Cascaded DMA Controller for Speedup of Indirect Memory Access in Irregular Applications

Tomoya Kashimata
Waseda University
Tokyo, Japan
kashi@kasahara.cs.waseda.ac.jp

Toshiaki Kitamura
Waseda University
Tokyo, Japan
toshi.kitamura@aoni.waseda.jp

Keiji Kimura
Waseda University
Tokyo, Japan
keiji@waseda.jp

Hironori Kasahara
Waseda University
Tokyo, Japan
kasahara@waseda.jp

***Abstract*—Indirect memory accesses caused by sparse linear algebra calculations are widely used in important real applications. However, they also cause serious inefficient memory accesses and pipeline stalls resulting in low execution efficiency even with high memory bandwidth and much computational resource. One of the important issues of indirect memory accesses, such as accessing A[B[i]], is it requires two successive memory accesses: the index loads (B[i]) and the following data element accesses (A[B[i]]). To overcome this situation, we propose the Cascaded-DMAC (CDMAC). This CDMAC is intended to be attached in each core of a multicore chip in addition to a CPU core, a vector accelerator, and a local data memory. It performs data transfers between an off-chip main memory and an in-core local data memory, which provides data to the accelerator. The key idea of the CDMAC is cascading two DMACs so that the first one loads indices, then the second one accesses data elements by using these indices. Thus, this organization realizes the autonomous indirect memory accesses by giving an index array and an element array, and obtains the efficient SIMD computations by lining up the sparse data into the local data memory. We implemented a multicore processor having the proposed CDMAC on an FPGA board. The evaluation result of sparse matrix-vector multiplications on the FPGA shows that the CDMAC achieves a maximum speedup of 17× compared with the CPU data transfer.**

***Index Terms*—Cascaded DMA Controller, CDMAC, DMAC, DMA, indirect memory access, sparse matrix vector multiplication, SpMV, SMVM**

I. Introduction

Computational power has been grown every year. In this decade, many part of this growth particularly relies on the improvement of SIMD accelerators like GPUs. They can attain high performance as long as the required data is continuously supplied to their internal functional units. This kind of high-bandwidth data transfer can be realized by regular memory accesses.

On the other hand, many real applications, like structural engineering and graph analysis, employ irregular memory accesses. *Indirect memory access* is a representative one. It consists of two memory accesses for one data access: index loading and data element accessing itself. While successive executions of index loading are regularly performed on a memory, corresponding data elements are randomly accessed. These random memory accesses result in an expensive memory access latency. Furthermore, the long latency causes an inefficient program execution. For instance, when computations and memory accesses are performed in a same hardware module and processed by an instruction execution sequence as in an ordinary CPU, the overlap execution among them cannot be expected even with a rich set of computational resources.

Many researchers have tackled this problem. Many of them have targeted sparse matrix-vector multiplication (SpMV), which is a typical computational kernel employing indirect memory accesses [1]–[4]. Yu, et al. proposed Indirect Memory Prefetcher (IMP), which is a special prefetcher for indirect memory accesses [5]. Tanabe, et al. proposed a special memory module, which can execute gather operations to collect randomly scattered data elements on a memory [6].

In this paper, we propose a novel direct memory access controller (DMAC): Cascaded DMAC (CDMAC). CDMAC cascades two DMACs so that the first one loads indices and the second one can access data elements by using indices loaded by the first one. It is intended to be attached between a SIMD or vector type accelerator having a local on-chip memory and an off-chip memory. Thus, for the case of indirect memory load, it gathers data elements from the off-chip memory and stores them into an array on the local memory continuously. Its cascaded organization can fully exploit the memory bandwidth of a system by issuing memory requests continuously both for indices and data elements as much as possible. In addition, the accelerator and CDMAC can be executed simultaneously because they are decoupled from each other. Therefore, this organization can fully utilize both the computational resources and the memory bandwidth. It is also useful for local memory management and data transfer optimization techniques by a compiler [7]. We also propose a data element cache in CDMAC to exploit existing data locality even in the scattered data elements. The proposed CDMAC is implemented on an FPGA board for the evaluation.

The contributions of this paper can be summarized as follows:

- We propose CDMAC to realize latency tolerant systems for indirect memory accesses.
- We add a data element cache in CDMAC to exploit data locality as much as possible.

Part of this paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO) and JSPS KAKENHI Grant Number JP18K19786.

- We implemented the proposed CDMAC on an FPGA board for the evaluation and obtained 17× speedup by SpMV compared with the CPU data transfer.

The rest of this paper is organized as follows: Section II describes the proposed CDMAC. Section III evaluates CDMAC with SpMV. Section IV investigates the cache parameters for sparse matrices. Section V reviews related researches. Finally, Section VI concludes this work.

II. Proposed Cascaded-DMAC

A. Cascaded DMAC

The proposed Cascaded-DMAC (CDMAC) is consists of two DMACs and the address calculation stage between them.

The first DMAC (DMAC1) takes an address of an index array as an input parameter. It fetches indices in a burst data transfer manner as an ordinary DMAC.

The address calculation stage calculates addresses of the data elements to be gathered from the fetched indices, the base address, and the element size. The base address and the element size are given as input parameters for CDMAC. DMA descriptors, which are parameter sets for DMAC, can be generated when needed.

Finally, the second DMAC (DMAC2) issues memory requests along with the descriptors. Note that since the data elements are intended to be randomly accessed, DMAC2 does not use a burst data transfer mode.

We assumed that both the indices and the data elements are 32bit width in this paper. Under this assumption, the same amount of data for the indies reading and the elements writing are transferred continuously. Therefore, the write port of the DMAC2, which has the responsibility of reading the data, can be connected to the DMAC1 as shown in Fig. 1, so that the write port of the DMAC1 can have the responsibility of writing the data elements to the memory. By using this design, the implementation can be simplified. For instance, the address calculation stage, which generates the descriptor, and the DMAC2 do not have to care about the destination addresses, and the implementation of the DMAC2 becomes easier than ordinary DMACs. In addition, the completion of a data transfer can be checked only by the DMAC1, instead of both DMACs.

The DMAC1 utilized Modular Scatter-Gather DMA [8] (mSGDMA) provided by Intel. The mSGDMA can expose *the source and sink streaming ports*. The fetched data by the memory read port can be sent to the source streaming port, and the given data to the sink streaming port can be sent to the memory write port. As shown in the Fig. 1, the source streaming port of DMAC1 is connected to the address calculation stage, and the sink streaming port is connected to the DMAC2.

Instead of utilizing the mSGDMA, the DMAC2 was implemented from scratch in this paper since the mSGDMA was not designed for random memory accesses that are required to the DMAC2. It receives a source address at the streaming input port and issues a read request to the memory. The memory bus master has to guarantee that it can always receive fetched data when it is requested. To satisfy it, we used FIFO to manage the number of memory requests.

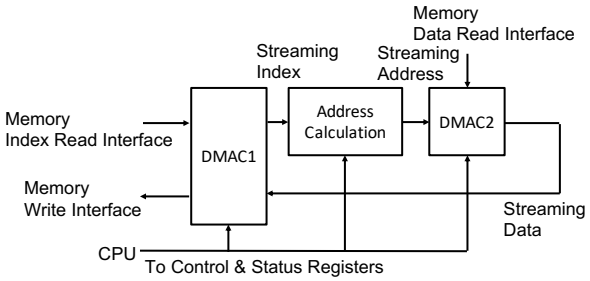


Fig. 1: Block diagram of the proposed CDMAC.

B. Data Element Cache

As described previously, the data elements are randomly accessed by the DMAC2. However, there is an opportunity to exploit data locality even in indirect accesses. For instance, some sparse matrices, such as band matrices, have spatial locality. When CDMAC deals with these matrices, the existing data locality should be utilized. Note that this cache should handle multiple cache misses simultaneously as much as possible to fully exploit the memory bandwidth.

Fig. 2 depicts the architecture of the whole cache. This cache system contains three components: “Cache”, “Memory request controller”, and “Confluence FIFO”.

The “Cache” is different from an ordinary cache in terms of its behavior at a cache miss. If it hits, it sends the address and the data to the Confluence FIFO. If it misses, it sends the address to the Confluence FIFO and the memory request controller. It does not handle the missed request anymore. When the fetched data arrives from the memory request controller, it replaces a cache line according to the FIFO algorithm. We implemented a fully associative cache, which has 32byte line and 512byte in total.

“Memory request controller” prevents duplicated requests to memory and manages memory interface signals.

“Confluence FIFO” is a key feature of this cache system. It receives requested address and hit data from the cache. Also it receives the fetched data and its address from the memory request controller. When it receives the fetched data, it compares the address with the stored address and updates the stored data. We have to prepare a deep FIFO because the number of overlapping memory accesses can be limited by its depth. 256 elements can be stored in our implementation.

We implemented this cache using Chisel [9], which is a high–level HDL using Scala. The cache contents can be invalidated by a control register.

C. CDMAC Data Transfers

1) Scatter Transfers: The scatter data transfer can be realized by setting read/write direction of the second DMAC in CDMAC (DMA2) opposite to that of the gather data transfer.

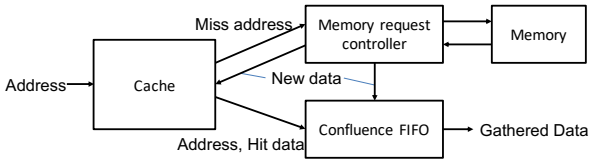


Fig. 2: Block diagram of the implemented cache system

2) Nested Indirect Accesses: CDMAC can also realize nested indirect accesses as shown in Fig. 3 by employing each indirect array access from the innermost index loading to the outermost index loading one by one. At this time, inner index loading requires a temporal array to store the loaded index. This requires some code restructuring. Fig. 4 depicts an example of a restructured code. The first and second loops perform the nested indirect array accesses in the right side of the original code by introducing temporal arrays, `tmparr1` and `tmparr2`. Then, the third loop performs the store operations of indirect array accesses (scatter operation).

```
for(i = 0; i < N; i++)
  out[idxout[i]] = in[idxin1[idxin2[i]]];
```

Fig. 3: Example code of a nested indirect memory accesses

```
for(i = 0; i < N; i++)
  tmparr1[i] = idxin1[idxin2[i]]
for(i = 0; i < N; i++)
  tmparr2[i] = in[tmparr1[i]]
for(i = 0; i < N; i++)
  out[idxout[i]] = tmparr2[i]
```

Fig. 4: Code restructuring example of a nested indirect memory accesses

III. Experimental Evaluation

A. Evaluation environment

We implemented Cascaded DMAC on DE5a-Net (DDR4) having Intel Arria 10 FPGA [10]. Fig. 5 depicts a block diagram of the total multicore system built on the FPGA. The system consists of multiple cores, or processing elements (PEs), each of which has a CPU, a vector accelerator, a DMAC for continuous data transfer, and a CDMAC. The modules in a PE are connected to a local memory, and the PEs are connected to the main memory (DDR4) through a memory bridge (Bridge) and a memory controller (Memory Controller). We used a system-integration tool named Qsys. It generates interconnects automatically. Multiple requests on them can be performed simultaneously as long as there is no contention among them.

The board has two SODIMM DDR4 ports and some outer memories, but this implementation uses only one SODIMM port. Its memory frequency is 1200MHz. However, it is too fast because the PEs and their buses are driven at 50MHz.

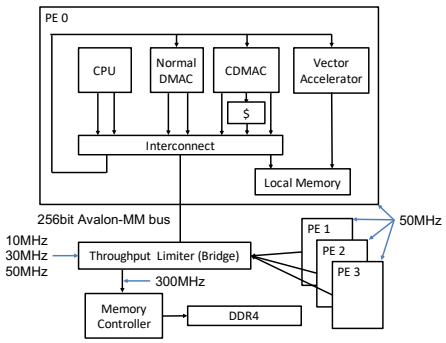


Fig. 5: Block diagram of the system architecture implemented on FPGA. We implemented six types of hardware that changed presence of the cache and the throughput limiter frequency.

Therefore, we limited the memory throughput by two clock crossing bridges. The first bridge connects a bus driven at 50 MHz with another bus driven at either 10MHz, 30MHz, or 50MHz. Its bus width is 256bit. The second bridge connects the low–frequency bus with a 300MHz bus because the clock frequency for the memory controller's user logic is 300MHz. We used 30MHz as the standard bus speed because a commercially available processor and memory have a similar frequency ratio. The 50MHz bus assumes that the memory has a high throughput to show the DMAC potential.

We used NIOS II [11] as the main CPU which has 32KB instruction cache and 32KB data cache. We also used Floating Point Hardware 2 for NIOS II [12].

We used the implemented multicore as a bare-metal system, and there is no address translation. There is also no cache coherence hardware in the system. Instead, the software has a responsibility of cache coherence [13].

Resource usage of the system with 30MHz bus and cache is shown in Table I. The other hardware has similar resource usage. The number of ALMs used in Cache looks too big because our implementation does not use block memory at all and it has many comparators for the confluence FIFO.

TABLE I: Resource usage

	ALMs	M20K	DSPs
Whole system	208,546	952	392
PE0	49,486	218	98
Cascaded DMAC	2,728	17	2
First DMAC	2,273	16	0
Second DMAC	45	1	0
Cache	20,031	0	0
Vector core	11,018	121	89

We used a vector processor as a SIMD accelerator in a PE for this implementation. It is similar to RISC-V vector extension [14]. It has 8 single–precision floating point adders/subtractors and multipliers/dividers each. It can access only to the local memory within the same PE. We suppose that DMACs transfer data between the main memory and the local memory. The implemented accelerator is an in-order processor, but it can overlap scalar operations and vector operations. This

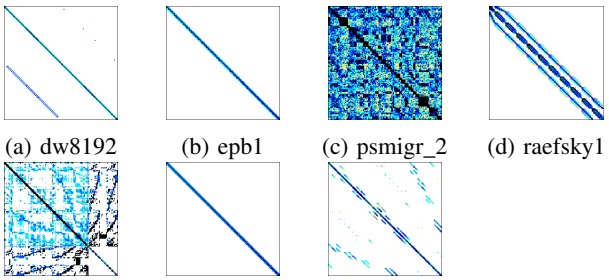
feature enables reducing overhead of scalar calculations such as address calculations and conditional branches.

B. Implementation of software

We used SpMV for the evaluation of CDMAC. We used 7 matrices that are also used in literature [3] except a one that uses complex numbers. They are obtained from University of Florida Sparse Matrix Collection [15]. Table II shows the specification of each matrix. All of them are square matrices. `N` is the length of one side of the matrices. `NNZ` represents the number of non–zero values in them. Fig. 6 shows distributions of non-zero values of each matrix.

TABLE II: Charactoristics of each matrix

matrix	N	NNZ	NNZ/N	NNZ ratio
dw8192	8,192	41,746	5.1	0.06%
epb1	14,734	95,053	6.5	0.04%
psmigr_2	3,140	540,022	172	5.48%
raefsky	3,242	294,276	91	2.80%
scircuit	170,998	958,936	5.6	0.003%
t2d_q9	9,801	87,025	8.9	0.09%
torso2	115,967	1,033,473	8.9	0.008%



(a) dw8192 (b) epb1 (c) psmigr_2 (d) raefsky1

(e) scircuit (f) t2d_q9 (g) torso2

Fig. 6: Distribution of non-zero value of each matrix [15]

dw8192, *epb1*, *raefsky*, *t2d_q9*, and *torso2* are band matrices. Especially, *raefsky* has wide band and high NNZ ratio. On the other hand, *psmigr_2* and *scircuit* are random matrices. *psmigr_2* has many non–zero values while its size is small. *scircuit* is larger and more random than *psmigr_2*. It has similar NNZ/N to the band matrices.

We converted the storage format of these sparse matrices from COO to SELL [16]. The SELL slice parameter is set to 256 because the vector length of the vector processor is 256 at most. All floating values and integers are converted to 32bit floating–point number and 32bit integer type, respectively. We used consecutive numbers starting from 1 as an input vector. They are 32bit floating–point type.

Processing an input vector of SpMV requires gather operation along with column number of the input sparse matrix. We measured the performance on three cases: gathered by CPU, gathered by CDMAC without cache, and gathered by CDMAC with cache.

The input data was initially placed on the off-chip DRAM memory as constant arrays. The input data and the program were compiled separately to prevent constant propagation by the compiler optimization.

C. Result

Fig. 7 shows the evaluation results. As shown in the figure, the performance of CDMAC is better than that of CPU. Especially, when *psmigr_2* is executed with 50MHz-bus and 1PE, the CDMAC without cache attains 15.8× speedup compared to the CPU (0.57 MFLOPS → 8.97 MFLOPS). Furthermore, the CDMAC with cache attains 16.9× speedup (0.57 MFLOPS → 9.62 MFLOPS) under the same conditions.

Higher frequency bus system has shorter memory latency and higher memory throughput in this system. Theoretically, the 10MHz-bus, 30MHz-bus, and 50MHz-bus have 320MB/s, 960MB/s, and 1600MB/s transfer throughput, respectively. Also, our preliminary evaluation reveals that their memory latencies are about 155 cycles, 68 cycles, and 51 cycles, respectively. For the 4PEs and 50MHz-bus system, the CDMAC without cache obtains 4.96× speedup compared to that with the 10MHz-bus in *scuircuit* (2.91 MFLOPS → 14.4 MFLOPS). Also, the CDMAC with cache obtains 4.1× speedup compared to that with the 10MHz-bus for *epb1* (37.1 MFLOPS → 153 MFLOPS).

The performance of CDMAC with cache is better than that of without cache. Fig. 7 shows that the CDMAC with cache attains 3.8× speedup compared to the CDMAC without cache in *psmigr_2* with 3PEs and 10MHz-bus (1.95 MFLOPS → 7.32 MFLOPS). For the 50MHz-bus system, its difference is appeared in the scalability. For example, the CDMAC with cache on the 50MHz-bus obtains 3.2× speedup from 1PE to 4PEs in *torso2* (57.5 MFLOPS → 185 MFLOPS) while that without cache obtains only 1.4× speedup (54.1 MFLOPS → 73.9 MFLOPS). This result shows that the CDMAC without cache exhausts memory bandwidth soon, and the cache reduces required bandwidth for the main memory.

For all bus speeds, the performance of *psmigr_2* and *scircuit* are worse than other matrices because the SELL format is not suitable for them. A random matrix needs more padding than a regular matrix when it is converted to the SELL format. This problem can be solved by changing store formats of the sparse matrix. However, the applicability of store formats is outside the scope of this paper.

IV. Cache Parameter Exploration

We confirmed that the CDMAC can process indirect accesses faster than a CPU through the experiments. However, suitable parameters for the data element cache are not considered yet. To reveal appropriate parameters of the cache for the sparse matrices, we examined cache performance on the matrices used in the evaluation. We made a simple software simulator for this evaluation to investigate the cache behavior in detail. We used both FIFO and LRU as replacement policies, but we show only the FIFO result in this paper because they have only a few differences. Fig. 8 shows miss ratio when the cache size and the line size are changed. Fig. 9 shows miss ratio when the cache size and the associativity are changed. Fig. 10 shows rates of increasing transfer bytes after cache.

As shown in Fig. 9, the miss ratio of the direct map is worse than the other associativities. In *dw8192* with 64 byte

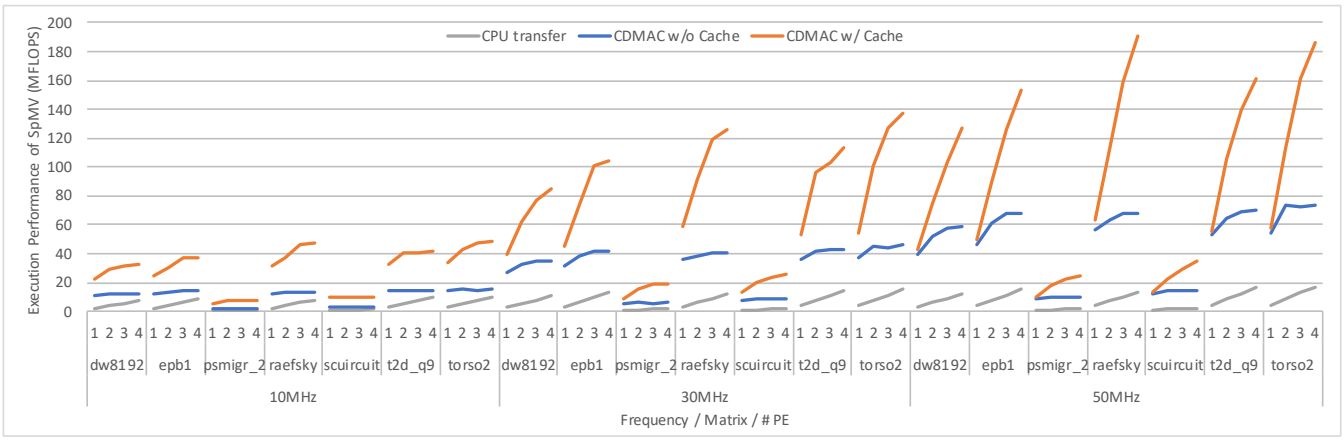


Fig. 7: Performance comparison between CPU, CDMAC w/o and w/ cache

line, the miss ratio of the direct map is 54% higher than that of the two-way set associative.

As shown in Fig. 8, the miss ratio of the cache for the band matrices has less relationship with the cache size. Especially, *epb1* and *t2d_q9* have no sensitivity for cache size. However, for the random matrices, the cache size has higher relationship with it. Especially for *psmigr_2*, the miss ratio was lower than 0.07% in 32KiB cache because the cache affords to have the whole of the input vector.

On the other hand, the bigger cache line intuitively seems to introduce smaller miss ratio resulting in better performance. However, as shown in Fig. 10, the bigger line increases the amount of transferred data. For example, in *scurcuit*, the miss ratio decreases to 28%, but the transferred bytes increases 2.3× as the line size is changed from 8 bytes to 64 bytes. Hence, the cache line should be determined by the trade–off between the amount of transferred data and the number of requests.

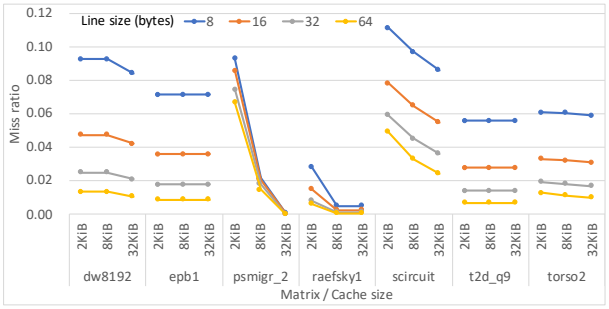


Fig. 8: Relationship between miss ratio, cache size and line size (Associativity is fixed to 2 way set associative)

V. Related work

A. SpMV optimization on general purpose processors and accelerators

Williams, et al. optimized SpMV on various multi–core platforms [17]. They tried several algorithm optimization techniques as well as cache optimization techniques.

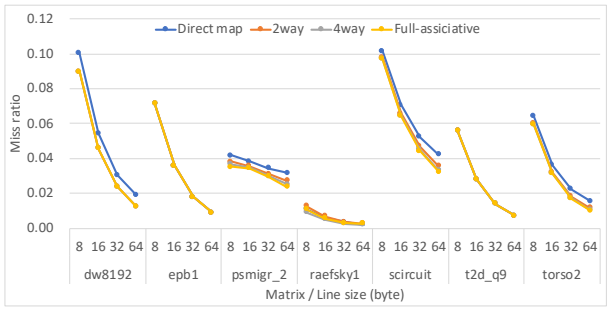


Fig. 9: Relationship between miss ratio, line size and associativity (Cache size is fixed to 8192KiB)

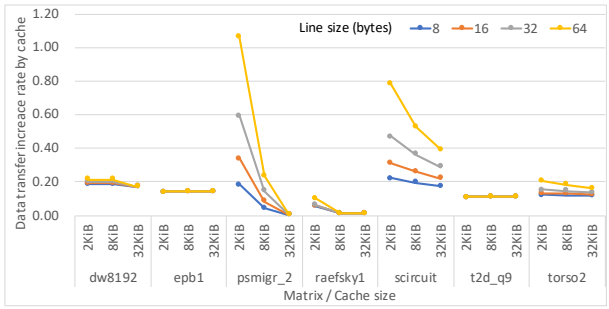


Fig. 10: Relative transfer bytes before and after passing through cache (Associativity is fixed to 2way set associative)

Bell and Garland implemented SpMV in some sparse–matrix formats on a GPU [18]. Also, Shan, et al. used a GPU for SpMV calculation of [2]. They reported 30× speedup compared to the CPU execution.

B. List vector in vector processors

Vector processors show high execution efficiency on the HPCG ranking list [19]. List vector is a mechanism to process indirect accesses in parallel. Its target is a rich memory system that can deal with many memory accesses simultaneously. List vector enables parallel execution by sending many small memory accesses using this many narrow memory channels.

However, this mechanism causes stalls because the load-store unit processes it directly. The CDMAC is decoupled from a processor instruction stream. Therefore, the processor is not involved in the low efficiency of indirect accesses.

C. Special hardware for SpMV

Some researches implemented SpMV calculation on FPGA for speedup. Nagar, et al. proposed a novel streaming reduction method [4]. Its system has a shared cache for matrices and local caches for an input vector. Fowers, et al. proposed a suited format of a sparse matrix for their implemented architecture [3]. Its system has many memory banks for the input vector and executes gather operations in parallel. The paper shows good power efficiency. Sadi, et al. co-optimized algorithm and hardware. The proposed hardware is a combination of an accelerator for SpMV and 3D stacked HBM [1]. Shan, et al., who used GPU for SpMV, also implemented on FPGA [2]. They reported 29× speedup compared to the CPU execution.

These solutions are not general for indirect accesses. Proposed CDMAC is a general solution for indirect accesses and it can be used in situations where applications and kernels are switched frequently.

D. DIMMnet

A special memory module for indirect accesses was proposed by Tanabe and et al [6], [20]. The module gathers elements and store them in it. After that, processors load the elements from it. This method can maximize effective bandwidth between the module and the processors. It is suitable for highly random accesses that a cache insufficiently work for. However, if the indices are duplicated, the transfer data contains the same data.

E. Indirect Memory Prefetch

Yu and et al. proposed a hardware prefetcher for indirect memory accesses [5]. They hide the long memory latency by detecting an index array and element size, and prefetching data according to them. Their method can be used transparently. On the other hand, our method does not require prediction. Also, Ainsworth and Jones proposed automatic software prefetch insertion by compiler [21]. Our method does not need additional calculations for software prefetching.

VI. Conclusion

In this paper, we proposed Cascaded DMA Controller (CDMAC), which can efficiently handle indirect memory accesses. It can overcome the latency bottleneck by adopting a streaming manner. We also proposed a suitable cache for CDMAC, which can withstand many cache misses. We implemented them and vector accelerators on an FPGA board and evaluated using sparse matrix–vector multiplication (SpMV). The CDMAC with the cache shows a maximum speedup of 17× compared to the CPU transfer.

References

[1] F. Sadi, L. Fileggi, and F. Franchetti, "Algorithm and hardware co-optimized solution for large spmv problems," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, Sep. 2017.

[2] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang, "Fpga and gpu implementation of large scale spmv," in *2010 IEEE 8th Symposium on Application Specific Processors (SASP)*, pp. 64–70, June 2010.

[3] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 36–43, May 2014.

[4] K. K. Nagar and J. D. Bakos, "A sparse matrix personality for the convey hc-1," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 1–8, May 2011.

[5] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 178–190, ACM, 2015.

[6] N. Tanabe, Y. Ogawa, M. Takata, and K. Joe, "Scaleable sparse matrix-vector multiplication with functional memory and gpus," in *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pp. 101–108, Feb 2011.

[7] K. Yamamoto, T. Shirakawa, Y. Oki, A. Yoshida, K. Kimura, and H. Kasahara, "Automatic local memory management for multicores having global address space," in *Languages and Compilers for Parallel Computing* (C. Ding, J. Criswell, and P. Wu, eds.), (Cham), pp. 282–296, Springer International Publishing, 2017.

[8] Intel, "Embedded Peripherals IP User Guide," 2019.

[9] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, pp. 1212–1221, June 2012.

[10] Intel, "Intel Arria 10 Device Overview," 2018.

[11] Intel, "Nios II Processor Reference Guide," 2019.

[12] Intel, "Nios II Custom Instruction User Guide," 2019.

[13] B. A. Adhi, M. Mase, Y. Hosokawa, Y. Kishimoto, T. Onishi, H. Mikami, K. Kimura, and H. Kasahara, "Software cache coherent control by parallelizing compiler," in *30th International Workshop on Languages and Compilers for Parallel Computing(LCPC)*, October 2017.

[14] RISC-V Foundation, "RISC-V "V" Vector Extension Version 0.7.2-draft-20190616," 2019.

[15] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

[16] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for gpu architectures," in *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, (Berlin, Heidelberg), pp. 111–125, Springer-Verlag, 2010.

[17] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pp. 1–12, Nov 2007.

[18] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–11, Nov 2009.

[19] R. Egawa, K. Komatsu, S. Momose, Y. Isobe, A. Musa, H. Takizawa, and H. Kobayashi, "Potential of a modern vector supercomputer for practical applications: Performance evaluation of sx-ace," *J. Supercomput.*, vol. 73, pp. 3948–3976, Sept. 2017.

[20] N. Tanabe, B. Nuttapon, H. Nakajo, Y. Ogawa, J. Kogou, M. Takata, and K. Joe, "A memory accelerator with gather functions for bandwidth-bound irregular applications," in *Proceedings of the first workshop on Irregular applications: architectures and algorithm, IA3 2011, Seattle, WA, USA, November 13, 2011*, pp. 35–42, 2011.

[21] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, (Piscataway, NJ, USA), pp. 305–317, IEEE Press, 2017.