An Access-Pattern-Aware On-Chip Vector Memory System with Automatic Loading for SIMD Architectures

Tong Geng∗, Erkan Diken†, Tianqi Wang∗†, Lech Jozwiak∗‡, Martin Herbordt∗
∗Department of Electrical and Computer Engineering; Boston University, Boston, MA, USA
†Department of Physics; University of Science and Technology of China, Hefei, China
‡Department of Electrical Engineering; Eindhoven University of Technology, Eindhoven, the Netherlands

Abstract—Single-Instruction-Multiple-Data (SIMD) architectures are widely used to accelerate applications involving Data-Level Parallelism (DLP); the on-chip memory system facilitates the communication between Processing Elements (PE) and on-chip vector memory. It is observed that inefficiency of the on-chip memory system is often a computational bottleneck. In this paper, we describe the design and implementation of an efficient vector data memory system. The proposed memory system consists of two novel parts: an access-pattern-aware memory controller and an automatic loading mechanism. The memory controller reduces the data reorganization overheads. The automatic loading mechanism loads data automatically according to the access patterns without load instructions. This eliminates overhead of fetching and decoding. The proposed design is implemented and synthesized with Cadence tools. Experimental results demonstrate that our design improves the performance of 8 application kernels by 44% and reduces the energy consumption by 26%, on average.

Keywords-component—SIMD; Vector Data Memory; Data Access Pattern; Automatic Loading

I. INTRODUCTION

As applications become ever more complex, they impose growing demands on power and real-time performance. Data-Level Parallelism (DLP) is critical in many applications, especially video and image and signal processing. DLP is almost universally exploited through Single-Instruction Multiple-Data (SIMD) hardware support [1], whether through vector extensions, GPU blocks, or directly with dedicated SIMD architectures [2]–[4]. The latter is our focus here.

In SIMD architectures, the inefficiency of the on-chip local memory system is often a bottleneck [5]–[8]. It has been found, however, that local memory systems that can be tuned to the applications’ data access patterns can substantially improve performance [9]–[13]. We refer to such memory systems as being data-access pattern aware. We have observed two problems that can be solved with such memory systems.

• Data reorganization overhead: The challenge here is especially with respect to discontinuous data accesses. For such applications, it is necessary to reorganize data either 1) when the data is written into the local memory of the SIMD processors or 2) in the register files of the processor. Both options are costly in power and performance.

• Data access overhead: It should be possible to reduce data access overhead by eliminating many memory access instructions (and their fetching and decoding).

To solve these problems, we propose a 1D data access-pattern-aware memory system that can work not only with continuous but also discontinuous (regular) data access patterns. The contributions of our work are twofold:

• Access-pattern-aware memory controller: We propose a 1D memory controller that generates data addresses according to the patterns extracted from application code. Data can be accessed from local data memory banks in parallel without data reorganization.

• Automatic-loading architecture: We propose to reduce data access overhead, again, by using application-awareness. In the automatic-loading design, the pattern parameters are kept in the processor and data is accessed from automatically without executing load instructions. Experimental results demonstrate that our design improves the performance of 8 application kernels by 44% and reduces the energy consumption by 26%, on average.

The rest of this paper is organized as follows. Section II discusses related work. Section III gives preliminaries of a SIMD architecture which is augmented by the memory controller proposed in this paper. Section IV analyzes the problems of data reorganization and data access overhead. Section V explains our designs in detail. Section VI provides experiments and discusses the results.

II. RELATED WORK

There has been much previous work with respect to on-chip memory systems for SIMD architectures. In [14], a two-dimensional access-pattern-aware memory controller is proposed. This memory controller supports access patterns which are composed of rectangle groups. For other patterns, however, the memory controller requires various and complex data layouts for conflict-free access; these substantially diminish the benefit. In [15], a 2D shift register file is used to feed data to PEs where the data have sufficient reuse. According to the access pattern width and length, the data stored in corresponding columns and rows of the register file are accessed. Similarly, in [16], a 2D barrel shifter is used as the interface between register file and local memory, working in the same way as the 2D shift register file in [15]. For the applications with a single group of rectangle access pattern, i.e. a continuous data access pattern, these two designs are efficient. However, for the applications which require discontinuous access of data, the 2D shift register and barrel shifter perform poorly. With our design, both continuous
data access and discontinuous data access can be handled efficiently. In [11], a 1D data access pattern aware memory controller is proposed, but again, is for continuous data access. In [17], an access-pattern-aware method for CTA clustering is proposed for GPU.

Our memory system supports all kinds of regular data access patterns including both continuous and discontinuous. In addition, using our memory system, data are accessed from on-chip memory automatically based on the previously acquired data access pattern. Thus, after the first load instruction is executed, no more are needed.

III. BACKGROUND

This section introduces the SIMD architecture which we will use as the baseline for this work. It was first proposed in [18] and is shown in Figure 1. It consists of two parts, a control processor (CP) and a wide one dimensional (1-D) array of PEs. CP and PE array run in lock-step. It results in a VLIW processor with one scalar issue slot for the CP and one vector issue slot for the PE array [19] [20]. As shown in Figure 1, each PE has its own data memory bank and can only access data from that memory bank. The PEs can read and exchange data from the neighborhood via a circular neighborhood network [18]. For a SIMD processor with N PEs, PE0 can access the data in the Register Files (RF) of PE1 and PE(N − 1) directly. If it is required for PE0 to access data from PE2, it is performed with 2 hops. Thus, a proper data layout of the vector data memory is important to getting high performance. There is no interface implemented between the local vector memory and shared main memory. Hence, to get a proper data layout, data need to be reorganized within PEs after they are written to RFS.

Figure 2 shows a block diagram of the 5-stage PE pipeline. The 5 stages are as follows: IF (Instruction Fetch), ID (Instruction Decode), EX1 (Execution 1), EX2 (Execution 2), and WB (Write Back). The LSU (Load Store Unit) is composed of AGU (Address Generation Unit) and DMEM blocks, and works in the EX1 and EX2 stages. The AGU calculates the address in one cycle and transfers it to DMEM. In the EX2 stage, the target data is accessed from the memory banks.

There is data forwarding from the results of the ALU and MUL, and the data accessed from DMEM, into the ID stage. We focus mainly on the hardware structure of AGU and control logic for data access.

IV. PROBLEM ANALYSIS

In this section, the two problems mentioned in Section I are explained in detail.

A. Reducing Data Reorganization Overhead

For our baseline SIMD processor, each PE can only access data from its own memory bank and communicate with its left and right neighbors [19]. If the data layout of the vector data memory is not in the required order, data reorganization is necessary. Data reorganization can be performed with hardware or software approaches. In the hardware approach, the data reordering hardware interface is used between local and shared main memory. In the software approach, data reorganization is performed by the PEs. Both solutions bring overhead such as extra latency and energy consumption. For the baseline SIMD, the hardware interface for data reorganization between shared main memory and vector data memory is not implemented. Thus, data is reorganized within processing elements.

```c
for (i=init=0, i<limit, i=i+SL)
{
    X[i]=C0*X[i+a]+C1*X[i+b]+ C2*Y[i+c];
}
```

Fig. 3. Generic form of a loop code in C
To provide some intuition of reorganization overhead, we use a generic loop kernel as an example (see Figure 3). We use a 4-PE SIMD processor to accelerate this kernel. The data of array X are linearly mapped to the 4-bank local data memory. The kernel is vectorized so the first 4 iterations are executed in parallel. Since each PE can only read data from its own memory bank, PEi can only read data of X[4*n+i](n ∈ N; i ∈ 0, 1, 2, 3) from its own DMEM directly. The contents of the RFs of the 4 PEs—after direct loading without any data reorganizations—are shown in Figure 4.

Fig. 4. RF contents of 4 PEs without data reorganization.

To execute the workload of the first 4 loops in vectorized manner (a corner turn), X[a+SL*i], X[b+SL*i], X[c+SL*i] need to be cached in the RF of PEi. The required contents of RFs are shown in Figure 5. To generate the required RF contents from the original shown in Figure 4, much work needs to be performed by the PEs.

Fig. 5. RF contents of 4 PEs to support vectorization: we assume a=0, b=1, c=2, SL=3.

Regular data access patterns can be extracted from application codes, especially from loop structures. A generic form of a loop code in C is shown in Figure 3. The access pattern extracted from this loop is shown in Figure 7. As 4 PEs work in parallel, the first 4 iterations are processed in the vectorized manner. In the first access, X[a], X[a+SL], X[a+2SL], X[a+3SL] are loaded, followed by X[b], X[b+SL], X[b+2SL], X[b+3SL], and then Y[c], Y[c+SL], Y[c+2SL], Y[c+3SL]. After these 3 accesses, all the required data of X and Y arrays for the first 4 iterations are fed into processors. As shown in Figure 7, in every access, 4 data with SL as the address step size are loaded from memory. As mentioned in Section IV, using the baseline SIMD, with linear data layout in DMEM, data reorganization is inevitable. To reduce the data reorganization overhead, a memory controller is proposed which replaces the original AGU in the baseline architecture.

A. Design of access-pattern-aware memory controller

To reduce the data reorganization overhead discussed in Section IV-A, an access-pattern aware memory controller is designed and implemented. With this memory controller, the linearly or linearly-like mapped data in DMEM can always be written into PEs with expected RF contents without memory access conflict or reconfiguration.

Fig. 7. Data access pattern extracted from the loop in Figure 3.

B. Reducing Data Access Overhead

For most loops, the data access pattern is regular and can be extracted from the code. We support memory access without explicit load instructions. Removing load instructions allows IF and ID stages for those instructions to be removed. We consider their cost as overhead. Figure 6 shows the relative energy costs of different stages [15]. The width of each block is proportional to the energy cost.

Fig. 6. Relative energy costs of different stages. The width of each block is proportional to the energy cost.

V. DESIGN OF THE PROPOSED MEMORY SYSTEM

In this section, we describe the design of the proposed 1D access-pattern aware vector data memory system. The first subsection covers the design of the access-pattern-aware memory controller while the second subsection explains the design of the automatic load architecture.

A. Design of access-pattern-aware memory controller

To reduce the data reorganization overhead discussed in Section IV-A, an access-pattern aware memory controller is designed and implemented. With this memory controller, the linearly or linearly-like mapped data in DMEM can always be written into PEs with expected RF contents without memory access conflict or reconfiguration.
addresses with SL as the step size. If B is assumed to be the base address, B+SL, B+2SL, and B+3SL will be computed as other addresses for data access. These 4 addresses will be transferred to the following modules.

**MEM Bank ID Generator (MBIG):** To access data X[B], X[B+SL], X[B+2SL], and X[B+3SL] from 4 memory banks in parallel without conflict, it is necessary to calculate the IDs of the memory banks for all target addresses generated by the AG. When SL is odd, conflict-free parallel access is always possible with a linear data layout; this has been proven in [21]. The memory bank IDs of addresses can be computed according to Equation (1). Here, \( N_{bank} \) is the number of memory banks and \( Addr \) is the address generated by the AG.

\[
BankID = Addr \mod N_{bank} \quad (1)
\]

![Fig. 9. Linear data layout when SL is 4.](image)

In the case of even SL, conflicts of memory access are inevitable using linear data layout. To avoid these conflicts, we adjust the linear data layout with multiple data shifts to a linear-like data layout, with which conflict-free parallel access becomes possible. We use a motivational example to explain how the data layout is adjusted. Assume that the SL is 4 and the base address is 0. Through AG, addresses of X[0], X[4], X[8], X[12] are generated and fed to MBIG. As shown in Figure 9, using a linear data layout, all of these 4 data are mapped to the same memory bank, the bank for PE0, which makes parallel access impossible. It is demonstrated in [21] that these conflicts during access can be eliminated through simple intra-row data shifting.

![Fig. 10. Linear-like data layout when SL is 4.](image)

The number of shift steps is calculated according to Equation (2). Note that the value of SL is decomposed into \( 2^s \) multiplied by an odd number. \( S_2 \) in Equation (2) denotes \( 2^s \). During the adjustment of data layout, there will be no data exchange between rows and all the data keep their row IDs. For our example with SL of 4, the shifted layout is shown in Figure 10. After data shifting, X[0], X[4], X[8], and X[12] are in different banks and free-conflict parallel access becomes possible. For even SL cases, even though linear layout cannot be directly used, it becomes useful after several data shifts (linear-like).

\[
Shift_{step} = (RowID \times N_{bank} + \left\lfloor \frac{RowID}{S_2/N_{bank}} \right\rfloor) \mod N_{bank} \quad (2)
\]

We implement a hardware interface with a shifting function between shared main memory and local vector data memory. With this interface, the data from off-chip main memory can be loaded into the local memory with a required linear-like data layout. The memory bank IDs of target addresses can be computed according to Equation (3).

\[
BankID = (Addr + \left\lfloor \frac{Addr/N_{bank}}{S_2/N_{bank}} \right\rfloor) \mod N_{bank} \quad (3)
\]

**Row Address Generator (RAG):** Besides memory bank IDs, the row addresses are also necessary for data accesses. As mentioned above, for all cases of SL, the row addresses are kept the same as in the linear layout. Therefore, the row addresses are always computed according to the same function, as shown in Equation 4.

\[
Row_{Addr} = \left\lfloor \frac{Addr}{N_{bank}} \right\rfloor \quad (4)
\]

Overall, with the SL extracted from the application codes and operands from the ID stage, the address generator module of the memory controller generates 4 addresses with SL as the step size. With the memory bank ID generator and row address generator, all addresses generated by the address generator module get their corresponding memory bank ID and row addresses. After traversing the reorder shuffle unit shown in 8, the row addresses are transferred to the target DMEM banks. This results in an appropriate data access in the EX2 stage. With the proposed access-pattern-aware memory controller, data with either linear or linear-like layout can always be accessed in parallel without conflict.

**B. Design of automatic load architecture**

In Figure 3, a generic loop structure is used to show a regular data access pattern. As shown in Figure 7, the data which are going to be used can be known in advance. For example, when the data in blue are being accessed, we already know that the next load command will request the access of the data shown in red. It is therefore possible to make the SIMD processor load data automatically based on data access patterns.

The access-pattern-aware memory controller shown in Figure 8 is augmented with an automatic load mechanism. The augmented memory controller is shown in Figure 11. A new AG module for automatic-load mechanism is added. All the access pattern related parameters are kept in special registers of this new module. There are 4 main parameters: SL, Offsets, initial value, and limitation. The new AG module generates addresses for all PEs according to the pattern-related parameters with the control of 1-bit flag from the extension of the original ISA.

With this auto-load address generator structure, load instructions are eliminated. In addition, data access and computation operations can be performed in parallel, which improves the computation speed and (Instruction Level Parallelism) ILP of the SIMD architecture.
The optimized structure of a single PE is shown in Figure 12. The computation instruction is fetched in IF stage. At the same time, the flag for auto-load AG, Instruction [28], is read by the shared memory controller. If the flag enables the automatic-loading process, new addresses will be automatically generated before the fetched instruction reaches ID stage. When the fetched command is decoded in the ID stage (in the next clock cycle), the addresses generated in the previous cycle will be read by DMEMs and the target data will be accessed. At the same time, the accessed data can be bypassed and used as one of the operands. Therefore, by using the optimized auto-load mechanism an instruction can use its 28th bit to request new data from DMEMs and use the accessed data as one of its operands. In this way, the data access and computation operation are performed in parallel, which improves the ILP of our architecture resulting in higher performance.

VI. EXPERIMENTAL RESULTS

In this section, eight different kernels of SPEC2000, including 4*2 Convolution, SAD, SIFT Gaussian, Integrate predictors, Hydro fragment, SIFT DoG, Inner product and Tridiagonal elimination, are used to evaluate the performance and energy efficiency of the proposed architecture by comparing it with the baseline. Full name and abbreviations of these kernels are listed in Fig.14. Both the original and the proposed SIMDs are equipped with a 1KB instruction memories and 4KB vector data memories.

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Convolution</th>
<th>Scale of Absolute Differences</th>
<th>Scale-invariant Feature Transform</th>
<th>Scale-invariant Feature Transform</th>
<th>Intra Prediction</th>
<th>Inter Prediction</th>
<th>Hydro Fragment</th>
<th>Inner Product</th>
<th>Tri-diagonal Elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abbreviation</td>
<td>CDF</td>
<td>SAD</td>
<td>SIFT Gauss</td>
<td>SIFT DoG</td>
<td>inter</td>
<td>inter</td>
<td>inter</td>
<td>inter</td>
<td>inter</td>
</tr>
</tbody>
</table>

Fig. 13. Synthesis results of proposed and original 4-PE SIMD designs.

In our experiments, the baseline architecture and the proposed design are synthesized with a 40nm CMOS technology using Cadence tools. Memory banks are not synthesized. The energy consumption of the memory banks is simulated using CACTI. For all other parts of both architectures, the energy consumption is measured using the RTL Compiler with 410MHz as the clock frequency. The synthesis results of the proposed design and the original design are shown in Figure 13. The results indicate that the access-pattern aware memory system does not decrease the highest achievable frequency. However, the cell area and power consumption are increased by 5.7% and 12.2%, respectively.

Fig. 14. Abbreviations of 8 kernels from SPEC2000 used in this paper

![Fig. 15. Numbers of clock cycles which are taken to execute 8 kernels from SPEC2000 using original design and proposed design](image)

![Fig. 16. Energy consumed by original and proposed design](image)
main memory. As mentioned in Section III, there is no such interface implemented in the baseline SIMD architecture. To make fair performance and energy consumption comparisons, data reorganizations both for the original design and for data shifting in the proposed design are performed by PEs. The numbers of clock cycles to execute different kernels using the original and proposed architectures are shown in Figure 15. The red blocks indicate the data reorganization overhead of original design. The yellow blocks indicate the data shifting overhead of the proposed design. For the kernels with even SL, like SAD and Convolution, the linear data layout in vector data memory needs to be adjusted to the required linear-like layout through data shifting before the computation starts.

Execution of all kernels using the proposed architecture takes fewer clock cycles than the original architecture, as the load instructions are eliminated and executed in parallel with operation commands. For SIFT Gaussian, SAD, and Convolution, interleaving data layout is used for data reuse. Thus, data reorganization is needed and overhead occurs. The extra improvements come from the pattern-aware memory controller, which can eliminate some of the data reorganization instructions.

For kernels with high-density load commands, more benefits can be obtained from the proposed automatic-load architecture. For kernels with special data layout, benefits can be obtained from the access-pattern-aware memory controller. For Inner Product, the performance is improved by $2.3 \times$. For 2*4 Convolution, SAD, and Gaussian kernels, data are reused efficiently in the register file and the load command density is low. As a result, the performance cannot be improved very much from the auto-loader, but is improved due to the access-pattern-aware memory controller.

The energy consumption results are shown in Figure 16. Again, the kernels with high-density load commands get more benefits from the auto-load architecture. The kernels which need data reorganization get extra benefit from the access-pattern-aware memory controller.

Overall, the performance of the proposed architecture is $1.2 \times$ to $2.3 \times$ better than the original design and 44% better on average. The energy consumption is reduced by 26% on average.

VII. CONCLUSION

In this paper, an access-pattern-aware on-chip vector memory system with an automatic load mechanism is proposed to improve the efficiency of the SIMD architecture proposed in [18]. With the access-pattern-aware memory controller, data can always be accessed in parallel without conflict according to the access pattern which is extracted from application codes. The required data layout of memory banks is easy to generate. For odd SL cases, linear data layout can be used directly. For even SL cases, the required data layout is converted with intra-row data shifting. This layout transformation can be easily handled by a hardware interface between shared main memory and local data memory. With the proposed memory controller, no data reorganization is needed, so that overhead is reduced greatly. With the automatic-load architecture, the SIMD processor accesses data from DMEM automatically according to the data access pattern. The fetch and decode cost of load instructions is reduced and the ILP improved. Overall, the auto-load architecture reduces the data access overhead, and brings better performance and energy efficiency. Compared with the baseline design, our contributions improve the performance by 44% and reduce energy consumption by 26%, on average.

ACKNOWLEDGEMENT

This work was supported in part by the National Science Foundation through Awards CNS-1405695 and CCF-1618303/ 7960; by grants from Microsoft and Red Hat; and by Altera through donated FPGAs, tools, and IP.

REFERENCES


