 3 Fixed-point FFT performance comparison in terms of cycle count

FFT size	Cycle count	
	NoC*†	ESCA‡
64	2680	496
128	3360	812
256	4788	1468
512	8760	2836
1024	16 718	5692
2048	33 560	11 652
4096	68 057	24 076

* Taken from Bahn *et al.* (2008). † 4×4 mesh; ‡ 4×4 Torus+BPN

With the flexibility of BPN network and the subword parallelism for a fine-grained size of data in PE (which concurrently operates with four 16-bit data points), ESCA implemented with the proposed FFT algorithm can significantly accelerate the fixed-point Fourier transform.

2. Floating-point FFT performance

We use the FFTW performance metric (benchFFT, 2003) for the floating-point evaluation. The results are normalized by MFLOPS as a scaled version of the speed:

$$\text{MFLOPS} = \frac{5N\log_2 N}{\text{time for one FFT in } \mu\text{s}}, \qquad (2)$$

where N is the total number of FFT data points.

Fig. 8 illustrates both single- and double-precision floating-point performance and hardware efficiency of FFT implementation on the ESCA prototype platform with 64 to 16K complex input points. Hardware efficiency is the ratio of practical performance to theoretical peak performance. Both the source and output of the FFT application are stored in the off-chip shared memory.

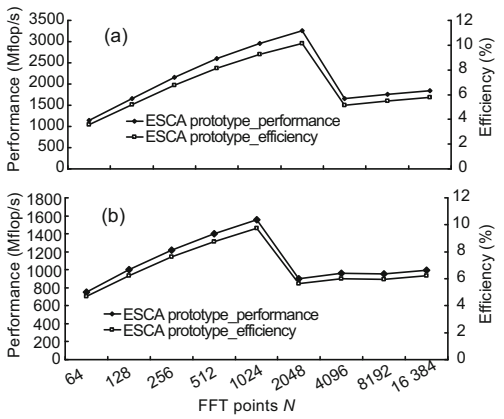


Fig. 8 Floating-point performance and hardware efficiency of the 16-core ESCA prototype platform at different input sizes of FFT. (a) Single-precision; (b) Double-precision

The peak FFT performance of ESCA is 3250.6 Mflop/s for single-precision and 1556.8 Mflop/s for double-precision, with the maximum efficiency of 10.16% and 9.73%, respectively. However, when the input size of FFT exceeds 2K for single-precision or 1K for double-precision, the performance degrades suddenly, causing its efficiency to drop.

The reasons for the abrupt inflexion of the FFT performance curve and low efficiency of the prototype platform are the limited on-chip local memory capacity and low bandwidth of the off-chip shared memory.

To estimate the FFT performance of the 16-core ESCA with a larger capacity of local memory, we configure the ESCA simulator wherein each PE holds 128 KB local memory and maintains the shared memory bus width at 64-bits with 125

MHz working frequency. Using the double-precision 1D FFT implementation on ESCA as the example, Fig. 9 shows that with a much larger local memory, the FFT performance (the curve labeled 'ESCA_128K_64_performance') now also increases steadily compared to the original prototype platform (the curve labeled 'ESCA_4K_64_performance'). However, the practical peak performance does not exceed 2000 Mflop/s, with a maximum efficiency of only 12.46%. The low shared memory bandwidth becomes the root cause and the inter-chip data transfer even occupies more than 50% of the total execution time.

We configure the ESCA simulator with 128 KB local memory and 256-bit shared memory bus-width, and simulate the shared memory as 250 MHz DDR to provide 256-bit data per ESCA clock period. The corresponding performance is shown as the curve labeled 'ESCA_128K_256_performance' in Fig. 9. The larger the memory bus-width, the shorter the data transfer time. The peak FFT performance can reach 4099 Mflop/s at 16K input size with a maximum efficiency of 25.62% under this configuration.

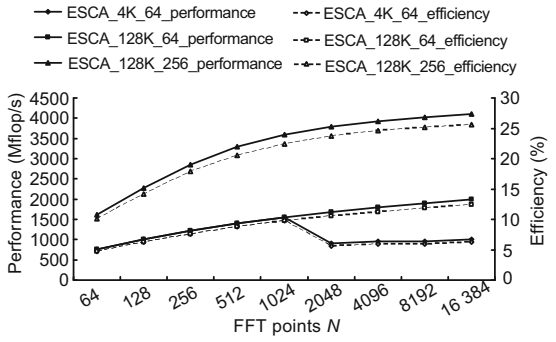


Fig. 9 Double-precision floating-point performance and efficiency of 16-core ESCA with different configurations of local memory, shared memory bus width, and shared memory working frequency

4.3.2 Performance comparison of ESCA architecture with other multi-core architectures

Based on the floating-point performance analysis of ESCA in the previous section, we go on to present the experimental results of the proposed parallel FFT algorithm implemented on ESCA and compare them with those of some referenced architectures, to demonstrate the computing power of ESCA architecture for high performance computing applications. The IBM Cell processor and GPU are selected as the performance comparison counterparts.

Cell BE (IBM, 2005; Williams *et al.*, 2006) architecture is a multi-core processor consisting of one power processor element (PPE) and eight synergistic processing elements (SPEs) designed specifically for high-performance numerical computations. The first generation of the Cell processor has been used in Sony's PlayStation 3 and IBM's QS20 Cell Blades. The IBM PowerXCell 8i processor, the new implementation of the Cell processor, includes an enhanced double-precision unit on the SPEs. Thus, the peak double-precision performance of PowerXCell 8i has been greatly improved. The key architectural characteristics of the PowerXCell 8i (Kistler *et al.*, 2009) are listed in Table 4.

Table 4 Architectural characteristics of the IBM Cell processor and ESCA processor

Parameter	Value/Description	
	PowerXCell 8i	ESCA
Configuration	1 PPE, 8 SPEs	4×4 PEs
Clock frequency	3.2 GHz	1 GHz
Local storage	256 KB	128 KB
NoC peak bandwidth	204.8 GB/s (EIB)	128 GB/s (BP-Mesh)
Memory bandwidth	25.6 GB/s	32 GB/s
Peak single-precision performance	204.8 Gflop/s (8SPEs)	64 Gflop/s
Peak double-precision performance	102.4 Gflop/s (8SPEs)	32 Gflop/s
Platform	IBM QS22	ESCA simulator

Table 4 also lists the architectural characteristics of ESCA. For impartial comparison, parameters of the ESCA architecture are selected with the following considerations: (1) 16-core ESCA is used for comparison with the 8-SPE configuration of the Cell processor; (2) local memory of each PE is configured as 128 KB to make total on-chip local storage capacity equal that of the Cell; (3) the working frequency is configured as 1 GHz for an advanced manufacture process (for instance, 65 nm); and, (4) shared memory is simulated as DDR2 SDRAM, and the bus-width between the ESCA processor and shared memory is 256-bits.

FFTW (Frigo and Johnson, 2005; 2007) is one of the most popular and one of the fastest implementations of the FFT algorithm. It can be used to compute different types, precisions, and input sizes of FFT on various processors. The PowerXCell 8i

specific benchmark suite of the FFTW based on the IBM QS22 Blade was provided by the Joint Cell Competence Center (JCCC) and the performance values can be obtained from the website of JCCC (Joint Cell Competence Center, 2009).

We compare both the single- and double-precision FFT performances of these two architectures, as well as their corresponding hardware efficiencies, with 1K to 128K complex input points (Fig. 10). The peak performance of 16-core ESCA architecture is 16.17 Gflop/s for single-precision at 64K input size and 8.31 Gflop/s for double-precision at 32K input size. With small input data sets, the practical performance of the ESCA architecture is superior to that of the IBM QS22. Though the peak performance of the 16-core ESCA architecture is inferior to that of the Cell processor, its hardware efficiency is much higher than its counterpart.

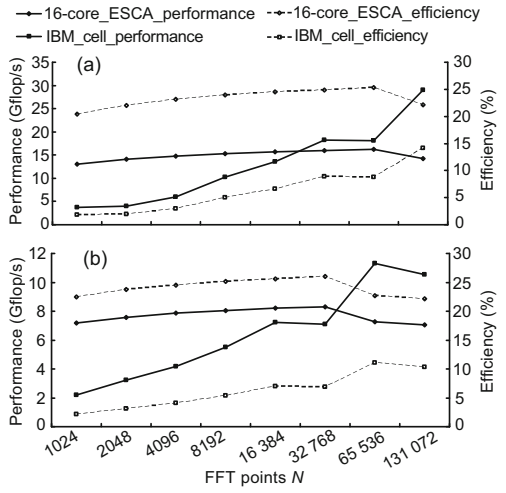


Fig. 10 Comparison of floating-point performance and efficiency between 16-core ESCA architecture and the IBM PowerXCell 8i processor at different input sizes of FFT. The performance results of PowerXCell 8i on IBM QS22 are from JCCC FFTW website (Joint Cell Competence Center, 2009). (a) Single-precision; (b) Double-precision

Next we evaluate the floating-point performance that may be attained by 256-core ESCA. GPU is selected as the counterpart for its tremendous computing power and as a computing-accelerator of the general-purpose graphics processing unit (GPGPU) architecture in the HPC fields. The state-of-the-art in GPU design is represented by NVIDIA's Fermi architecture, which is built in the Tesla 20-series GPU supercomputing processors. The key architectural characteristics of the Tesla C2050 workstation (NVIDIA, 2009) are listed in Table 5.

Table 5 Architectural characteristics of GPU and ESCA architecture

Parameter	Value/Description	
	Tesla 20-series GPU	ESCA
Number of cores	448	256
Clock frequency	1.15 GHz	1 GHz
Peak single-precision performance	1030 Gflop/s	1024 Gflop/s
Peak double-precision performance	515 Gflop/s	512 Gflop/s
Memory bandwidth	144 GB/s (GDDR5)	128 GB/s (DDR2)
Platform	Tesla C2050	ESCA simulator

Again, for impartial comparison, the parameters of the ESCA architecture are configured with a maximum of 256 PEs on the chip with 16 KB local memory for each PE. The shared memory is simulated as DDR2 SDRAM, and the memory bus width is configured to 1024-bits for a future 3D-stacked VLSI package (Deng and Maly, 2010).

The floating-point performance results of Tesla C2050 are obtained from its benchmark results on the cuFFT 3.1 library (NVIDIA, 2010), and we consider its best FFT implementation performance with ECC off.

Fig. 11a shows the single-precision performance and efficiency comparison between 256-core ESCA and Tesla C2050 with 1K to 256K complex input points. The peak single-precision performance of 256-core ESCA reaches 240.72 Gflop/s at 128K input size, with a maximum efficiency of 23.5%.

The corresponding double-precision performance and efficiency comparison between these two processors is shown in Fig. 11b. The peak double-precision FFT performance of the ESCA architecture reaches 122.56 Gflop/s at 64K input size with a maximum efficiency of 23.9%, which is competitive to that of Tesla C2050.

By employing the unified map strategy, the data needs more pipeline stages to be transferred through the higher level of NoC for global communication, and the vector length of PE computation instruction is relatively short, with fewer data points for each PE. Thus, the performance of 256-core ESCA architecture is inferior to that of Tesla C2050 at a small input size.

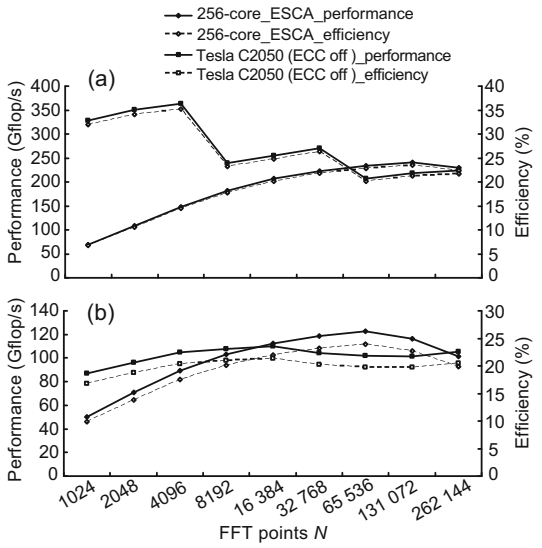


Fig. 11 Comparison of floating-point performance and efficiency between 256-core ESCA and Tesla C2050 at different input sizes of FFT. The performance results of Tesla C2050 are from Tesla C2050 Performance Benchmarks with cuFFT 3.1 (NVIDIA, 2010). (a) Single-precision; (b) Double-precision

5 Conclusions and future work

In this paper, we present a multi-core processor ESCA to accelerate computation-intensive scientific and multimedia applications. An efficient FFT algorithm is proposed to exploit the architecture features of ESCA and implemented on the ESCA to evaluate its scalability and performance for various FFT input sizes and different data types. The scalability of the proposed parallel FFT algorithm, as well as the proposed multi-core architecture of ESCA, has been verified. The performance of fixed- and floating-point complex FFT has been evaluated and compared with those of other referenced high performance FFT implementations. ESCA is superior for the implementation of 16-bit fixed-point parallel FFT, compared to the referenced multi-core NoC, in terms of cycle count. The comparison between ESCA, the IBM Cell processor, and GPU in terms of the implementation of the floating-point FFT, has demonstrated the efficiency of the proposed parallel FFT algorithm and the ESCA system.

There are still several avenues for future work. All twiddle factors are stored in the local memory of each PE and need to be transferred during the butterfly transform in the proposed parallel FFT algorithm. Although ESCA has a powerful BPN network to support data broadcast and permutation, twiddle factors transfer would increase the communication overhead, and the storage of them occupies the limited local memory space. It is possible to add a broadcasting frame buffer (FB) to hold the twiddle factors and broadcast the required ones in each stage on-the-fly to PEs.

Acknowledgements

We thank the following members joining in the ESCA project for their great contribution to the building of the prototype system and function verification: Ming LI, Mian DONG, Cheng-nuo DENG, Dan-dan SONG, Guang-heng HU, and Wei HUANG.

References

Agarwal, R.C., Gustavson, F.G., Zubair, M., 1994. A High Performance Parallel Algorithm for 1-D FFT. Proc. Supercomputing, p.34-40. [doi:10.1109/SUPERC.1994.344263]

Bahn, J.H., Yang, J., Bagherzadeh, N., 2008. Parallel FFT Algorithms on Network-on-Chips. 5th Int. Conf. on Information Technology: New Generation, p.1087-1093. [doi:10.1109/ITNG.2008.55]

Barker, K.J., Davis, K., Hoisie, A., Kerbyson, D.J., Lang, M., Pakin, S., Sancho, J.C., 2008. Entering the Petaflop Era: the Architecture and Performance of Roadrunner. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, p.1-12. [doi:10.1109/SC.2008.5217926]

Barua, S., Thulasiram, R.K., Thulasiraman, P., 2004. Improving Data Locality in Parallel Fast Fourier Transform Algorithm for Pricing Financial Derivatives. Proc. 18th Int. Parallel and Distributed Processing Symp., p.235-240. [doi:10.1109/IPDPS.2004.1303283]

Bellens, P., Perez, J.M., Badia, R.M., Labarta, J., 2006. CellSs: a Programming Model for the Cell BE Architecture. Proc. ACM/IEEE SC Conf., p.5-15. [doi:10.1109/SC.2006.17]

benchFFT, 2003. FFT Benchmark Methodology. Available from http://www.fftw.org/speed/method.html [Accessed on Jan. 16, 2011].

Cooley, J.W., Tukey, J.W., 1965. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, **19**(90):297-301. [doi:10.1090/S0025-5718-1965-0178586-1]

Cvetanovic, Z., 1987. Performance analysis of the FFT algorithm on a shared-memory parallel architecture. *IBM J. Res. Dev.*, **31**(4):435-451. [doi:10.1147/rd.314.0435]

Deng, Y.D., Maly, W.P., 2010. 3-Dimensional VLSI: a 2.5-Dimensional Integration Scheme. Tsinghua University Publishing House, Beijing, China, p.144-158. [doi:10.1007/978-3-642-04157-0_7]

Frigo, M., Johnson, S.G., 2005. The design and implementation of FFTW3. *Proc. IEEE*, **93**(2):216-231. [doi:10.1109/JPROC.2004.840301]

Frigo, M., Johnson, S.G., 2007. FFTW on the Cell Processor. Available from http://www.fftw.org/cell/index.html [Accessed on Jan. 16, 2011].

IBM, 2005. The Cell Architecture. Available from http://www.research.ibm.com/cell/home.html [Accessed on Jan. 16, 2011].

Joint Cell Competence Center, 2009. FFT Performance Results of IBM QS22 Cell Blade. Available from http://cell.icm.edu.pl/index.php/FFTW_on_Cell [Accessed on May 10, 2011].

Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D., 2005. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, **49**(4-5):589-604. [doi:10.1147/rd.494.0589]

Kistler, M., Gunnels, J., Brokenshire, D., Benton, B., 2009. Programming the Linpack benchmark for the IBM PowerXCell 8i processor. *Sci. Progr.*, **17**(1-2):43-57. [doi:10. 3233/SPR-2009-0278]

Nishikawa, Y., Koibuchi, M., Yoshimi, M., Miura, K., Amano, H., 2007. Performance Improvement Methodology for ClearSpeed's CSX600. Int. Conf. on Parallel Processing, p.77. [doi:10.1109/ICPP.2007.66]

NVIDIA, 2009. High Performance Computing — Supercomputing with Tesla GPUs. Available from http://www.nvidia.com/object/tesla_computing_solutions.html [Accessed on May 10, 2011].

NVIDIA, 2010. Tesla C2050 Performance Benchmarks. Available from http://nvworld.ru/files/articles/calculations-on-gpu-advantages-fermi/fermipeformance.pdf [Accessed on May 10, 2011].

Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Philips, J.C., 2008. GPU computing. *Proc. IEEE*, **96**(5):879-899. [doi:10.1109/JPROC.2008.917757]

Swarztrauber, P.N., 1984. FFT algorithms for vector computers. *Parall. Comput.*, **1**(1):45-63. [doi:10.1016/S0167-8191(84)90413-7]

Takahashi, D., 2000. High-Performance Parallel FFT Algorithms for the HITACHI SR8000. Proc. 4th Int. Conf./Exhibition on High Performance Computing in the Asia-Pacific Region, p.192-199. [doi:10.1109/HPC.2000.846545]

Takahashi, D., 2002. A blocking Algorithm for Parallel 1-D FFT on Shared-Memory Parallel Computers. 6th Int. Conf. of Applied Parallel Computing, Advanced Scientific Computing, p.380-389.

Taylor, M.B., Psota, J., Saraf, A., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., Agarwal, A., Lee, W., Miller, J., *et al.*, 2004. Evaluation of the Raw Microprocessor: an Exposed-Wire-Delay Architecture for ILP and Streams. Proc. 31st Annual Int. Symp. on Computer Architecture, p.2-13. [doi:10.1109/ISCA.2004.1310759]

Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K., 2006. The Potential of the Cell Processor for Scientific Computing. Proc. 3rd Conf. on Computing Frontiers, p.9-20. [doi:10.1145/1128022.1128027]

Wu, D., Dai, K., Zou, X.C., Rao, J.L., Chen, P., 2010. A High Efficient on-Chip Interconnect Network in SIMD CMPs. 10th Int. Conf. on Algorithms and Architecture for Parallel Processing, p.149-162. [doi:10.1007/978-3-642-13119-6_13]