Data Management: The Key to Implementing Efficient On-Chip Many-Core Processors (Post-print)

Authors: Zhou Yongbin, Zhang Junchao, Zhang Shuai, Zhang Hao

Date: 2016-11-02T00:00:00+00:00

Abstract

With the increasing demand for high-performance computing, on-chip many-core (many-core) has become the development direction of future processor architectures. Currently, most many-core processor prototypes adopt a tiled topology structure, connected through an on-chip network. When facing communication-intensive applications, the throughput (throughput) or latency (latency) of the on-chip network often becomes a bottleneck that constrains system performance. By providing architectural support for on-chip data storage and communication management, we have achieved efficient on-chip data communication in many-core processors. The main contributions of this paper include: (1) implementing a tiled many-core processor architecture that supports program-controlled on-chip data storage management and data transfer management; (2) proposing that the asynchronous data block transfer mechanism is an effective method for tolerating latency in 2D Mesh on-chip networks. Finally, we evaluated the Fast Fourier Transform (FFT) on the many-core processor, achieving 43.9Gflops computational performance with a computational efficiency of 22.9%.

Full Text

Data Management: The Key to Efficient On-Chip Many-Core Processors

Authors: Yongbin Zhou, Junchao Zhang, Shuai Zhang, Hao Zhang
Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences

Abstract

As high-performance computing demands continue to grow, on-chip many-core architectures have emerged as the future direction for processor design. Most current many-core processor prototypes adopt a tiled topology interconnected via on-chip networks. For communication-intensive applications, the throughput or latency of these on-chip networks often becomes the bottleneck limiting system performance. By providing architectural support for on-chip data storage and communication management, we have achieved efficient on-chip data communication in many-core processors. The main contributions of this paper include: (1) implementing a tiled many-core processor architecture that supports programmable on-chip data storage management and data transfer management; and (2) proposing asynchronous data block transfer as an effective method for tolerating latency in two-dimensional Mesh on-chip networks. Finally, we evaluate the Fast Fourier Transform (FFT) on our many-core processor, achieving 43.9 Gflops computational performance with a computational efficiency of 22.9%.

Keywords: Many-core; Godson-T; Fast Fourier Transform; Computation and Communication Overlap

1 Introduction

The advancement of semiconductor technology has made power consumption and heat dissipation increasingly severe challenges in microprocessor design. On-chip multi-core processors (CMP) have gradually replaced single-core processors as the mainstream product in general-purpose computing. In response to growing high-performance computing demands, on-chip many-core processors have also become a key research focus. As wire delay has become a non-negligible factor affecting system performance, cost, and power consumption [**?**], many traditional design concepts are no longer applicable to many-core processors. For example, NUCA cache (Non-Uniform Cache Architecture) [**?**] offers better scalability compared to traditional UCA cache (Uniform Cache Architecture) structures. Simultaneously, on-chip networks [**?**, **?**] based on topologies such as rings and Meshes have replaced traditional shared buses as the interconnection fabric for many-core processors.

While achieving satisfactory theoretical peak computational performance, the development of many-core processors has also intensified pressure on on-chip storage and networks. Network congestion caused by excessive on-chip communication loads frequently limits the effective computational performance improvements in many-core processors. The Fast Fourier Transform (FFT), a fast algorithm for Discrete Fourier Transform, is widely used in communications, digital signal and image processing, bioinformatics, and other fields, and has been extensively studied [**?**].

The Cell Broadband Engine (Cell BE) [**?**], developed jointly by IBM, Sony, and Toshiba, integrates one PPE (Power Processing Element), eight SPEs (Syner-

gistic Processing Elements), one Memory Interface Controller (MIC), and one bus controller via four ring buses. The PPE is a dual-threaded, in-order processor core with 32KB L1 cache and 512KB L2 cache. Each SPE includes 256KB Local Store Memory (LS) and a Memory Flow Controller (MFC). Data transfers between local memory and system memory are accomplished through the DMA engine integrated in the MFC. S. Williams evaluated single-precision floating-point one-dimensional FFT on the Cell processor, achieving an effective performance of 41.9 Gflops [**?**].

Cyclops-64 [**?**], a many-core processor under research at IBM, contains 80 dual-threaded in-order execution units. Most notably, Cyclops-64' s on-chip memory system does not employ traditional cache structures but instead uses 160 on-chip SRAMs to form a two-level storage structure: global memory (GM) and scratchpad memory (SP). Additionally, Cyclops-64 utilizes a crossbar to connect execution units, global memory, and scratchpad memory. Long Chen et al. optimized single-precision floating-point one-dimensional FFT on Cyclops-64, achieving a performance of 20.7 Gflops [**?**].

Recent studies [**?**] have shown that on-chip data management can effectively reduce on-chip data communication, alleviate network congestion, and improve program performance. In this paper, we first propose a many-core processor architecture called Godson-T Pro that supports on-chip data storage and transfer mechanisms, and then evaluate its performance executing one-dimensional FFT, achieving an effective computational performance of 43.9 Gflops and computational efficiency (effective performance/peak performance) of 22.9%. The main contributions of this paper include:

1. Based on a many-core processor architecture, we propose a hybrid on-chip storage structure that combines caches and scratchpad memory, enabling on-chip data storage management and data transfer management through a Data Transfer Agent (DTA).

2. On a two-dimensional Mesh on-chip network employing packet-switching protocols, we evaluate the impact of asynchronous data block transfer mechanisms, demonstrating that computation and communication overlap can effectively tolerate on-chip network latency.

The remainder of this paper is organized into five sections. Section 2 introduces the architecture of the Godson-T Pro many-core processor. Section 3 briefly describes the traditional six-step FFT algorithm. Section 4 presents optimizations of the six-step FFT algorithm on Godson-T Pro. Section 5 describes our experimental environment and analyzes the results. Section 6 summarizes the proposed on-chip data management mechanism based on many-core processor architecture and suggests future research directions.

2 Many-Core Processor Architecture

[Figure 1: see original paper] shows the architecture of Godson-T Pro, a high-

performance computing oriented on-chip many-core processor. Godson-T Pro comprises 64 homogeneous execution tiles, four DDR2 memory controllers, three I/O interfaces, and one on-chip Synchronization Manager (SM). The synchronization manager provides fast and effective hardware support for mutual exclusion and synchronization operations. Godson-T Pro' s on-chip network employs a two-dimensional Mesh structure with packet-switching and X-Y simple routing strategy, where each channel has independent 128-bit bidirectional links.

Each execution tile includes an in-order dual-issue execution unit with one floating-point multiply-add unit and one floating-point add unit. Therefore, at a 1GHz operating frequency, Godson-T Pro' s single-precision theoretical computational performance reaches 192 Gflops. The on-chip memory system consists of a two-level cache structure combined with a programmable scratchpad memory. The shared 4MB L2 cache is distributed across the 64 execution tiles in a NUCA fashion, with each tile containing 8KB private data cache (Dcache), 24KB scratchpad memory, and a 64KB shared L2 cache bank. Private data cache and L2 cache maintain cache coherence through scope consistency [**?**]. Additionally, by allocating independent address spaces for scratchpad memory, data conflicts between private data cache, L2 cache, and scratchpad memory are avoided.

Each execution tile also contains a Data Transfer Agent (DTA) that functions as a coprocessor core. The primary role of the DTA is to perform data block transfers between remote and local scratchpad memory units, as well as between scratchpad memory and L2 cache. [Figure 2: see original paper] illustrates the data block transfer modes supported by the DTA, where each block consists of packets. [FIGURE:2(a)] depicts contiguous data transfer mode; [FIGURE:2(b)] shows block-stride data transfer where packets within each block are contiguous; and [FIGURE:2(c)] shows block-stride data transfer where packets within each block are also spaced at constant packet stride.

The DTA provides programmers with the following API programming interfaces for asynchronous data block transfers:

- `check_dta_status(type)`: Queries the DTA' s working status. Since the processor core does not implement interrupt functionality, a polling approach is used to check whether the DTA has completed its operation. When `type` is `DTA_STATUS`, it queries whether the current DTA operation is complete—new DTA operations can only be initiated when the DTA is idle. When `type` is `BLOCK_STATUS`, it queries the number of blocks already transferred in the current DTA operation.
- `dta_get(src_addr, src_stride, dst_addr, dst_stride, packet_cnt, block_cnt)`: Reads data from remote memory space starting at `dst_addr` and stores it in local scratchpad memory starting at `src_addr`. Supports the data block transfer modes shown in [FIGURE:2(a)] or [FIGURE:2(c)].
- `dta_put(src_addr, src_stride, dst_addr, dst_stride, packet_cnt, block_cnt)`: Sends data from local scratchpad memory starting at

`src_addr` to remote memory space starting at `dst_addr`. Parameters have the same meaning as `dta_get` and support the data block transfer modes shown in [FIGURE:2(a)] or [FIGURE:2(c)].

- `dta_get2(src_addr, src_packet_stride, src_block_stride, dst_addr, dst_packet_stride, dst_block_stride, packet_cnt, block_cnt)`: Reads data from remote memory space starting at `dst_addr` and stores it in local scratchpad memory starting at `src_addr`. Supports the data block transfer mode shown in [FIGURE:2(c)].
- `dta_put2(src_addr, src_packet_stride, src_block_stride, dst_addr, dst_packet_stride, dst_block_stride, packet_cnt, block_cnt)`: Sends data from local scratchpad memory starting at `src_addr` to remote memory space starting at `dst_addr`. Parameters have the same meaning as `dta_get2` and support the data block transfer mode shown in [FIGURE:2(c)].
- `dta_syn_to_l2(void)`: Ensures that all data written to L2 cache has actually entered L2 cache. This operation requires a response from the L2 cache controller to guarantee completion.

3 Six-Step FFT Algorithm

This section provides a brief introduction to the six-step FFT algorithm. For the Discrete Fourier Transform:

$$X_k = \sum_{j=0}^{n-1} x_j \exp(-2\pi i(jk/n)), \quad 0 \le k < n \tag{2.1}$$

Let $\omega = \exp(-2\pi i/n)$. Then:

$$X_k = \sum_{j=0}^{n-1} x_j \omega^{jk} \tag{2.2}$$

If there exist factors n_0 and n_1 such that $n = n_0 \times n_1$, we can express j and k as:

$$j = j_1 n_1 + j_0, \quad k = k_0 n_0 + k_1 \tag{2.3}$$

where $0 \le j_0, k_0 < n_0$ and $0 \le j_1, k_1 < n_1$. We define two-dimensional matrices for x_j and X_k:

$$x_{j_0,j_1} = x_{j_1 n_1 + j_0}, \quad X_{k_1,k_0} = X_{k_0 n_0 + k_1} \tag{2.4}$$

Substituting (2.3) and (2.4) into (2.2) yields:

$$X_{k_1,k_0} = \sum_{j_0=0}^{n_0-1} \sum_{j_1=0}^{n_1-1} x_{j_0,j_1} \omega^{(j_1 n_1 + j_0)(k_0 n_0 + k_1)} \tag{2.5}$$

The computation of (2.5) is divided into two steps. First, compute n_0 FFTs of length n_1:

$$y_{j_0,k_1} = \sum_{j_1=0}^{n_1-1} x_{j_0,j_1} \omega_{n_1}^{j_1 k_1} \tag{2.6}$$

Then perform n_1 FFTs of length n_0 from (2.7):

$$z_{k_1,k_0} = \sum_{j_0=0}^{n_0-1} (y_{j_0,k_1} \omega_n^{j_0 k_1}) \omega_{n_0}^{j_0 k_0} \tag{2.7}$$

Finally, from (2.5) we obtain:

$$X_{k_1,k_0} = z_{k_1,k_0} \tag{2.8}$$

Beginning from (2.5), the FFT is computed through six steps:

1. In (2.6), the $n_0 \times n_1$ matrix (x_{j_0,j_1}) requires multi-column FFT computation. If row-major storage is used in memory, the matrix must be transposed.
2. Perform n_1 independent one-dimensional n_0-point FFT computations on the matrix obtained from step 1. This corresponds to (2.6).
3. Multiply the matrix from step 2 by twiddle factors $\omega_n^{j_0 k_1}$. This corresponds to (2.6).
4. Transpose the matrix from step 3 to an $n_1 \times n_0$ matrix, serving the same purpose as step 1.
5. Perform n_0 independent one-dimensional n_1-point FFT computations on the $n_1 \times n_0$ matrix from step 4. This corresponds to (2.7).
6. Transpose the matrix from step 5 to obtain the $n_0 \times n_1$ matrix X_{k_1,k_0}. This corresponds to (2.8).

In step 2, the commonly used algorithm is the radix-2 FFT algorithm. Its time complexity is $O(n \log n)$, and the computation process contains $\log n$ steps, with each step containing $n/2$ butterfly operations. As shown in [Figure 3: see original paper], each butterfly operation requires one complex multiplication and two complex additions. Therefore, computing a radix-2 FFT requires $(n/2) \log n$ complex multiplications and $n \log n$ complex additions.

For the n-input matrix described above, we consider the simple case where $n_0 = n_1 = \sqrt{n}$, resulting in an input matrix of size $\sqrt{n} \times \sqrt{n}$. Step 2 requires one radix-2 FFT per column, step 5 requires one radix-2 FFT per row, and step 3 multiplies each matrix element by a twiddle factor. Therefore, completing the six-step FFT requires $n \log \sqrt{n}$ complex multiplications and $2n \log \sqrt{n}$ complex additions. Since one complex multiplication is equivalent to 4 real multiplications and 2 real additions, and one complex addition is equivalent to 2 real additions, the total computation is equivalent to $2n \log n$ real multiplications and $3n \log n$ real additions.

4 Optimization of the Six-Step FFT Algorithm

The FFT implementation in SPLASH-2 simplifies the six-step FFT algorithm by assuming $n_0 = n_1 = \sqrt{n}$. On Godson-T, we have studied the optimization and implementation of this algorithm. We assume the processor has p cores, and all FFT data $X[n]$ including its transpose $X^T[n]$ are stored in L2 cache. The L2 cache also stores precomputed twiddle factors ω_n and ω_{n_0}. First, we analyze the storage hierarchy overhead. lists the storage requirements for the six-step single-precision FFT algorithm in private data cache/scratchpad memory and L2 cache when the processor has 64 cores, where $X_p[n]$ represents the matrix elements assigned to each processor core (requiring n/p rows for FFT), and $\omega_{n,p}$ represents the twiddle factors based on n corresponding to the assigned matrix elements.

Storage overhead of the six-step FFT algorithm on Godson-T

Data Size	Private Data Cache/Scratchpad	L2 Cache
4K FFT	128KB	212 (4K)
16K FFT	128KB	214 (16K)
64K FFT	128KB	216 (64K)

4.1 Hiding Matrix Transposition

Steps 1, 4, and 6 of the six-step FFT algorithm all involve matrix transposition, which incurs significant overhead, as we will see in Section 5. If we store X, ω_{n_0}, and ω_n in the processor core' s scratchpad memory, the matrix transposition process can be hidden within the data transfer between L2 cache and scratchpad memory by invoking the DTA programming interfaces `godsont_dta_get` or `godsont_dta_put`. As shown in [Figure 4: see original paper], blocks with the same pattern represent data associated with the same processor core. The processor core performs FFT computations on scratchpad memory elements row by row, with consecutive data blocks in the same row. When these blocks are written to L2 cache, they must be mapped to the same column to achieve transposition, with adjacent stride equal to the storage space occupied by n points.

By using the DTA for data transfer, the six-step FFT algorithm can be implemented as shown in [Figure 5: see original paper], where the transposition operations in steps 1, 4, and 6 are hidden within the communication between processor cores and L2 cache. When the DTA first writes data from scratchpad memory to L2 cache in column order, all participating processor cores must synchronize to ensure that data subsequently fetched from L2 cache is correct.

4.2 Asynchronous Data Block Transfer

The strategy shown in [Figure 5: see original paper] reveals that the DTA and processor core work in a serial fashion. Since there are no data dependencies between rows that the processor core needs to process during n-point FFT, we can further optimize using a computation-communication overlapping programming model through data prefetching with the DTA, as illustrated in [Figure 6: see original paper]. Let the processor core ID be `MyNum` and the number of processors be p. Each processor core performs n-point FFT on columns `(MyFirst : MyLast-1)` of the $n \times n$ matrix X, where `MyFirst = MyNum × n/p` and `MyLast = MyFirst + n/p`. The scratchpad memory stores two columns of matrix X elements (`_x[0]` and `_x[1]`), though the actual number can be determined based on scratchpad size.

[FIGURE:6(a)] shows the DTA first fetching column `[MyFirst]` elements from L2 cache into `_x[0]` while performing transposition, with the processor idle. [FIGURE:6(b)] shows that when all elements of column `[MyFirst]` are in `_x[0]`, the processor core begins FFT on `_x[0]` while the DTA prefetches column `[MyFirst+1]` into `_x[1]`. [FIGURE:6(c)] shows that when column `[MyFirst+1]` elements are fully stored in `_x[1]`, the processor core has completed FFT on `_x[0]` and begins FFT on `_x[1]`, while the DTA stores the transformed `_x[0]` elements into transposed matrix `T[MyFirst]`. [FIGURE:6(d)] shows that when all data in `_x[0]` has been stored in `T[MyFirst]`, the processor core is still performing FFT on `_x[1]`, and the DTA begins prefetching column `[MyFirst+2]` into `_x[0]` (or the processor may be idle if FFT on `_x[1]` is complete). [FIGURE:6(e)] shows that when `T[MyFirst+2]` column elements are fully stored in `_x[0]`, the processor core has completed FFT on `_x[1]` and begins FFT on `_x[0]`, while the DTA stores `_x[1]` elements into transposed matrix `[MyFirst+1]`. [FIGURE:6(f)] shows the DTA writing `_x[1]` elements to transposed matrix `T[MyFirst-1]` while the processor core is idle, with all column FFTs for `(MyFirst, MyLast-1)` completed.

[Figure 6: see original paper] depicts the implementation of steps 1-3 in the algorithm shown in [Figure 5: see original paper]; steps 4-6 can be implemented similarly. Through this pattern, the work of processor cores and the DTA overlaps to some extent, helping to reduce or eliminate communication overhead between processor cores and L2 cache.

Let F_τ be the time (in cycles) for a processor core to complete an n-point FFT, L_τ be the average latency for the DTA to read data from L2 cache, and S_τ be

the average latency for the DTA to write n FFT-processed points to L2 cache. From [FIGURE:6(c)] and [FIGURE:6(d)], we see that to hide L2 cache access latency within computation as much as possible, the following condition must be satisfied:

$$S_\tau < F_\tau \tag{4.1}$$

From Section 2' s introduction, performing an n-point FFT requires $5n \log n$ floating-point operations (not considering bit-reverse overhead or multiplication with twiddle factors after FFT). For single-precision floating-point, the required floating-point operations are $5n \log n$. Assuming ideally the processor core completes 3 floating-point operations per cycle, then $F_\tau = (5/3)n \log n$. Substituting into (4.1) yields:

$$n > \frac{3}{5} \cdot \frac{S_\tau}{\log n} \approx 0.83 \frac{S_\tau}{\log n} \tag{4.2}$$

When the average access latency between processor core and L2 cache satisfies (4.2), communication latency can be completely hidden within the n-point FFT computation.

Further observation of [Figure 6: see original paper] reveals that after matrix transposition via DTA, matrix element positions in matrix X and transposed matrix X^T are identical, as shown in [FIGURE:6(a)] and [FIGURE:6(c)]. During this process, processor cores are completely independent with no contention. However, analyzing steps 4-6 of the optimized six-step FFT algorithm reveals that matrix element positions change between X and X^T, creating potential contention. Nevertheless, when scratchpad memory can store all required data for each processor core, we can prefetch all needed data into respective scratchpad memories in step 4 to avoid contention. Therefore, transposed matrix X^T can share the same address space as X. As shown in , this reduces the FFT algorithm' s L2 cache storage overhead by nearly two-thirds, significantly improving on-chip storage resource utilization.

5 Experimental Methods and Results

Our experiments were conducted on a cycle-accurate, event-driven Godson-T Pro simulator assuming a 1GHz operating frequency. To evaluate the many-core processor' s on-chip data management mechanism, we configured the on-chip memory system as shown in . The first configuration, Godson-T Base, includes only a 64KB L2 cache bank and 32KB private data cache in each execution tile, without scratchpad memory or DTA. The second configuration, Godson-T Pro, includes a 64KB L2 cache bank, 8KB private data cache, 24KB scratchpad memory, and DTA in each execution tile. In our experiments, the traditional parallel one-dimensional FFT algorithm described in Section 3 runs on the Godson-T Base environment, while the optimized one-dimensional FFT

algorithm described in Section 4 runs on the Godson-T Pro environment. Other simulator timing parameters are listed in .

Configuration of Godson-T Base and Godson-T Pro

Component	Godson-T Base	Godson-T Pro
L2 Cache	4MB, 64 banks, 8-way	4MB, 64 banks, 8-way
Private Data Cache	32KB, 4-way	8KB, 1-way
Scratchpad Memory	-	24KB
DTA	-	Yes

Simulator timing parameters

Operation	Latency (cycles)
Private data cache hit	1
Scratchpad access	1
L2 cache hit	12-40
Router node	2
Off-chip memory	60-120

First, we evaluated the overhead of matrix transposition (percentage of time spent transposing relative to total FFT computation time) for steps 1, 4, and 6 of the six-step FFT algorithm on Godson-T Base. As shown in [Figure 7: see original paper], the vertical axis represents the percentage of matrix transposition overhead in total FFT computation time, while the horizontal axis shows the number of active processor cores. As the number of active cores increases, matrix transposition overhead ranges between 17% and 23%, demonstrating the advantage of eliminating matrix transposition through DTA operations.

When on-chip network load becomes heavy, network performance degrades due to congestion [**?**]. When optimizing one-dimensional FFT on Godson-T Pro, we utilize the DTA's asynchronous data block transfer, which overlaps computation with communication but also increases on-chip network load. Therefore, we classified on-chip communication packets into three categories: L2 cache load requests, L2 cache store requests, and L2 cache refill data. We compared the average on-chip network latency for these three request types when running 4K, 16K, and 64K one-dimensional FFTs on both Godson-T Base and Godson-T Pro, as shown in [Figure 8: see original paper]. [FIGURE:8(a)] shows average network latency for L2 cache read requests, [FIGURE:8(b)] for L2 cache write requests, and [FIGURE:8(c)] for L2 cache refill data.

[Figure 8: see original paper] reveals that as the number of processor cores and data size increase, the three message types in the traditional one-dimensional FFT on Godson-T Base show similar average network latency, ranging between

15-20 cycles. However, after optimization on Godson-T Pro, the average network latency for these message types shows significant differences. L2 cache read request latency degrades most severely, increasing 2-3 times compared to the traditional approach and growing noticeably with more active cores. L2 cache refill message latency is least affected, showing little variation with increasing cores and remaining nearly equivalent to Godson-T Base. L2 cache write request latency falls between these extremes, increasing nearly 1 times compared to Godson-T Base. This demonstrates that the asynchronous data block transfer in Godson-T Pro' s data management mechanism does increase on-chip network pressure, but as our performance analysis shows, computation-communication overlap effectively tolerates on-chip network latency.

[Figure 9: see original paper] compares the single-precision floating-point performance of 4K, 16K, and 64K FFTs on Godson-T Base and Godson-T Pro. As the number of active processor cores increases, both schemes show performance improvements. Compared to the traditional one-dimensional FFT on Godson-T Base, the optimized one-dimensional FFT on Godson-T Pro achieves 2-3 times performance improvement through hidden matrix transposition and asynchronous data transfer. Combined with the network latency analysis in [Figure 8: see original paper], this demonstrates that despite the pressure on the on-chip network, the data management mechanism remains crucial for improving program performance on many-core processors. With 64 active cores, Godson-T Pro achieves 14.8 Gflops for 4K FFT, 30.8 Gflops for 16K FFT, and 43.9 Gflops for 64K FFT. Larger FFT sizes increase parallel granularity and reduce the proportion of synchronization overhead, thereby improving computational efficiency.

[FIGURE:10(a)] shows the scalability of the optimized one-dimensional FFT on Godson-T Pro, with the vertical axis representing self-speedup normalized to single-core performance on Godson-T Pro. [FIGURE:10(b)] shows the scalability of the traditional one-dimensional FFT on Godson-T Base, normalized to single-core performance on Godson-T Base. From 4K to 64K FFT, as data size increases and synchronization overhead decreases, both schemes show improved speedup. For 64K FFT with 64 active cores, speedup exceeds 50x. Thus, the FFT algorithm demonstrates good scalability with Godson-T Pro' s data management mechanism.

Finally, compares the performance of 64K single-precision one-dimensional FFT on Godson-T Pro with IBM' s Cell processor and IBM' s Cyclops-64 many-core processor. The results show that Godson-T Pro' s one-dimensional FFT performance surpasses IBM' s Cell processor, ranking first at 43.9 Gflops, while its peak efficiency (actual performance/peak performance) ranks second at 22.9%, just below Cyclops-64' s 25.8% efficiency.

Performance comparison of 64K single-precision one-dimensional FFT on Cell, Cyclops-64, and Godson-T Pro

Processor	Peak Speed (Gflops)	Actual Performance (Gflops)	Efficiency
Cell [**?**]	204.8	41.9	20.4%
Cyclops-64 [**?**]	80	20.7	25.8%
Godson-T Pro	192	43.9	22.9%

6 Conclusion

Through the above analysis, we conclude that the on-chip data management mechanism based on the Godson-T Pro many-core processor is crucial for improving computational performance and efficiency. The hybrid on-chip storage structure combining caches and scratchpad memory ensures good programmability while enabling convenient data storage management for programmers and improving on-chip storage resource utilization. With support from the coprocessor DTA, not only can data block pattern transformations such as matrix transposition be implemented, but also data transfer management through asynchronous data transfer, overlapping computation with memory access to tolerate on-chip network latency and improve effective computational performance on many-core processors.

In future work, we will introduce flow control mechanisms into the Godson-T Pro architecture to mitigate the pressure on the on-chip network caused by asynchronous data transfer, reduce on-chip network latency, and further improve system computational performance and efficiency.

References

[1] Dally, W.J., Interconnect-limited VLSI architecture, in Interconnect Technology. 1999

[2] Kim, C., D. Burger, and S.W. Keckler, An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches, in Proceedings of the 10th international conference on Architectural support for programming languages and operating systems. 2002, ACM: San Jose, California

[3] Dally, W.J. and B. Towles, Route packets, not wires: on-chip inteconnection networks, in Proceedings of the 38th conference on Design automation. 2001, ACM: Las Vegas, Nevada, United States

[4] Admed Hemani, A.J., Shashi Kumar, Adam Postula, Network on chip: An architecture for billion transistor era, in NorChip. 2000

[5] Chishti, Z., M.D. Powell, and T.N. Vijaykumar, Optimizing Replication, Communication, and Capacity Allocation in CMPs. SIGARCH Comput. Archit. News, 2005. 33(2): p. 357-368

[6] Beckmann, B.M. and D.A. Wood, Managing Wire Delay in Large Chip-Multiprocessor Caches, in Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture. 2004, IEEE Computer Society: Portland, Oregon

[7] Cho, S. and L. Jin, Managing Distributed, Shared L2 Caches through OS-Level Page Allocation, in Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. 2006, IEEE Computer Society

[8] Ali, A., L. Johnsson, and J. Subhlok, Scheduling FFT computation on SMP and multicore systems, in Proceedings of the 21st annual international conference on Supercomputing. 2007, ACM: Seattle, Washington

[9] Franchetti, F., et al., FFT program generation for shared memory: SMP and multicore, in Proceedings of the 2006 ACM/IEEE conference on Supercomputing. 2006, ACM: Tampa, Florida

[10] Bailey, D.H., FFTs in external or hierarchical memory. J. Supercomput., 1990. 4(1): p. 23-35

[11] Kahle, J.A., et al., Introduction to the cell multiprocessor. IBM J. Res. Dev., 2005. 49(4/5): p. 589-604

[12] Williams, S., et al., Scientific computing Kernels on the cell processor. Int. J. Parallel Program., 2007. 35(3): p. 263-298

[13] Cuvillo, J.d., et al., Toward a Software Infrastructure for the Cyclops-64 Cellular Architecture, in Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment. 2006, IEEE Computer Society

[14] Chen, L., et al., Optimizing the Fast Fourier Transform on a Multi-core Architecture, in Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. 2007. p. 1-8

[15] Ainsworth, T.W. and T.M. Pinkston, On Characterizing Performance of the Cell Broadband Engine Element Interconnect Bus, in Proceedings of the First International Symposium on Networks-on-Chip. 2007, IEEE Computer Society

[16] Iftode, L., J.P. Singh, and K. Li, Scope consistency: a bridge between release consistency and entry consistency, in Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures. 1996, ACM: Padua, Italy

[17] Thottethodi, M., A.R. Lebeck, and S.S. Mukherjee, Self-Tuned Congestion Control for Multiprocessor Networks, in Proceedings of the 7th International Symposium on High-Performance Computer Architecture. 2001, IEEE Computer Society

Author Biographies:

Yongbin Zhou: Ph.D. candidate, Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences

Junchao Zhang: Ph.D., Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences

Shuai Zhang: M.S. candidate, Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences

Hao Zhang: Ph.D., Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences

Note: Figure translations are in progress. See original paper for figures.