Data-Level Parallelism Oriented Memory Access and On-Chip Buffering Mechanisms for a Loop Accelerator

Weikang Zhou\(^1\), Xianfeng Li\(^2\), Mingtao Chen\(^1\) and Yang Li\(^1\)

\(^1\)School of Electronic and Computer Engineering, Peking University, Shenzhen, China
\(^2\)Macau University of Science and Technology, Macau, China

Corresponding author’s Email: xifli@must.edu.mo

Abstract. Memory access latency is always a bottleneck for the performance improvement of data-intensive applications. Exploiting the memory access patterns of Data-Level Parallelism (DLP) is a promising way for loop accelerators to reduce the latency significantly. This paper proposes two DLP-oriented data provisioning mechanisms to alleviate memory access latency: 1) DLP-oriented memory access (DoMA) for efficiently utilizes the available memory bandwidth. 2) a data access patterns aware on-chip buffer (PABUF) for exploiting reuse in a user-transparent manner. Unlike those loop accelerators using traditional DMA to access global memory, DoMA efficiently reduces the transmission of useless data by adjusting the size of requests intelligently. In addition, PABUF, which manages data using DLP’s memory access patterns without software engineering efforts, allows the loop accelerators to access data in parallel. Experiments show that when our mechanisms are integrated into a loop accelerator based on Rocket Chip Coprocessor (RoCC), it can achieve 4.20x-10.65x (6.81x on average) speedups with negligible overhead of power and area compared to L1 Cache.

Keywords: Memory access, On-chip buffer management, Data-Level Parallelism, Parallel computing

1. Introduction
The improvement of computer performance is increasingly limited by memory wall and power wall. However, the new generation applications, such as machine learning, big data and cloud computing, are usually data-intensive and performance-demanding, which makes these problems more apparent. Fortunately, these applications typically spend execution time on some hot inner loops and contain a wealth of exploitable DLP. Exploiting memory access patterns are a promising approach for loop accelerators to significantly improve data access performance in a very energy-efficiency manner. Traditionally, the data access of Loop accelerators can be divided into hardware-managed caches to using locality implicitly and software-managed scratchpad memory to explicitly take advantage of memory access patterns.

The hardware-managed cache is an excellent storage structure for general-purpose processors, which automatically captures an application's temporal and spatial localities while being transparent to programmers and compilers. However, the cache is a suboptimal solution to provide data for loop accelerators. Loop accelerators often need to accelerate the innermost loop for parallel computing, and the cache line is too small to provide data at a high rate. Moreover, caches will lead to a higher area and power cost than scratchpad memory because of tag check, associativity, line size speculative read, etc.
Loop accelerators, which exploit data access patterns for extraordinary performance improvement and power efficiency, usually apply software-managed scratchpad memory as a storage structure. For example, DianNao[1] uses three respective scratchpads to store input, weight and output parameters, and GPGPUs apply shared memory for reuse and sharing data between thread blocks. The software-managed scratchpad memory usually utilizes DMA to transfer data from/to off-chip memory explicitly. In this way, the DMA can generate address by regular memory access patterns and decouples from the computational execution unit. However, the benefits of scratchpads come at a price. Firstly, scratchpads’ organization and management bring additional burdens to programmers. Second, DMA access to global memory will incur cache-coherent problems between the cache of general-purpose core and accelerator scratchpad memory. The consistency problems will further increase the efforts of software engineering.

For consistent global memory access, D^2MA[2] can generate fixed-size coalesced memory requests quickly to global memory. These requests will be pipelined through a snoopy protocol to maintain consistency and data transmission efficiency. However, the fixed request size is low utilization of memory bandwidth for non-unit stride memory access, as the data segments of global memory requests may include useless data, especially when both the stride and the request size are larger.

To overcome the above data-providing mechanisms’ shortcoming, we proposed DLP-oriented data provisioning mechanisms for loop accelerators. When implementing our data provisioning mechanisms in a loop accelerator, the experiment results show that it achieves a 6.81x performance speedups on average and only increased 2.33% area and 4.38% power overhead compared to hardware-managed cache. In summary, the main contributions of this work include:

- We proposed a new off-chip memory access mechanism called DLP-oriented data access(DoMA) to overcome the low efficiency of D^2MA for non-stride access patterns.
- A data access patterns aware on-chip buffer(PABUF) will be designed and implement to automatically organize and manage data referring to DLP’s memory access patterns.
- To test our data provisioning mechanisms, we integrate these mechanisms into a loop accelerator based on Rocke Chip Coprocessor(RoCC)[3] and conduct a series of comprehensive benchmarks.

2. DLP-oriented data provisioning mechanisms

The objective of our DLP-oriented data provisioning mechanisms is to make full use of the memory available bandwidth and provide data for the loop accelerators in parallel with low software management cost. DLP applications have a variety of memory access patterns. To better exploit DLP applications’ parallel potential, we will first classify DLP applications’ memory access patterns.

2.1. Classification of memory access patterns

The memory access patterns of DLP applications can be divided into regular and irregular. Regular memory access patterns usually have great parallel potential because of simple data access and control flow. The commonly used regular memory access patterns are unit stride and non-unit stride. As the data is continuously and tightly stored in memory, the unit stride case can provide data for loop accelerators at a high throughput rate and is an ideal memory access pattern. Figure 1(a) shows unit-stride access pattern by element-wise vector multiplication without inter-iterations data dependence. Loop accelerators can access the vector elements simultaneously for parallel computing. Unlike unit stride access, non-unit stride memory access has a fixed step, and the vector elements are stored in memory dispersely. As shown in Figure 1(b) code fragment that comes from the innermost loop of matrix multiplication, array B will be accessed by a non-unit stride pattern and array A by unit stride pattern.

![Figure 1. Memory access patterns with parallel capability.](image_url)
Different from regular patterns, irregular memory access patterns of DLP applications, either with data dependence or complex control flows, are challenging for loop accelerators to accelerate. In this paper, we will solve some of the most commonly used irregular memory access patterns, and the rest will be our future work. For data dependence of variables between iterations, like variable 'c' in Figure 1(b), we will temporarily save it in the loop accelerators. Another irregular memory access pattern is the branch in the loop body (Figure 1(c, d)). A mask generated by the branch condition will be used to determine which correct data to be written back.

2.2. DLP-oriented memory access
Our DoMA is based on D2MA[2], which improves GPU global memory access performance by decoupling address generation and shader's computational unit. However, unlike D2MA using a fixed request size, our DoMA can resize requests flexibly, which is more intelligent and achieves higher bandwidth utilization.

2.2.1. DoMA for unit stride access. DoMA for unit stride is similar to D2MA. A bulk of continuous data can be represented by the base address, data type and request data number. According to this information, the request address using to access off-chip memory continuously can be calculated by Figure 2(a) logic. To maintain cache consistency, DoMA will be split a memory request into many segments. The Segment Size is a fine-grained request size compared to DMA, and it's usually a power of two and is not larger than the cache line size (64B in Rocket Core[3]). And the ByteCnt, Total Bytes and CurAddr denote the number of bytes currently fetched, the total bytes of request and the address of the current request fragment, respectively. The NextAddr is the output of address generation and is controlled by ByteCnt and Total Bytes. When ByteCnt is equal to Total Bytes, the address generation is invalid and means that the request is complete. As the address generation decouples from the computational execution module, a series of consecutive global memory access requests can be pipelined and overlapped with the execution unit.

![Figure 2.](image)

(a) Unit stride address generation  (b) Non-unit stride address generation

2.2.2. DoMA for non-unit stride access. The low efficiency of D2MA is fixed segment size for non-unit stride cases, which can be illustrated in Figure 3. A fixed-size segment of 64B will move 16 elements from/to off-chip memory. For Figure 3(a) memory access, when the element size is 4 bytes with a distance of 2 strides between each element, a segment data of off-chip memory request can gain 50% of interesting data, and it's relatively efficient. However, when the stride is no less than 16, only one of 16 elements is valuable, which results in a waste of precious memory bandwidth.

![Figure 3.](image)

(a) ElemSize = 4B, Stride = 2, Segment size = 64B  (b) ElemSize = 4B, Stride = 16, Segment size = 64B

Figure 3. Non-unit stride memory access with different strides.
In order to further improve the utilization of available memory bandwidth, we first determine the segment size according to the non-unit stride memory access patterns. It can be calculated by equations (1)(2). Since there is a long latency in initializing a request, our principle is to reduce the number of useless data transmissions as much as possible while minimizing the number of requests. As shown in Figure 2(b), the address generation is similar to the unit stride case, except non-unit stride will change segment size when the stride is too large for saving the memory bandwidth.

\[
\text{ElemNum} = \left\lfloor \frac{\text{cache line size}}{\text{stride} \times \text{ElemSize}} \right\rfloor
\]

(1)

\[
\text{Segment Size} = \begin{cases} 
\text{ElemSize}, & \text{ElemNum} = 1 \\
\text{ElemNum} \times \text{ElemSize} \times \text{stride}, & \text{ElemNum} > 1 
\end{cases}
\]

(2)

2.3. Data access patterns aware on-chip buffer

There is a common design method of loop accelerators that we can find in vector processors and XLOOP[4], that is, unrolling the innermost loop entirely or partially and using processing elements (PEs) array to perform each iteration in parallel. And the design details of a loop accelerator will be described in section 3. This section will present the details of on-chip memory designs for this kind of loop accelerator, called data access patterns aware on-chip buffer (PABUF). For parallel data acquisition and reuse, PABUF, a multi-bank on-chip SRAM with a hardware table to record the memory access patterns, can store the frequently used data (Figure 4). It is worth mentioning that, unlike traditional software-managed scratchpad, PABUF can organize and manage data without manual efforts according to the Used Table. Moreover, PABUF can also write back the interesting data through the mask.

2.3.1. Data organization. The data organization follows three principles: 1) make full use of precious local memory, 2) provide multi-port parallel data acquisition with a low-overhead on-chip network that satisfies the common parallel permutations \(\pi_k(n)[5]\) for loop accelerators, 3) data types transparency including single precision and double precision. To achieve these functions, we apply a multi-bank buffer to multi-port parallel access and store data of interest (Figure 4(a)). It’s important to note that PABUF stores data tightly, no matter unit stride or non-unit stride patterns, to save on-chip memory space. As for the on-chip network, we add data type processing logic to the last level of the Omega network[5] (Figure 4(b)), which can split the first four data of 8 bytes into 16 single precision data, so single- and double-precision floating-point data types can be both supported by our hardware.

**Figure 4.** The structure of PABUF

2.3.2. PABUF management. The PABUF management is the responsibility of the Controller in Figure 4(a), a finite state machine. We apply the Used Table to record and perceive memory access patterns.
When a loop accelerator wants to read data, it will send a request message including base address, stride, data type and the number of elements to the Controller. The Controller will first find the matching pattern item in the Used Table according to the request message. If a record is found, the data in Multi-bank Buffer will be accessed; otherwise, the Controller will send a request to DoMA to access the off-chip memory with a long latency and then update Used Table and Multi-bank Buffer when the data returns. Note that when all entries are occupied, a random record will be replaced according to the replacement cost. That is, the dirty table item will be replaced at last. Moreover, the data fragment that each table item points to is usually larger than the size of the loop accelerators request, which is for data prefetching and better reuse. Finally, whether the request that matches the Used Table's entry needs to meet the condition that the dataset represented by the table entry completely contains the requested data, which can be judged by the memory access patterns. As for those partially matched table entries, they need to be set invalid or even be written back when the dirty field is true.

3. Experimental methodology

3.1. Loop accelerator

Since our primary purpose is to introduce the data provisioning mechanisms, the loop accelerator design and implementation will be briefly described in this section.

The Rocket Chip Coprocessor(RoCC) is a well-known coprocessor interface of Rocket Chip Generator[3] in the RISC-V architecture community. Based on RoCC, we design and implement a loop accelerator to test our data provisioning mechanisms' performance. As shown in Figure 5, the loop accelerator integrates DoMA and PABUF modules for accessing data efficiently. As a comparison, the loop accelerator also provides L1 Cache interface to access data. The user-defined extended RoCC instructions will be passed to RoCC. These instructions will be decoded and generate the control signal to drive the PEs array and data access. Under the control of the signal, PABUF can provide data for the PEs array in parallel, and when the calculation is complete, the target registers can be written back to PABUF by the control of the mask.

![Figure 5. A loop accelerator based on RoCC overview.](image)

3.2. Simulation methodology

The Rocket Chip Generator provides a cycle-level software RTL simulation based on a Verilator tool. Besides, The Rocket Chip Generator enable to use DRAMSim2[6] as an off-chip memory simulation, a cycle-level accurate model of a DRAM simulator. We will employ these cycle-level accurate simulations to evaluate loop accelerator performance.

4. Experiment results

We evaluate the memory access structure and compare its performance, power consumption and area cost with the L1 Cache with the same memory size of 32KB. There are 32 items in the Used Table, and there are 64 PEs for parallel computing. Therefore, we apply 64 banks to store data and provide data for the PEs array in parallel.
4.1 Performance evaluation
Our experiments first evaluate DoMA performance through sending requests to DoMA directly and bypassing the PABUF, and the same requests will also be sent to L1 Cache for comparison. We apply RoCC instructions to expand six benchmarks’ parallelizable innermost loops from PolyBench[7] and employ these benchmarks to evaluate our data access mechanisms' overall performance.

4.1.1. DoMA vs. L1 Cache. The memory access performance can be shown in Figure 6. The Hot$ means that the L1 Cache has been warmed up before memory operation. On the other hand, Cold$ indicates that the L1 Cache has been flushed before memory operation. For example, in the Figure 6 legend, DoMA/Cold$(read)$ donates the speedup ratio of DoMA and L1 Cache when they load the same data from off-chip memory. When the cache is flushed, the cache's performance is obviously behind the DoMA and DoMA achieves 3.11x-9.73x performance improvement. Although L1 Cache has stored most data when the stride is small in the Hot$ case, DoMA is better than L1 Cache. The reason is that DoMA amortizes the request latency to each element of a segment data request. However, each L1 Cache request just returns an element. Since L1 Cache is continuously stored in a cache line, the data of interest will decrease when the stride increases. However, DoMA can adjust the request size to save memory bandwidth when the stride is relatively large. This is why DoMA performance outperforms L1 Cache in the large stride cases.

![Figure 6](image)

**Figure 6.** Comparison of performance between DoMA and L1 Cache off-chip memory access.

4.1.2. Benchmarks.

![Figure 7](image)

**Figure 7.** Speedup of ours data provisioning mechanisms over the L1 Cache.
We compare the overall performance of our data provisioning mechanisms with the L1 Cache, and the results are shown in Figure 7. The DoMA case means that the loop accelerator accesses global memory directly without local memory. The average performance of DoMA is about 3.68x better than L1 Cache and 6.81x when the PABUF is used (legend Ours of Figure 7). It is worth mentioning that our data provisioning structure’s performance is not only proportional to the data reuse scale of the applications but also relates to the DLP memory access patterns. For example, the kernel of covariance will repeatedly manipulate the columns of the matrix. As our structure can be aware of the memory access pattern, the loop accelerator can obtain these columns directly from the PABUF in parallel once the corresponding column data is saved in PABUF. However, the L1 Cache will be invalidated because there is usually a large data interval between adjacent column data.

4.2 Power and area estimation

We synthesize the chisel-generated Verilog with Synopsis Design Compiler and TSMC 65nm technology library for power and area estimation. The results show that our power and area overheads are a little larger than L1 Cache, as Table 1 presents. Compared to L1 Cache, the area and power overhead mainly come from a multi-bank on-chip buffer, as multiple ports are provided for parallel data access. Another reason is that the omega network needs to occupy a certain amount of overhead. However, the area and power overhead are negligible compared to the performance improvement.

Table 1. The area and power overhead.

|                  | L1 Cache | Our structure | Our structure/L1 Cache |
|------------------|----------|---------------|------------------------|
| Area (mm²)       | 1.29     | 1.32          | 102.33%                |
| Power (mW)       | 115.03   | 120.07        | 104.38%                |

5. Conclusions

This paper presents parallel data provisioning mechanisms for loop accelerators. This mechanism uses memory access patterns of DLP’s applications, including the unit stride and the non-unit stride for memory access and data management. And its data providing process consists of a DoMA for off-chip memory access and a PABUF for data reuse. In order to decouple from the computational execution unit, the DoMA enables to generate coalesced memory requests referring to memory access patterns. Compared with previous DMA-like data access, the DoMA can adjust the segment size of off-chip memory requests to gain better memory bandwidth utilization. The PABUF is a hardware-managed on-chip buffer. The data stored in PABUF are organized and managed by a table that records memory access patterns. As a result, the PABUF can be aware of loop accelerators’ memory access by comparing the memory access patterns. We have implemented these data provisioning mechanisms on a loop accelerator base on RoCC, and the experimental results show a significant speedup for DLP applications. As our data provisioning mechanism enables us to provide a unified interface for data access, it’s a feasible solution to significantly improving the performance of loop accelerators data access.

6. References

[1] Chen T, Du Z, Sun N, Wang J and Teman O 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. SIGARCH Comput. Archit. News 42 1 269–284
[2] Jamshidi D A, Samadi M, and Mahlke S 2014. D²MA: accelerating coarse-grained data transfer for GPUs. Int. conf. on Parallel architectures and compilation. ACM. pp 431–442
[3] Asanović K, Avižienis R, Bachrach J, et al 2016. The Rocket Chip Generator, Technical Report UCB/EECS-2016-17, EECS Department University of California Berkeley
[4] Srinath S, Ilbeyi B, Tan M, Liu G, Zhang Z and Batten C 2014. Architectural specialization for inter-iteration loop dependence patterns. Int. Symp. on Microarchitecture. IEEE Computer Society(USA) pp 583–595
[5] Lanfant J 1978. Parallel permutations of data: a benes network control algorithm for frequently used permutations. IEEE Trans. Comput. 27 7 pp 637-647
[6] Rosenfeld P, Cooperbalis E and Jacob B 2011. DRAMSim2: A Cycle Accurate Memory System simulator[M]. IEEE Comput. Archit. Lett. 10 1 pp 16–19
[7] Pouchet L N 2014. Polybench: the polyhedral benchmark suite. URL: http://www.cs.ucla.edu/~pouchet/software/polybench/

Acknowledgments
The work is sponsored by Shenzhen Science and Technology Innovation Committee(SZSTI) under the grant for basic research(NO. JCYJ20170818090006804).