A Specialized Low-Cost Vectorized Loop Buffer for Embedded Processors

Libo Huang, Zhiying Wang, Li Shen, Hongyi Lu, Nong Xiao, Cong Liu
School of Computer, National University of Defense Technology
Changsha 400073, China

***Abstract*—Current loop buffer has been mainly explored as an effective architectural technique for low-power execution in embedded processor. Another avenue, however, for exploiting loop buffer is to obtain its performance benefit. In this paper, we propose an application specific loop buffer organization for vectorized processing kernels, to achieve low-power and high-performance goals. The vectorized loop buffer (VLB) is simplified with single loop support for SIMD devices. Since significant data rearrangement overhead is required in order to use the SIMD capabilities, the VLB is specialized for zero-overhead implicit data permutation. We extend several instructions to the baseline ISA for programming and integrate it into an embedded processor for evaluation. Our results show that VLB improves the performance and power measures significantly compared to conventional SIMD devices.**

I. Introduction

High-performance and low-power are the two most important criteria for multimedia processor designs. Due to the limited power-supply capability of current battery technology, we are facing a dilemma that has two design problems to conquer, and they are considered to be problems of opposite natures. There have been many approaches to achieve these two design goals. One promising approach is to incorporate SIMD devices into the baseline processors that exploit data-level parallelism (DLP) in the form of SIMD instructions [1], [2]. It can leverage new opportunities for obtaining large performance gains at low-cost (low hardware occupation, low-power). SIMD devices are ubiquitous in embedded processors ranging from DSP [1] to gaming machines (CELL [2]). However, the SIMD architecture can be further optimized by loop buffering.

The loop buffering is usually regarded as an effective scheme to reduce energy consumption in the instruction memory hierarchy [3]. It stores a segment of loop codes in a small direct-mapped cache instead of a conventionally large instruction cache, which reduces the power consumption. However, its benefits, such as loop-back branch prediction, elimination of taken-branch bubbles, and fast instruction access time make it a good scheme for performance improvement.

In this paper, a vectorized loop buffer (VLB) for SIMD devices is proposed. It is specialized to reduce the overhead instructions that are required to assist the SIMD computation instructions by rearranging data and handling the loop branches and address generation [4], [5].

The first specialization of VLB is to employ implicit data permutation (IDP) mechanism into its organization via a special designed permutation vector register file (PVRF) [4]. It permits reducing the rearrange overhead of data array at the time of reading them. The second specialization is achieved by designing the simplified single loop buffer organization with flexible loop branch handling and vector memory address generation, which can further reduce the cost and power. We extend several instructions to the baseline ISA for VLB programming and integrate it into a embedded processor for evaluation. Our results show that VLB can significantly improves the performance and power measures of conventional SIMD devices.

II. Specialization of Loop buffer

Generally, the loop behavior of multimedia kernels can be classified into two different types: the static loop is usually fixed counted while the dynamic loop is usually variable counted or while type. In each loop body, the typical vector memory access patterns can be classified into four different types: constant access, sequential access, misaligned access, and strided access, as will be partly shown in Figure 2.

Consider the following simple code segment, assuming array a, b, and c are aligned.

```
for(int i=0; i<SIZE; i++)
  if(a[i] > b[i])
    c[i+1]=a[2i]+b[i+2];
```

To vectorize this loop, three techniques can be used: 1) if-conversion ($if(a[i] > b[i])$) [6], 2) misaligned handling ($c[i+1]$, $b[i+2]$) [7], 3) data reorganization ($a[2i]$) [4]. The vectorized implementation of this loop is shown in Code 1. The useful SIMD operations are shown in bold. All the other operations are regarded as overhead instructions. We can see that, 19 out of 24 instructions in the loop are overhead instructions.

Provided the loop characteristics existing in multimedia kernels, adopting loop buffer design into SIMD devices requires specializations in the following three aspects.

- *Specialization for data permutation handling*: Efficient implementation of data permutation in loop buffer to satisfy the SIMD processing is very important for performance improvement [5].
- *Specialization for address generation*: Simple and effective address generator is a key to the performance improvement of program.
- *Specialization for loop handling and programming*: The loop buffer should be more general to extend its capability

978-3-9810801-7-9/DATE11/©2011 EDAA

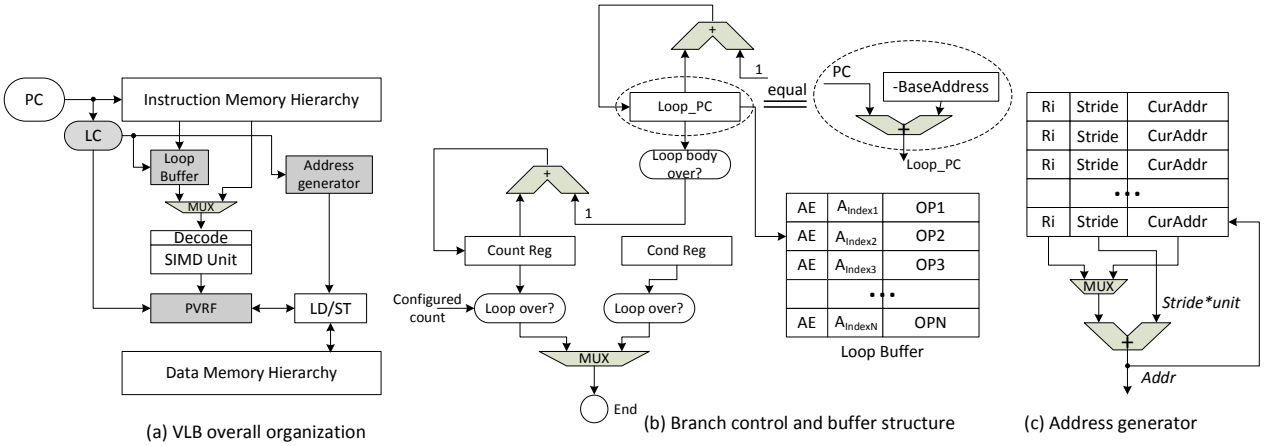

Fig. 1. Block diagram of VLB

Code 1 Example for loop overhead operations

```
1:   i=0;                          // Initialization
2:   V3 = vload(b);                // SIMD load of b[3:0]
3:   V5 = vload(c);                // SIMD load of c[3:0]
4:   V7 = vperm(V5, 0, p1);        // Pattern {V5[0],0,0,0}
5:     L0: V1 = vload(a);          // SIMD load of a[3:0]
6:     a=a+4;                      // Address calculation
7:     V2=vload(a);                // SIMD load of a[3:0]
8:     a=a+4;                      // Address calculation
9:     b=b+4;                      // Address calculation
10:    V4=vload(b);                // SIMD load of b[3:0]
11:    c4=c+4;                     // Address calculation
12:    V6=vload(c4);               // SIMD load of c[7:4]
13:    V10 = vcmp(V1, V3);         // SIMD compare
14:    V1 = vperm(V1, V2, p2);     // Pattern {V2[2],V2[0],V1[2],V1[0]}
15:    V3 = vperm(V3, V4, p3);     // Pattern {V4[1],V4[0],V3[3],V3[2]}
16:    V5 = vperm(V5, V6, p4);     // Pattern {V6[0],V5[3],V5[2],V5[1]}
17:    V6=V1+V3;                   // SIMD computation instructions
18:    V8 = select(V5,V6,V10);     // select from V5 and V6
19:    V7 = vperm(V7, V8, p5);     // Pattern {V8[2],V8[1],V8[0],V7[3]}
20:    c=vstore(V7);               // SIMD store of c[3:0]
21:    c=c+4;                      // Address calculation
22:    V1 = V2;                    // Value propagation for loop
23:    V3 = V4;                    // Value propagation for loop
24:    V5 = V6;                    // Value propagation for loop
25:    V7 = V8;                    // Value propagation for loop
26:    i=i+4;                      // Branch related overhead
27:    cmp(i, SIZE);               // Branch related overhead
28:    JS L0;                      // Branch Overhead
29:  V7 = vload(c);                // SIMD load of c[3:0]
30:  V7 = vperm(V7, V8, p6);       // Pattern {V7[3],V7[2],V7[3],V8[3]}
31:  c = vstore(V7);               // SIMD store of c[3:0]
```

to dynamic loops. The operations provided by loop buffer should not change existing ISA and facility the ease of use for the programmers.

III. VLB Approach

A. VLB Microarchitecture

Figure 1 shows the internal organization of a SIMD architecture enhanced with VLB. The modified part is presented in gray color as shown in Figure 1(a). It is designed to easily adapted to existing SIMD architectures. Four new hardware components are introduced, namely a loop controller (LC), a loop buffer, a address generator (AG) as well as PVRF.

Rather than proposing a more complicated power consuming loop buffer structure, this buffer is managed simply by hardware supporting only one loop at one time. The nested-loops are supported by transforming them into simple ones using techniques such as loop peeling, loop collapsing. Figure 1(b) illustrates its detailed structure. For static loop, branch related to loop increments (based on indices used for referencing data) is stopped by pre-defined count. For dynamic loop, the branch related to runtime behavior is stopped by testing certain condition register. The buffer has three fields: the *AE* field, which indicates the memory access instruction and its memory address is generated by AG unit, the A_{index} field, which represents the index of address information in the AG unit, and the *OP* field, which restores the instructions of the loop body.

The AG unit as illustrated in Figure 1(c) is used to automatically generate appropriate addresses that are required for the vector memory accesses in the loop with configured way. It has an address information table which contains three fields: R_i, the index of the register holding base address variable, $Stride$, the incremental value for successive loads or stores; and $CurAddr$, current address. Two methods are supported by this mechanism. The first static method is to add the value restored in $CurAddr$ by $stride \times unit$ after every memory access, where $unit$ is the bitwidth of SIMD unit. The second dynamic method is to add the value restored in register R_i by $stride \times unit$. Since the R_i could be changed at runtime, this method is suitable for irregular memory access. To avoid unnecessary setting, the default vector memory access pattern is sequential access. Though the proposed AG is simple, it is effective for handling the typical vector memory accesses as shown in Figure 2.

The specialization of VLB also incorporates an efficient IDP mechanism [4], which is implemented by two-stage permutation. In the first one, permutation pattern is set explicitly. Then in the second stage, elements from different vectors are packed together according to this pattern. As a permutation in each iteration usually has similar pattern, its pattern setting instruction can be moved out of the loop as invariant. So a lot of permutation instructions can be reduced leading to performance improvement. This is achieved by a permutation

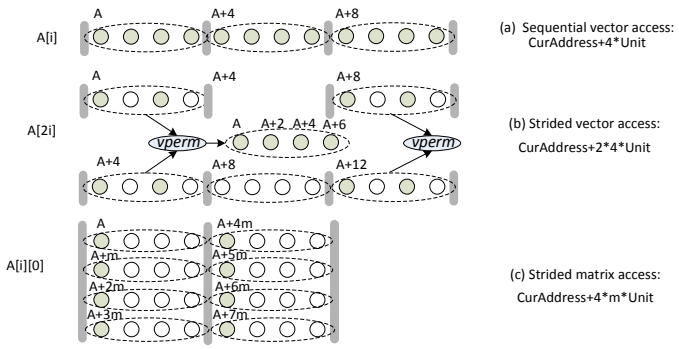

Fig. 2. Address generation for memory accesses

vector register file (PVRF) with its control logic integrated into loop controller, whose detailed description can be found in [4].

B. VLB Programming

Six new instructions are extended for the utilization of VLB. The *StartVLBMode(mode, CNT, R_i, VR)* instruction is used to set the internal execution states for VLB. It has four source parameters: *mode* specifies the execution mode encoded in 2-bit that indicates the loop condition. When '11', then the immediate number 16-bit *CNT* is used to guard the loop controller; when '10', then the value of R_i is used to guard the loop controller; Otherwise, false value of register is used to stop the loop execution, which is specified by the fourth parameter *VR*. The *EndLVBMode()* instruction is used to stop the VLB execution. The second type of extended instruction is used to configure memory patterns: *SetSS(offset, stride)* and *SetDS(offset, R_i, stride)*. There are three different parameters: *offset* specifies the offset of instruction in vector code, *stride* specifies the stride, and R_i is the register index for base address. The last type of extended instruction is related to IDP mechanism. The *SetVPerm(VR, mode)* and *ClearVPerm(VR)* instructions are used to set/cancel the internal permutation states of according to the information stored in register or encoded in instruction.

The main task of VLB programming is to determine the buffered loop body and its execution mode. The transformed vectorized code from motivating example is given in Code 2 by using VLB. The *SetVPerm* and *SetSS* instructions are added to configure the inner states of VLB before the loop. Since the pattern of accessing array *b* and *c* are sequential access, they do not require any configuration instruction. So the total number of overhead instructions in loop is decreased from 19 to 7. Thus the performance can be greatly improved.

IV. Evaluation

A. Methodology

In this section, we evaluate VLB in the context of a multimedia embedded processor, SDTA, which is based on the transport triggered architecture (TTA) [8]. The SDTA processor is utilized for accelerating multimedia kernels like data compression and graphics functions. It is augmented with

Code 2 VLB version for motivating example

```
1:  a2 = a+4;                          // base address for array a
2:  V3 = vload(b);                     // SIMD load of b[3:0]
3:  V5 = vload(c);                     // SIMD load of c[3:0]
4:  V7 = vperm(V5,0, p1);              // pattern {V5[0],0,0,0}
5:  SetVPerm(V1, p2);                  // pattern {V2[2],V2[0],V1[2],V1[0]}
6:  SetVPerm(V3, p3);                  // pattern {V4[1],V4[0],V3[3],V3[2]}
7:  SetVPerm(V5, p4);                  // pattern {V6[0],V5[3],V5[2],V5[1]}
8:  SetVPerm(V7, p5);                  // pattern {V8[2],V8[1],V8[0],V7[3]}
9:  SetSS(0, 8);                       // Set stride pattern
10: SetSS(1, 8);                       // Set stride pattern
11: StartVLBMode(1, SIZE/4, 0);        // The loop start indicator
12:   V1=vload(a);                     // SIMD load of a[3:0]
13:   V2=vload(a2);                    // SIMD load of a2[3:0]
14:   V4=vload(b);                     // SIMD load of b[3:0]
15:   V6=vload(c4);                    // SIMD load of c[7:4]
16:   V10 = vcmp(V1, V3);              //SIMD compare
17:   V6=V1+V3;                        // SIMD computation instructions
18:   V8 = select(V5,V6,V10);          // select from V5 and V6
19:   c=vstore(V7);                    // SIMD store of c[3:0]
20:   V1 = V2;                         // Value propagation for loop
21:   V3 = V4;                         // Value propagation for loop
22:   V5 = V6;                         // Value propagation for loop
23:   V7 = V8;                         // Value propagation for loop
24: EndLVBMode();                      // The loop end indicator
25: V7 = vload(c);                     // SIMD load of c[3:0]
26: V7 = vperm(V7, V8, p6);            // Pattern {V7[3],V7[2],V7[3],V8[3]}
27: c = vstore(V7);                    // SIMD store of c[3:0]
```

TABLE I
VLB implementation results

Resources	Area (mm^2)	Delay (ns)
Address Generator	0.09	0.71
Loop controller	0.05	0.42
Instruction Buffer	0.19	0.55
PVRF	0.35	0.52
Overall	0.68	-

128-bit wide SIMD devices and includes all the features of current SIMD devices such as general *vperm* instruction and conditional instruction for if-conversion.

An experimental framework was built on the TTA compiler tool chain. All the SIMD versions are programmed directly using C intrinsics. The tested kernels used in this study are selected from BMKL workload [9], a set of computationally important kernel functions in multimedia applications.

B. Experimental results

1) hardware implementation: Table I shows the area and delay for the VLB hardware under 90nm TSMC technology. The hardware resources consist of address generation, loop circuitry, instruction buffer and PVRF. We can see that the overall area required for implementing these hardware facilities is $0.68mm^2$, which is relatively inexpensive to a CPU core. As the introduced delay of VLB can be overlapped with other processor pipeline stages such as memory access, we do not consider these delays as drawback of performance since they would not change the processor timing.

2) Performance results: Figure 3 displays the obtained speedups for the different benchmarks. Three code versions are provided: scalar, conventional SIMD and proposed VLB. All results show VLB architecture has a significant speedup over conventional SIMD which ranges from 1.22 to 1.56. Compared to scalar codes, most of these benchmarks saw a speedup of two to five. This is due to the specialization

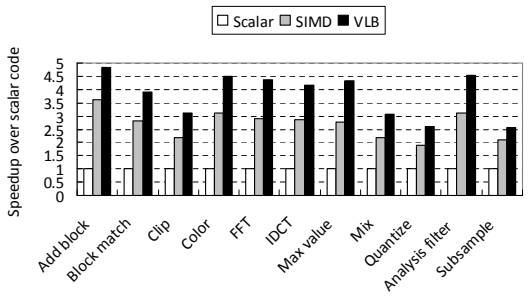

Fig. 3. Speedup comparison over scalar code

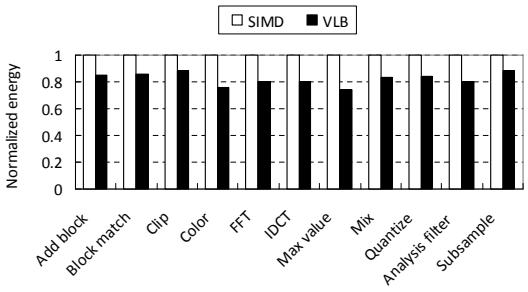

Fig. 4. Energy reduction of all benchmarks

of VLB architecture, which can overlap different iterations of one loop body and execute *SIZE-1* copies of the loop automatically, so *SIZE-1* copies of the loop handling, memory address generation and data permutation overhead for SIMD processing are eliminated, and only the start and the exist instructions of the loop body are retained.

3) Power results: The power consumption was measured by applying kernels for different SDTA scenarios (with and without VLB) and estimated by using PrimePower tool with the generated transistor netlist and Modelsim simulator value change file (VCD file). Figure 4 shows the energy reduction obtained by augmenting VLB into SDTA processor. On average, 18% of energy can be reduced. This energy reduction can be explained as follows: First, through the localization of instruction fetch, the effective energy per access can be reduced. Second, overhead instructions for loop handling and vector generation are eleminated, which can save most of power consumption for the execution of these instructions. Third, the IDP mechanism in VLB can also reduce the power consumption. This is because the power consumption of an implicit data permutation operation is far less than the that of an explicit data permutation instruction execution.

V. Related works

Several loop buffering schemes have been proposed in the past [3], [10], [11], [12]. In [10], a small filter cache was inserted before the L1 cache. Power is reduced by filtering cache references through this very small filter cache. In [11], compiler techniques were proposed to select segments of codes to be placed in the filter cache so that the miss ratio in the filter cache is reduced. Another approach [3], [12] proposed an architecture that has a small buffer located along side the L1 cache. The basic idea is to fetch instructions once at an early iteration of the loop and store them in a small buffer. The design of VLB also use this kind of loop buffer.

In this paper, we apply loop buffer into SIMD devices. Different from previous work such as loop units in some DSP and MediaBreeze processor [5], which does not consider the characteristics of multimedia kernels, we specialize the loop buffer to effectively handling the special memory access and data rearrange existing in multimedia kernels resulting in both low-power and high-performance.

VI. Conclusion

This paper presents VLB architecture for SIMD devices. The proposed VLB organization is specialized to multimedia kernels with zero overhead IDP mechanism, flexible loop branch handling and vector memory address generation. We integrate it into a multimedia embedded processor for evaluation. The experimental results has shown its benefit compared to conventional SIMD devices without it. For the future, it might be interesting to study more specialization techniques for executing multimedia kernels in VLB architecture.

VII. Acknowledgements

We gratefully acknowledge the supports of the National Basic Research Program of China under the project number 2007CB310901 and the National Natural Science Foundation of China under Grant No. 61025009, No. 60803041, and No. 60773024.

References

[1] D. Naishlos, M. Biberstein, S. Ben-David, and A. Zaks, "Vectorizing for a simdd dsp architecture," in *CASES'03*, pp. 2-11, 2003.

[2] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cells multicore architecture," in *IEEE Micro*, vol. 26, no. 2, pp. 10-24, 2006.

[3] Anderson, T., Agarwala, S., "Effective hardware-based two-way loop cache for high performance low power processors," in *ICCD'00*, p. 403-407, 2000.

[4] H. Libo, S. Li, W. Zhiying, S. Wei, X. Nong, and M. Sheng, "Sif: Overcoming the limitations of simd devices via implicit permutation," in *HPCA 16*, pp. 355-366, 2010.

[5] D. Talla and L. K. John, "Cost-effective hardware acceleration of multimedia applications," in *ICCD'01*, pp. 415-424, 2001.

[6] J. E. Smith, G. Faanes, and R. Sugumar, Vector instruction set support for conditional operations, in *ISCA'00*, pp. 260-269, 2000.

[7] A. E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for simd architectures with alignment constraints," in *SIGPLAN Not.*, vol. 39, no. 6, pp. 82-93, 2004.

[8] V. A. Zivkovic, R. J. W. Tangelder, and H. G. Kerkhoff, "Design and test space exploration of transport-triggered architectures," in *DATE'00*, pp. 146-153, 2000.

[9] N. Slingerland and A. J. Smith, "Measuring the performance of multimedia instruction sets," *IEEE Trans. Comput.*, vol. 51, no. 11, pp. 1317-1332, 2002.

[10] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *MICRO 30*, pp. 184-193, 1997.

[11] N. B. I. Hajj, G. Stamoulis, N. Bellas, and C. Polychronopoulos, "Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors," in *ISLPED'98*, pp. 70-75, 1998

[12] R. S. Bajwa, M. Hiraki, H. Kojima, D. J. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki, "Instruction buffering to reduce power in processors for signal processing," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 5, no. 4, pp. 417-424, 1997.