Hindawi
Scientific Programming
Volume 2017, Article ID 1496104, 9 pages
https://doi.org/10.1155/2017/1496104

Research Article

Scalable Parallel Distributed Coprocessor System for Graph Searching Problems with Massive Data

Wanrong Huang, Xiaodong Yi, Yichun Sun, Yingwen Liu, Shuai Ye, and Hengzhu Liu

School of Computer, National University of Defense Technology, Deya Road No. 109, Kaifu District, Changsha, Hunan 410073, China

Correspondence should be addressed to Wanrong Huang; huangwr1990@163.com

Received 2 May 2017; Accepted 20 November 2017; Published 19 December 2017

Academic Editor: José María Álvarez-Rodríguez

The Internet applications, such as network searching, electronic commerce, and modern medical applications, produce and process massive data. Considerable data parallelism exists in computation processes of data-intensive applications. A traversal algorithm, breadth-first search (BFS), is fundamental in many graph processing applications and metrics when a graph grows in scale. A variety of scientific programming methods have been proposed for accelerating and parallelizing BFS because of the poor temporal and spatial locality caused by inherent irregular memory access patterns. However, new parallel hardware could provide better improvement for scientific methods. To address small-world graph problems, we propose a scalable and novel field-programmable gate array-based heterogeneous multicore system for scientific programming. The core is multithread for streaming processing. And the communication network InfiniBand is adopted for scalability. We design a binary search algorithm to address mapping to unify all processor addresses. Within the limits permitted by the Graph500 test bench after 1D parallel hybrid BFS algorithm testing, our 8-core and 8-thread-per-core system achieved superior performance and efficiency compared with the prior work under the same degree of parallelism. Our system is efficient not as a special acceleration unit but as a processor platform that deals with graph searching applications.

1. Introduction

Information technology, the Internet, and intelligent technology have ushered in the era of big data. Data-intensive applications, as a typical representative of big data applications represented by graph searching, have been receiving increased attention [1]. Many real-world applications could be abstracted as a large graph of millions of vertices, but this procedure is a considerable challenge for processing. These applications represent the connections, relations, and interaction among entities, such as social networks [2], biological interactions [3], and ground transportation [1]. Poor data-driven computation, unstructured organization, irregular memory access, and low computations-to-memory ratio are the prime reasons for parallel large-graph processing inefficiency [4]. To traverse larger graphs caused by data-intensive applications, a variety of scientific programming methods has been proposed [5, 6]. Tithi et al. [5] optimized the programme and used dynamic load balancing with Intel click++ language. Chen et al. [6] proposed a new parallel model called Codelet model. They all do a good job in speeding up access to memory. However, new parallel computing machines could provide a better platform for software methods. Heterogeneous processing, with reconfigurable logic and field-programmable gate array (FPGAs) as an energy efficient computing systems [7], performs competitively with the multicore CPUs and GPGPUs [4, 8]. The performance of breadth-first search (BFS) on large graphs is bound by the access to high-latency external memory. Thus, we designed considerable parallelism and relatively low clock frequencies to achieve high performance and customized memory architecture to deal with irregular memory access patterns.

The bottleneck of processing graph search is memory. Communication is a primary time overhead in the expansion of processors. In this study, we propose a scalable and novel FPGA-based heterogeneous multicore system for big data applications. The core is multithread for streaming processing, and the communication network is InfiniBand (IB) for scalability. The address mapping is a binary search algorithm mapping, and three levels of hierarchy of memory exist.

The remainder of this paper is organized as follows: Section 2 introduces the details of our 1D decomposition hybrid BFS algorithm. Section 3 shows the details of the proposed parallel system architecture. Section 4 describes the implementation of binary search address mapping. Section 5 provides details of the single processor architecture. Section 6 exhibits the three-level memory hierarchy. Section 7 analyzes the experiment results.

2. Related Works

While parallel computers with millions of cores are already in production, the trend is geared toward higher core densities with deeper memory hierarchies [10] even though other node resources (e.g., memory capacity per core) are not scaling proportionally [11]. Anghel et al. (2014) [12] analyzed node-to-node communications and showed that the application runtime is communication bound and that the communication makeup is as much as 80% of the execution time of each BFS iteration [13].

The Graph500 benchmark is the representative of the graph-based analytic class of applications and is designed to assess the performance of supercomputing systems by solving the BFS graph traversal problem [14, 15].

Fast graph traversal has been approached from a range of architecture methods. Fast graph traversal has been approached from a range of architecture methods. In general-purpose CPU and multicore/supercomputing approaches [16, 17], Agarwal et al. performed locality optimizations on a quad-socket system to reduce memory traffic [18]. A considerable amount of research on parallel BFS implementations on GPUs focuses on level-synchronous or fixed-point methods [19, 20]. The reconfigurable hardware approach in solving graph traversal problems on clusters of FPGAs is limited by graph size and synthesis times [4, 8]. Betkaoui et al. (2012) [4] and Attia et al. (2014) [8] explored highly parallelized processing elements (PEs) and decoupled computation memory. Umuroglu et al. (2015) [11] demonstrated the density, rather than the sparsity, of the treatment of the BFS frontier vector in yielding simpler memory access patterns for BFS, trading redundant computation for DRAM bandwidth utilization, and exploring faster graphs.

3. 1D BFS Algorithm for Testing

In 2013, Beamer et al. [21] proposed a bottom-up algorithm on BFS, which dramatically reduces the number of edges examined, and presented the combination of a conventional top-down algorithm and a novel bottom-up algorithm. The combined algorithm provides a breakthrough for level-synchronized parallel BFS in parallel computation, and this novel bottom-up algorithm is applied in 2D sparse matrix partitioning-based solutions. Today, the 2D bottom-up BFS implementation is a general application in Blue Gene architecture.

Experiments show that with a large number of processors relative to the 1D decomposition, the 2D decomposition can effectively reduce the total communication between processors. In the 2D decomposition, the BFS algorithm has better performance. By contrast, with a small number of processors, the BFS algorithm is suitable in the 1D decomposition. Moreover, our system has eight processors in parallel with unified fine-grained address mapping. Algorithm 1 uses 1D decomposition-optimized BFS algorithm which is proposed by Yasui et al. [22]. In Algorithm 1, V is the set of vertex in graph, while E is the set of edges; that is, $E(u, v) = 1$ means u and v are connected. The parent $[k]$ gives the parent of vertex k in the BFS tree whose source vertex is s; when k is unreachable from s, parent $[k] = -1$. $v \in V$; if v is in the frontier queue, then next $[v] = 1$. The same meaning is given to next (v) (next frontier for each BFS iteration) and visit (v) (when visit $(v) = 1$, v has been visited). Array next, visit, and frontier are stored as bitmap. A new vertex appears in the search; then end $= 0$. When no new vertex appears in the current iteration of BFS, the iteration will end.

4. Massive Parallel Coprocessor System Architecture

Our massive parallel coprocessor system architecture is organized by a single master processing node and large numbers of coprocessing nodes for special computation tasks. The master processing node is an embedded system with ARM processor as its core. The communication architecture of our system is the IB communication network.

The coprocessor is a development board with FPGA (Virtex-7), which is a reconfigurable processor for solving graph problems. Two blocks of DDR3 memory are integrated on each board, and data are transferred by a memory controller (MC). We modified the MC's IP core so that two blocks of DDR3 memory could be accessed in parallel. Any processing node would assign the tasks and transfer data to all coprocessors through the I/O interface and target channel adapter (TCA), which is the communication interface we implement based on the IB protocol. Communications data from the TCA are sent to the IB switch interface through a transmitter (TX) by the IB protocol, and the communication data from IB switch interface are received by TCA through the receiver (RX). The max theoretical line rate is 13.1 Gb/s, and the actual line rate is 10 Gb/s. We have four lines; thus, the communication bandwidth is 40 Gb/s. When the system is initialized, the master node distributes data to the DDR3 memory of each coprocessor via PCI-E bus. After the system has started, each processing node communicates through the IB communication network whose interface is TCA. The program in the master node sends its instructions or data after address mapping (i.e., AM in Figure 1) and each coprocessor communicates after the address mapping. Address mapping is implemented by the FPGA, and the scheme is a functional hardware unit for each node. The architecture is described in Figure 1.

The core is a streaming processor that uses a multithreading vector. Cross-multithreading is a fine-grained multithreading in which threads are executed alternately. Our massive parallel coprocessor system is a scalable system and a platform for parallel processing of big data applications.

```
     Input: V[1 ··· n], E[1 ··· n][1 ··· n], s (source vertex)
     Output: parent[1 ··· n].
(1)  for ∀v ∈ V do
(2)      visit[v] = 0
(3)      parent[v] = −1
(4)      frontier[v] = 0
(5)      next[v] = 0
(6)  frontier[s] = 1
(7)  level = 0
(8)  end = 0
(9)  while fend ≠ 1 do
(10)   end = 1
(11)   if (level < α) or (level > β) then
(12)     for ∀v ∈ V do
(13)       if frontier[v] = 1 then
(14)         for ∀u ∈ E(v) do
(15)           if visit[u] = 0 then
(16)             visit[u] = 1
(17)             next[u] = 1
(18)             parent[u] = v
(19)             end = 0
(20)   else
(21)     for ∀v ∈ V do
(22)       if visit[v = 0] then
(23)         for ∀u ∈ E(v) do
(24)           if frontier[u] = 1 then
(25)             visit[v] = 1
(26)             next[v] = 1
(27)             parent[v] = u
(28)             end = 0
(29)             BREAK
(30)   BARRIER
(31)   for ∀v ∈ V do
(32)     frontier[v] = next[v]
(33)     next[v] = 0
(34)   level = level + 1
(35)
```

Algorithm 1: Parallel 1-D BFS algorithm.

5. Binary Search Address Mapping Unit

The architecture of our address mapping is shown in Figure 2. In our scheme, the memory of DDR3 is divided into two areas: the local data blocks and the global translation blocks. The local data blocks store the data that the program needs from I/O requests and TCA. The global translation blocks hold the mapping of all data. Furthermore, the global translation blocks in each node are the same, and they are managed in a fine-grained page.

The basic idea of binary search is as follows: In ascending order of the tables, we take intermediate records as objects of comparison. If the given item is equal to the intermediate records, then the search is done. However, if the given item is smaller than the intermediate records, then we have a binary search in the first half of the ascending table; otherwise, we have a binary search in the bottom half of the ascending table.

The implementation of the binary search address mapping is a pipeline in which the virtual address is the input, and the output data are the physical address. We divided the registers in the pipeline storage unit into three groups. The first group contains the OMR, DVR, and MTR register, which stores the status of data in RAM. The RAM stores the address mapping of the visited arrays (visited array in the BFS algorithm). The input of the virtual address is the frontier arrays (frontier array in the BFS algorithm). This situation means that we could not find the corresponding mapping in RAM and we would obtain one of the address mappings of the frontier array from the DDR3 memory to be stored in RAMs. This situation is object missing (array missing) where no object is missing in the beginning of a binary search and after the first stack, but the data are missing. The range of virtual address in the array would be entered in the range register (RR). When the input address is not in the range of each RR, a warning that an object is missing is triggered. Then, we would update the data in RR; the order of updating uses the least recently used mechanism. The pipeline then stalls, and we acquire one dataset in the frontier array to continue the

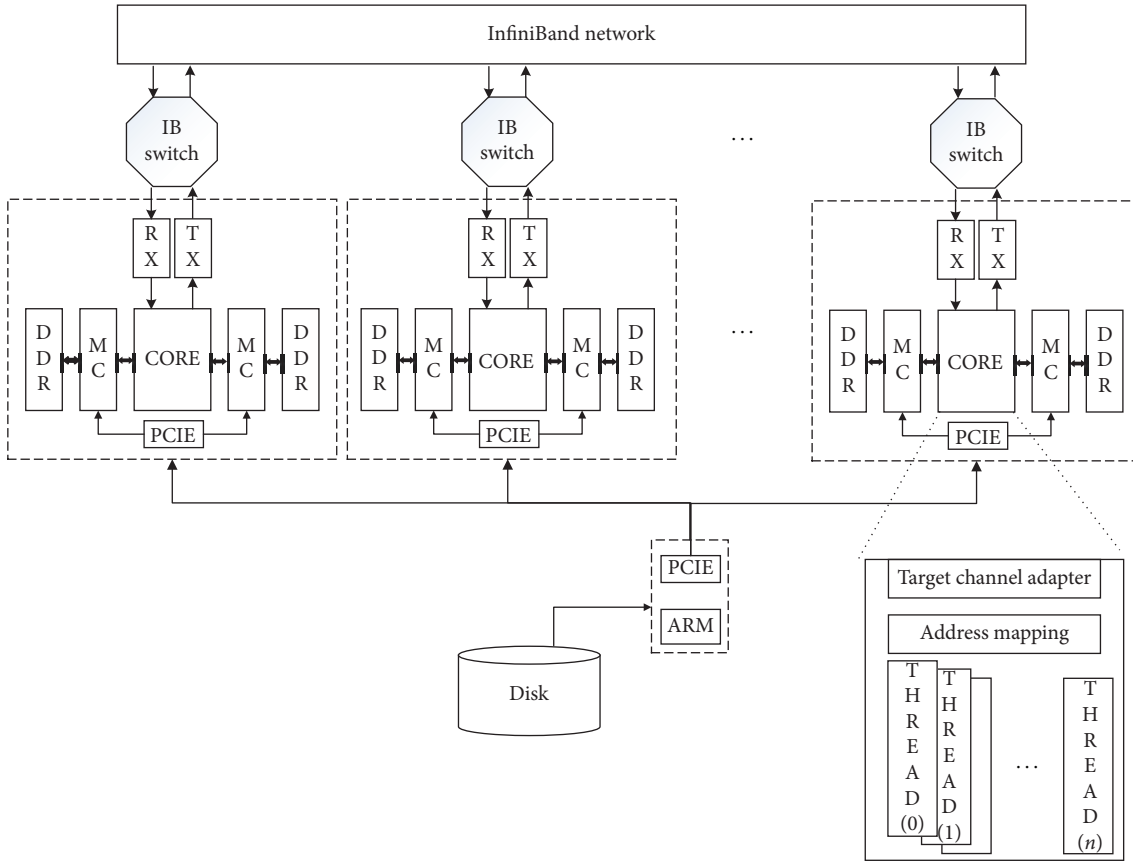

Figure 1: The overview of the system.

search. One data item of the frontier array exists in the current stack of RAMs. Thus, when searching the next stack, the next data in RAMs are invalid. We would obtain the address mapping data of the Frontier array from the DDR3 memory to be stored in this RAMs, and the situation is that the data are missing. The ith OMR is the object missing indicator of the ith stack in pipeline. Furthermore, the MTR is the data missing indicator. The 1 bit in DVR indicates the 64 bits of data in the RAMs that are currently stacked. When an object is missing, the DVR is 0. After updating in RAMs, the corresponding bit would be set to 1.

The second group contains the VAR and DVR, which are temporary storage that passes the data to the next stack when the pipeline is running. The third contains the LBR, RBR, and CAR. In the binary search algorithm, in each searching step, the smaller value is in the LBR, and the larger value is in the RBR. The output of the RAMs provides the intermediate element, and CAR records the comparable results in all steps. For instance, in a kth stack, when the virtual address is in the range of LBR and output of RAMs, the kth bit in CAR is 1 and the value of $(k-1)$th to 1 bit is the same as the value in CAR in the $(k-1)$th stack. The number of stacks is equal to the number of processors.

The LRU unit is implemented by a series of shift registers (the number of bits width is the number of shift register minus 1). When RR is updated, we set 1 to the corresponding shift register, and each register shifts left. The register, which is 0, is the least recently used. The LRU unit provides the number, and the corresponding RR is waiting to be updated in the next object missing.

Unlike time complexity $O(n)$ in direct search, the time complexity of binary search is $O(\log n)$. When the value of n increases, the advantages of the method are obvious.

6. Architecture of Streaming Processor

Our streaming processor design is on the basement of the multithreading vector. Cross-multithreading is a fine-grained multithreading, in which threads are executed alternately. This design requires the switching of threads in each clock whether the thread is stalling or not. This mechanism ensures that the pipeline constantly runs. When a stall exists in a thread, the latter would initially have a request. Every time the thread turns to use the pipeline, it would wait until the stall is handled. The execution of the thread using a cycle mode and its architecture is shown in Figure 3.

The architecture of the microprocessor without interlocked piped stages (MIPS) was simplified to efficiently process graph searching problems. The pipeline was divided into Thread Select (TS), Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), and Write Back (WB) sections.

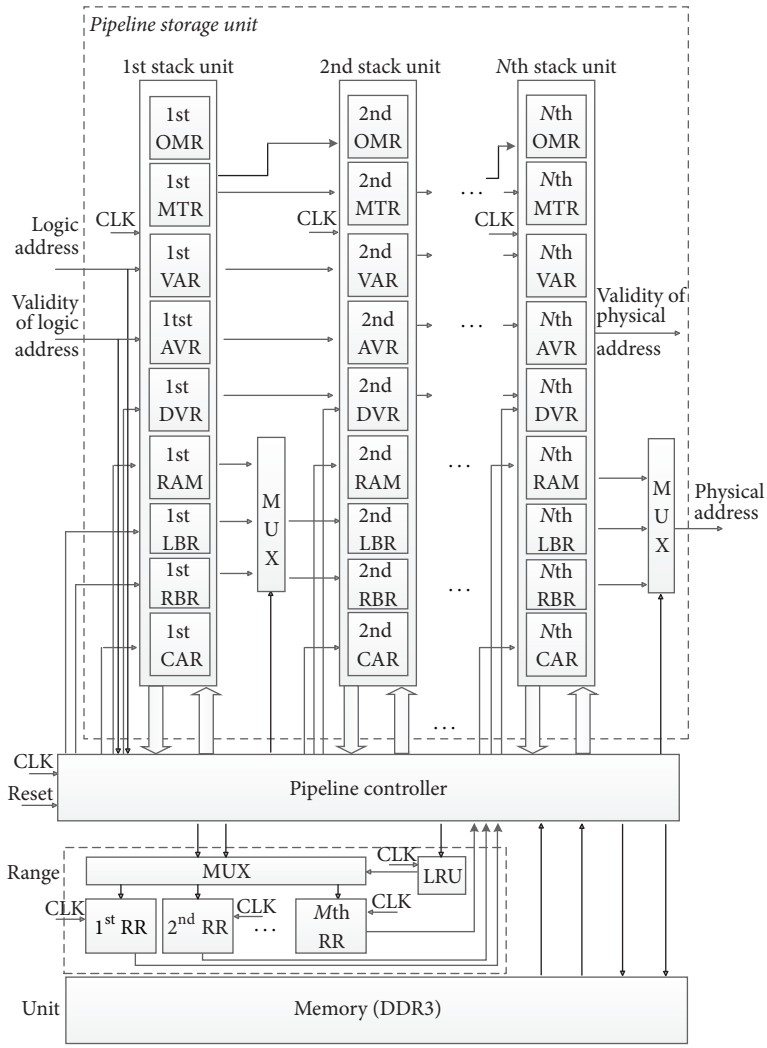

Figure 2: Implementation of binary search address mapping.

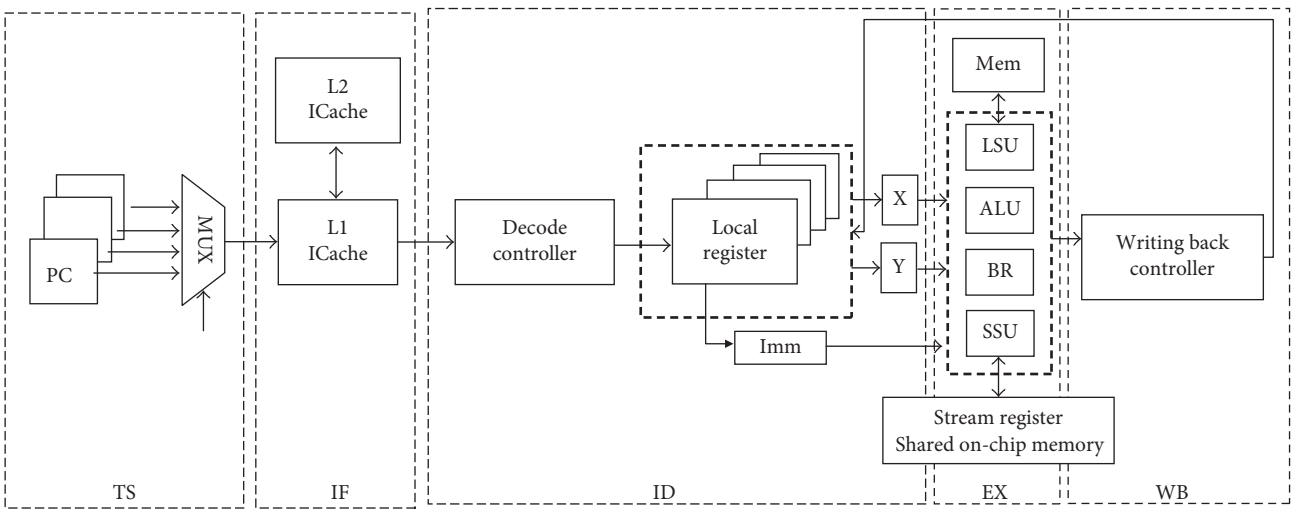

Figure 3: The architecture of streaming processor.

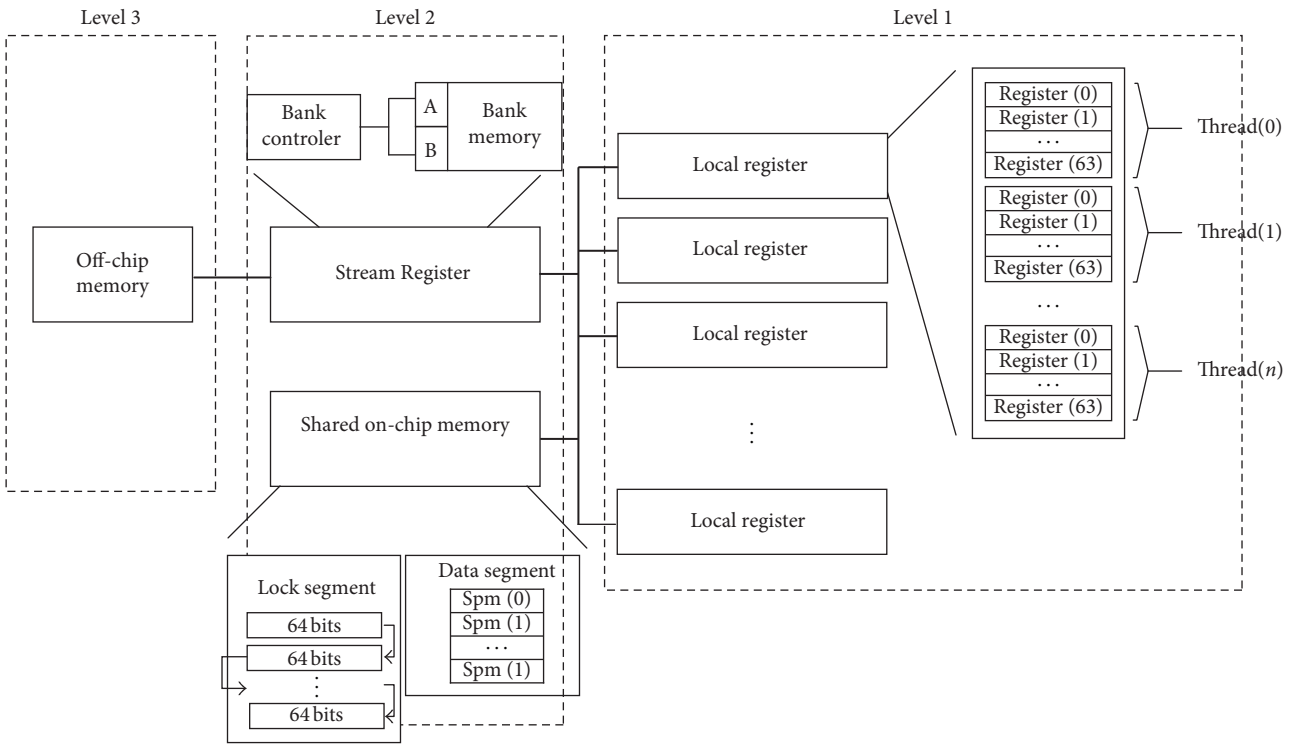

Figure 4: Three-level memory hierarchy.

In the Thread Select section, a thread would be selected and the value of the corresponding program counter (PC) register is obtained as the address in the IF section. Each thread has its corresponding PC register and the value in the register is updated according to the Next Program (NPC, in EX section). The update must be done before another thread is selected. Eight threads are implemented and used according to the instruction register in the cycle because of the limitation of the FPGAs resources.

In the Instruction Fetch section, the pipeline obtained the correct instruction according to the corresponding PC value. Two instruction caches exist: L1 and L2 caches. The L1 cache is private for a process, and the L2 cache is shared given more than one process. The L1 cache is 0.5 kB, and it adopts fully associative mapping programs. The L2 cache is 4 kB and adopts direct mapping programs. An instruction register is designed for each thread to store the last instruction. When the thread is blocked, it could take the last instruction from the instruction register.

In the Instruction Decode section, the decode controller matches the instruction and reads the correct data in the data register.

Four types of operation exist in the Execute section: the operation of arithmetic logic, load and store, access in stream register and shared on-chip memory, and branches jump instruction. In the Write Back section, the result in the Execute section is written in the local register. The result could be from LSU (load and store unit), ALU, and SSU (stream register and shared on-chip memory unit). The address is in the instruction.

Load and store unit (LSU) is designed for operation access, which includes vector or scalar access. The load operation is from the memory to the stream register. The store operation is from the stream register to the memory. Shared memory on-chip unit (SSU) is to perform operation accessed stream register or shared on-chip memory. The structures of LSU and SSU are similar.

7. Three-Level Memory Hierarchy

The stored data in the graph search problems has two characteristics: sparsity of the storage and lack of locality in memory accessing. The optimization in cache cannot effectively use the locality of the memory access. By drawing on the experience of the design of stream architecture, a new three-level memory hierarchy is proposed.

Figure 4 shows that the first-level memory is a local register. Our architecture is a multicore and multithread structure. The local register distributes in the inner part of processor. When multithreads exist in the execution in each process, the processor needs to protect the state. A distributed thread registers to create each thread with its own private registers to store their own states. As they have a local register, no access conflicts occurred between the threads. Implement register mapping is unnecessary. Thus, a small access in the address space and low access delay and power exist.

The local register is implemented by a block RAM resource in FPGAs. In the multithread execution, the local register would simultaneously process two requests of reading from decoding and one request of writing from writing back.

Considering that a single block RAM has read-and-write ports, we copy a local register for each thread. When two requests for reading exist, they could be read separately; however, if one request for writing exists, they would be written simultaneously.

The second level is the stream register and the shared memory on the chip. The stream register is a data buffer on the chip that is implemented by block RAM and combined with a bank of stream register to support multirequest from the local register. A bank of stream register contains a bank memory and a controller (as shown in Figure 4). The bank memory provides two read-and-write ports (ports A and B), and they are the interfaces of the local register and the off-chip memory.

The controller of the stream register has three functions. One is handling the data and the request from the local register. Another is handling the data and request from off-chip memory. The last is coordinating the read-and-write ports that consume the data writing or reading in the stream register with a correct sequence.

The shared on-chip memory is a special RAM on a chip. Furthermore, it is controlled through software and is programmable, which is different from cache. The data is accessed by addresses, and it is shared by processors (if we have) and threads. This study's design of a shared on-chip memory is a sharing and communicating interface for the processors and threads. The function of the design is data transaction and synchronization. The shared on-chip memory is divided with the lock and data segments. When more than one processor or thread exists to read and write in the same address, the atomic lock is supported by the lock segment to ensure correct program sequence. A base address exists in the lock segment. The bits begin with the base address, and one bit determines an atomic lock. Each process or thread could read only one bit in a lock segment at a time. The data segment is special shared data in a multithread or multiprocess program execution. Barrier synchronous counter is one of the special shared data items. The data segment is divided into several 64 bits block, and one local register is composed of 64 bits.

Owing to the width in a transaction burst at 512 bits, the width between the interface of the off-chip memory and the stream register is also 512 bits. The data width between the local register and stream register is 64 bits, and it is the same as the data in the local register. The valid data transaction between the stream and local registers is identified by a mask. The mask is given by a program instruction and its range is from 0 to 7.

The third-level memory hierarchy is the off-chip memory. Our off-chip memory is a dual-data rate (DDR3) and is provided by FPGAs.

8. Results and Comparison

The proposed scalable parallel distributed coprocessor system has 8 cores, and each core has 8 threads. We used the 8 Xilinx Virtex-7 FPGA VC709 evaluation board (xc7vx485t-2ffg1761) and a commercial switchboard called Mellanox IS5030, which is based on the IB protocol, to implement the system. The Xilinx Virtex-7 FPGA VC709 evaluation board has 2 SODIMM DDR3 memory with a storage capacity of 4 GB. Eight channel PCIE interfaces and four line GTH transceivers are included. The communication bandwidth is 40 GB/s, and the memory bandwidth in each core is 10 GB/s in theory. Moreover, two computers that run Linux are used. One of them is responsible for the initialization of the switchboard, and the other is responsible for generating the BFS algorithm, loading data, and receiving returned results. The number of nodes in the system can be expanded as needed. We use the Verilog HDL to achieve a parallel architecture system in Xilinx Vivado 2013.4, which is written to the FPGA chip through the JTAG interface.

In accordance with the Graph500 benchmark, we generated a series of information through a Kronecker graph generator. Then, the information is converted to any type of data structure, which is the input of the BFS algorithm. We verified the results after execution. In the above steps, the creation of the data structure and the design of the BFS algorithm can be customized by the user.

In the Graph500, a fair comparison of the processor with different test bench is obtained using TEPS. According to the performance, which is calculated in Graph500, we proposed a formula to calculate performance P. The details are shown in (1). When the dataset and the root node are determined, E is a constant.

E is the number of edges in the connected region of the root node in the graph. f is the working frequency, which is 200 MHz in our implementation. T_{clk} indicates the number of clock cycles between the beginning and the end of the program. We obtain the T_{clk} through the chip scope. The test dataset in Graph500 is used. Table 1 presents our performance and comparison.

$$P = \frac{E \times f}{T_{\text{clk}}}. \tag{1}$$

We run the parallel BFS algorithm described as Algorithm 1 on our prototype system for testing. The scale of graph searching using the BFS is 19 to 23, which means that the scale of graph data is 2^{19} to 2^{23}, and the edge factor of the graph is 16.

In the first experiment step, we use Vivado 2014.1 to load the test data to FPGAs with a computer. The test data is from Graph500. Then we run the BFS programme in Linux which is running in ARM cortex. The ARM cortex is provided by the evaluation board. The ARM cortex initializes the searching and gets the results from FPGAs through PCIE bus. Finally we use ChipScope which is in Vivado 2014.1 to analyze the performance.

As most works targeting high-performance BFS use MTEPS as a metric, comparing raw traversal performance is possible but the available memory bandwidth in the hardware platform sets a hard limit on achievable BFS performance. Our experimental results are from a Virtex-7 platform with much less (utilization of bandwidth is 64%, the theoretical bandwidth is 10 GB/s, the actual bandwidth is 6.4 GB/s) memory bandwidth and work frequency (200 MHz) than platforms in prior work; thus, it is comparatively slow. Our

Table 1: Comparison to prior work.

Work	Platform	No. of parallel units	Avg. MTEPS	BW (GB/s)	MTEPS/BW
[4]	Convey HC-2	512	1600	80	20
[4]	Convey HC-2	256	980	80	12.5
[4]	Convey HC-2	128	510	80	6.375
[4]	Convey HC-2	64	350	80	4.375
[4]	Convey HC-2	32	210	80	2.625
[8]	Convey HC-2	64	1900	80	23.75
[9]	Nehalem + Fermi	32	800	128	6.25
This work	Virtex-7 & InfiniBand	8	169	10	16.9
This work	Virtex-7 & InfiniBand	64	763	80	9.54

system has 8 cores and supports 8 threads per core, which is equivalent to 64 threads in parallel. Considering the memory bandwidth, the traversals per unit bandwidth is used as a metric to enable fair comparison with prior work. Table 1 allows the comparison with several related works on the average performance, available memory bandwidth, and traversals per bandwidth over RMAT graphs.

9. Conclusions

We can draw the following conclusions:

(1) Compared with the approach of Betkaoui et al. [4], our system is more efficient. The performance of 64 parallel units is similar to that of approximately 256 parallel units in [4]. Our data of traversals per unit bandwidth in 64 parallel units are between 256 units and 128 parallel units in Betkaoui et al. [4].

(2) Attia et al. [8] is on BFS algorithm optimization, and our system is a scalable general processor platform that performs instruction set decoding and the address mapping. Attia et al. [8] is limited to scale, and it is a special acceleration unit.

(3) Our data of traversals per unit are twice that of Hong et al. [9], and the performance is approximately equal. Moreover, our proposed system has the advantages of power and scalability.

(4) The proposed system can be used as a scalable general processing system for graph application with big data.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Nature Science Foundation of China (no. 61602496).

References

[1] C. Demetrescui, A. Goldberg, and D. Johnson, "The Shortest Path Problem: Ninth DIMACS Implementation Challenge, Proceedings.dimacs Book.ams," in *Proceedings of the The Shortest Path Problem: Ninth DIMACS Implementation Challenge, Proceedings.dimacs Book.ams*, p. 4, 2006.

[2] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E: Statistical, Nonlinear, and Soft Matter Physics*, vol. 69, no. 2, Article ID 026113, pp. 1–26113, 2004.

[3] D. Bu, Y. Zhao, L. Cai et al., "Topological structure analysis of the protein-protein interaction network in budding yeast," *Nucleic Acids Research*, vol. 31, no. 9, pp. 2443–2450, 2003.

[4] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A reconfigurable computing approach for efficient and scalable parallel graph exploration," in *Proceedings of the 2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2012*, pp. 8–15, Netherlands, July 2012.

[5] J. J. Tithi, D. Matani, G. Menghani, and R. A. Chowdhury, "Avoiding locks and atomic instructions in shared-memory parallel BFS using optimistic parallelization," in *Proceedings of the 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013*, pp. 1628–1637, USA, May 2013.

[6] C. Chen, S. Koliai, and G. Gao, "Exploitation of locality for energy efficiency for breadth first search in fine-grain execution models," *Tsinghua Science and Technology*, vol. 18, no. 6, Article ID 6678909, pp. 636–646, 2013.

[7] A. Putnam, A. M. Caulfield, E. S. Chung et al., "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA '14)*, pp. 13–24, IEEE, Minneapolis, Minn, USA, June 2014.

[8] O. G. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno, "CyGraph: A reconfigurable architecture for parallel breadth-first search," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2014*, pp. 228–235, usa, May 2014.

[9] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques, PACT 2011*, pp. 78–88, USA, October 2011.

[10] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling techniques for massive scale-free graphs in distributed (external) memory," in *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2013*, pp. 825–836, USA, May 2013.

[11] Y. Umuroglu, D. Morrison, and M. Jahre, "Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform,"

in *Proceedings of the 25th International Conference on Field Programmable Logic and Applications, FPL 2015*, UK, September 2015.

[12] A. Anghel, G. Rodriguez, and B. Prisacari, "The importance and characteristics of communication in high performance data analytics," in *Proceedings of the 2014 IEEE International Symposium on Workload Characterization, IISWC 2014*, pp. 80-81, USA, October 2014.

[13] A. Amer, H. Lu, P. Balaji, and S. Matsuoka, "Characterizing MPI and hybrid MPI+threads applications at scale: Case study with BFS," in *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2015*, pp. 1075–1083, China, May 2015.

[14] M. Anderson, "Better benchmarking for supercomputers: The usual yardstick is not a good metric," *IEEE Spectrum*, vol. 48, no. 1, pp. 12–14, 2011.

[15] A. D. Bader, J. Berry, S. Kahan, and R. Murphy, "The graph 500 list," 2010, http://graph500.org.list.

[16] R. Berrendorf and M. Makulla, "Level-synchronous parallel breadthfirst search algorithms for multicore and multiprocessor systems," in *Proceedings of the the Sixth Intl. Conf. on Future Computational Technologies and Applications*, vol. 26, 2014.

[17] M. Ceriani, G. Palermo, S. Secchi, A. Tumeo, and O. Villa, "Exploring manycore multinode systems for irregular applications with FPGA prototyping," in *Proceedings of the 21st Annual International IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2013*, p. 238, USA, April 2013.

[18] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010*, USA, November 2010.

[19] M. Bisson, M. Bernaschi, and E. Mastrostefano, "Parallel distributed breadth first search on the Kepler architecture," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 2091–2102, 2015.

[20] U. A. Acar, A. Charguéraud, and M. Rainey, *Fast parallel graph-search with splittable and catenable frontiers, Inria*, 2015.

[21] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[22] Y. Yasui, K. Fujisawa, and K. Goto, "NUMA-optimized parallel breadth-first search on multicore single-node system," in *Proceedings of the 2013 IEEE International Conference on Big Data, Big Data 2013*, pp. 394–402, USA, October 2013.