Software Programmable Data Allocation in Multi-Bank Memory of SIMD Processors

Jian Wang, Joar Sohl, Olof Kraigher and Dake Liu

Department of Electrical Engineering

Linköping University

Linköping, Sweden

Email: {jianw, joar, kraigher, dake}@isy.liu.se

Abstract—The host-SIMD style heterogeneous multi-processor architecture offers high computing performance and user-friendly programmability. It explores both task level parallelism and data level parallelism by the on-chip multiple SIMD coprocessors. For embedded DSP applications with predictable computing feature, this architecture can be further optimized for performance, implementation cost and power consumption. The optimization could be done by improving the SIMD processing efficiency and reducing redundant memory accesses and data shuffle operations. This paper introduces one effective approach by designing a software programmable multi-bank memory system for SIMD processors. Both the hardware architecture and software programming model are described in this paper, with an implementation example of the BLAS *syrk* routine. The proposed memory system offers high SIMD data access flexibility by using lookup table based address generators, and applying data permutations on both DMA controller interface and SIMD data access. The evaluation results show that the SIMD processor with this memory system can achieve high execution efficiency, with only 10% to 30% overhead. The proposed memory system also saves the implementation cost on SIMD local registers; in our system, each SIMD core has only 8 128-bit vector registers.

Index Terms—SIMD processor; multi-bank memory; conflict-free memory access; data allocation

I. INTRODUCTION

Contemporary digital signal processing (DSP) applications in embedded systems demand a large computing power, especially in real-time systems. These applications are featured as data rich and compute-intensive, involving video compression, graphics computing, speech processing and communications. To meet the high computing requirements, most of current embedded systems use the multi-core System-on-Chip (SoC) solution, which consists of a host controller, such as an ARM9, PowerPC or MIPS, and multiple data processing IP cores designed for specific DSP algorithms. Typical examples of this architecture are the TI OMAP platform[1] and the STi7100 HDTV set-top-box SoC[2] from ST Microelectronics. The multi-core SoC approach provides the right level of performance and power dissipation for a specific application; however, it has limits in flexibility and application domain. For the emerging applications with larger data volume and more complex computing, like Full HD (1920×1080) video codec and multi-antenna MIMO technology in wireless communication, the existing SoC platforms could not provide enough computing power and cannot be used for those new

applications. Another category of solutions is to use heterogeneous multi-processor to offer high computing power in a broader application domain. This architecture implements a host controller with a set of programmable coprocessors. The host controller targets program control and the coprocessors target data processing. A well-known example is the IBM, Sony and Toshiba Cell broadband engine[3]. The Cell processor is based on a PowerPC processor (PPE) connected with eight parallel processing elements (SPE). One advantage of the heterogeneous multi-processor architecture is that the coprocessors are designed to be programmable to support different computing algorithms. To further improve data processing performance for computing intensive applications, most coprocessors employ the single instruction multiple data (SIMD) architecture, including the SPE coprocessor in Cell. The SIMD architecture is very efficient in exploring data level parallelism since it executes multiple data operations concurrently by using a single instruction. However, the performance of SIMD architecture can be significantly affected by irregular data access, for example, the not aligned data access where the start address of the data is not at the alignment boundary. For irregular data accesses, the processor must issue more than one memory access to prepare one group of data for a single SIMD arithmetic instruction, which will decrease the performance. It is even worse when more complex data access is required and data shuffling is needed. A previous application profiling [4] has shown that for computing algorithms in a MPEG2 decoder, on average only 30% of the total SIMD memory accesses are regular. Thus a memory subsystem that minimizes the irregular data access overhead is essential for enhancing the SIMD processor performance.

The current solutions of memory subsystems for SIMD processors include single port memory, multi-port register file, and multi-bank memory. The single port memory is simple to implement but only supports aligned access. The multi-port register file offers good flexibility for irregular data accesses, e.g. the eLite SIMD architecture[5], but the hardware cost and power consumption are high, especially for large memory size. Multi-bank memory is widely used as a trade-off between implementation cost and data access flexibility. There have been much research to reduce the SIMD data access overhead on multi-bank memory. A Compiler based dynamic memory allocation on a multi-bank memory system is proposed[4]. But,

since SIMD compiler itself can not show enough performance at auto vectorization of high level languages such as C, current optimization of SIMD programs is still by using intrinsics and assembly. Another approach is by using cross-bar or customized cross-bar[6] for data-shuffling. This hardware solution offers more data addressing flexibility. The drawbacks are the high implementation cost and power consumption. The customized crossbar reduces the hardware complexity, but it also limits the application domain.

Investigating the signal processing applications in modern embedded systems, we can find that a large part of the computing algorithms have the characteristics of data rich, computing intensive, and data access predictable. This means the kernel DSP programs are usually with little control, large volume data and complex arithmetic operations. The predictable computing indicates that the data access patterns are known at design time. Even though, a great design challenge exists because of the irregular access patterns in various applications. And even more, many algorithms require multiple access patterns on the same data block. This increases the flexibility requirement of the memory system for SIMD architecture. The ePUMA [7] (embedded Parallel DSP architecture with Unique Memory Access) project at Linköping University aims at developing a heterogeneous multi-processor DSP targeting predictable embedded signal processing. The SIMD coprocessors in ePUMA use a novel software programmable local memory system to improve the SIMD processing efficiency. This memory system implements the P3RMA (Predictable Parallel Memory Architecture for Predictable Random Memory Access) technology introduced in [8]. More application references can be found in [9].

The remainder of this paper is organized as follows. It starts from an introduction of the ePUMA parallel DSP platform (Section II), then describes the SIMD local memory system architecture (Section III) and the software programming model (Section IV). In Section V we implement the BLAS routine *syrk* as an example to demonstrate the flexible vector access supported by ePUMA memory subsystem. Some evaluation results are provided in Section VI to prove the SIMD execution efficiency. Finally a conclusion of this paper is made in Section VII.

II. AN OVERVIEW OF EPUMA

The ePUMA project aims at designing a sub-1 watt parallel DSP for high performance embedded signal processing. As shown in Figure 1, the ePUMA platform uses a heterogeneous chip multiprocessing architecture consisting of one host controller and eight SIMD coprocessors. It decouples program control and data processing to maximize the computing performance. The host controller is a RISC processor, it controls the coprocessors and the memory subsystem to execute independent compute and transfer data. Each SIMD coprocessor has an 8-way parallel data path. The combined eight SIMD processors deliver the majority of the system's computing power. The ePUMA memory subsystem, including the DMA controller, the on-chip interconnection and the SIMD local

Fig. 1. ePUMA chip multi-processor architecture

memory system, is designed to maximally reduce the data access overhead for predictable embedded DSP applications. The essential technologies include the enhanced DMA controller and the software programmable SIMD local memory system. This paper focuses on the SIMD local memory system design.

III. SIMD LOCAL MEMORY ARCHITECTURE

The conventional SIMD extension architecture has very simple data buffer for SIMD computing. It shares the existing registers in the host processor, for example Intel MMX reuses the floating point registers of the CPU, or uses small and dedicated wide registers for vector data, like in the Intel SSE and the ARM NEON media acceleration. This architecture tightly couples the SIMD processing unit to the host processor and uses a single instruction stream. The vector data load/store operations on the SIMD local buffers are handled by the host processor.

As SIMD architecture is extended to the chip multi-processor platform as a vector coprocessor, it first separates the instruction flow from the host program. SIMD coprocessor runs its own software, which is called kernel program. On the host-multi-SIMD platforms like IBM Cell, different SIMD coprocessors can concurrently execute different kernel programs to achieve a task level parallelism. Another change is on the SIMD local memory. The SIMD coprocessor is associated with a larger local data memory. The input data are preloaded to this local memory through a Direct Memory Access (DMA) controller before the kernel execution. SIMD core accesses its local memory by using the vector load and store instructions. After the execution of kernel program, the result data are moved out from the local memory by another DMA transaction.

ePUMA SIMD coprocessor's local memory architecture is illustrated in Figure 2. It uses a multi-bank memory consisting of eight 16-bit single port SRAM modules to provide 8×16 -bit parallel access from both the 128-bit wide interconnection bus and the 8-way SIMD core. In order to support more flexible vector addressing, we implement data permutation logic on both sides of the multi-bank memory. One permutation is used for the DMA transfer and another permutation for SIMD data access. Two software programmable vector address generation unites (Vector AGU or VAGU) control the data permutation

Fig. 2. SIMD local memory subsystem

blocks, as shown by VAGU-1 and VAGU-2 in the figure. VAGU-1 is programmed by the host processor, and VAGU-2 receives control signals from SIMD core.

A. Multi-bank Memory Address Mapping

In ePUMA system, each multi-bank memory has a unified and linear local address space exposed to software developers to control the data access. Figure 3 shows the address map of an 8-bank memory. The VAGU can be programmed to generate a conflict free address group for parallel access of the multi-bank memory. The programming of VAGU is described in Section IV. The address pattern can be very flexible, as the example shown in Figure 3, where a vector of data (shown in grey blocks) coming from eight memory banks is accessed in parallel and shuffled automatically by using the VAGU to generate a conflict free address vector.

B. Vector Address Generator

The hardware implementations of the address generation logic in VAGU-1 and VAGU-2 have the same architecture, as illustrated in Figure 4. It consists of two branches; the scalar branch (SC) calculates a base address, while the vector branch (PT) generates an offset vector. The offset vector is calculated based on a lookup table (LUT). It is added to the base address to get the final address for vector access. The LUT is stored in a small constant memory. A decoder unit is implemented to

Fig. 3. Multi-bank memory address mapping

Fig. 4. SIMD vector address generation unit

TABLE I
VAGU-1 CONFIGURATION PARAMETERS

Config. reg.	Function
sc.init	SC initial value
sc.top	upper bound for modular addressing
sc.bottom	lower bound for modular addressing
sc.stride	step size for next SC address
sc.incr	SC increment control
pt.val_sel	PT vector select, [0..7] or LUT
pt.init	LUT index initial value
pt.top	LUT index upper bound
pt.bottom	LUT index lower bound
pt.incr	LUT index increment control

decode the vector address into each memory bank's physical address and output control signals to the data shuffle block.

The DMA controller has the same data width as the local vector memory (LVM). It can transfer one vector per cycle. VAGU-1 is used to generate addresses for DMA block data transfer. It is pre-configured before a DMA transaction. After DMA starts, VAGU-1 pops one address for each vector access and update itself automatically. The configuration parameters of VAGU-1 are listed in Table I. We give a VAGU-1 configuration example for an unaligned stride-based access in Listing 1. The address sequence generated from this configuration is shown in Figure 5. In this example, the scalar branch generates a base address starting from SC initial value 2 and increases by a stride of 16 for the next address. The sc.top and sc.bottom registers are programmed to set the boundaries for base address update, when it achieves the upper limit 100, it returns back to the lower bound 2. The register pt.val is set to 0 to disable the table look-up. The vector branch will always output a constant offset from 0 to 7 in this case.

Listing 1. Configure an unaligned stride-based access

```

sc.init = 2;
sc.top = 100;
sc.bottom = 2;
sc.stride = 16;
sc.incr = 1;
pt.val_sel = 0;

```

The second vector address generator VAGU-2 is tightly coupled to the SIMD processing core. It offers the SIMD core with same access flexibility as the DMA controller. The control of VAGU-2 is through the address registers in the SIMD processor. The software programmers can program a specific

Vector address 0:	2	3	4	5	6	7	8	9
1:	18	19	20	21	22	23	24	25
2:	34	35	36	37	38	39	40	41
:	:	:	:	:	:	:	:	:

Fig. 5. VAGU-1 output sequence for unaligned stride-based access

vector access pattern for each data load/store operation.

Combining VAGU-1 and VAGU-2, ePUMA software can explicitly control the data allocation, the conflict free vector access, and the data shuffling in the multi-bank memory. The SIMD coprocessor doesn't need to do complex data packing/unpacking and shuffle operations on register files. This saves the implementation cost of SIMD local registers. In our design, each ePUMA SIMD coprocessor has only 8 128-bit vector registers.

IV. PROGRAMMING MODEL

This section introduces the ePUMA programming model supporting the novel memory subsystem. As a chip-multiprocessor platform, ePUMA has a separate host program and SIMD coprocessor program. The host program is written in C with certain extension functions. These extensions are for coprocessor control and DMA data transactions. The SIMD coprocessors are programmed in assembly. The program size is usually small. A typical SIMD kernel program contains tens of SIMD instructions, as shown in Table II.

TABLE II
SIMD KERNEL PROGRAM SIZE

Algorithm	Instruction count
Complex matrix mul.	14
8-tap real FIR filter	10
48-tap complex FIR filter	27
Radix-2 FFT transform	70
Radix-4 FFT transform	56

We decide to program the SIMD coprocessor in assembly for the following two reasons. The first is to get the greatest performance of the SIMD core by programming in low level. Secondly, today's compiler techniques can not show satisfactory performance for software vectorization. This is especially true for our ePUMA SIMD processor with more powerful and complex vector operations than conventional SIMD architectures.

A. Host Program with C Extensions

The host program uses standard C with custom extension functions. DMA function is one of these extensions. It handles data transfer between off-chip memory and SIMD local memories. The configuration of VAGU-1 is included in the DMA function, as shown in Listing 2. The software programs the VAGU-1 by setting the control registers and the LUT table contents directly.

Listing 2. VAGU-1 configuration in host program

```
/* Define a DMA task */
DMATask T1{
```

```
    TYPE = MMLVM;
    MODE = BROADCAST;
    SIMD_SEL = 0x000F;
    SIZE = 128;
    VAGU { // SIMD0 VAGU-1 configuration
        SIMD_ID = 0;
        SC = {
            INIT = 0;
            TOP = 512;
            BOT = 0;
            STRIDE = 8;
            INCR = 1;
        };
        PT = {
            VAL_SEL = 1;
            INIT = 0;
            TOP = 5;
            BOT = 0;
            INCR = 1;
        };
        LUT = {
            7,6,5,4,3,2,1,0;
            0,9,18,27,36,45,54,63;
            ...
        };
    };
};
```

```
/* Configure DMA controller */
int dma_hdl;
dma_hdl = ConfigDMATask(T1);
```

```
/* Start DMA transaction */
StartDMATask(dma_hdl);
```

```
/* Some other tasks */
...
```

```
/* Wait for DMA finish */
WaitOnDMAFinish(dma_hdl);
```

B. SIMD Data Access Instructions

ePUMA SIMD coprocessor implements an instruction set with unique memory access. It is different from conventional RISC processors or application specific DSP processors, which support a group of fixed access patterns, such as stride-based, modular, and bit-inverse addressing. ePUMA SIMD has a software programmable data access based on lookup table. It can easily cover all the previous addressing patterns. Furthermore, the software programmers can use the LUT to develop specific addressing patterns for various algorithms. The example below shows an ePUMA SIMD instruction doing data copy from local memory m0 to vector register vr0.

```
vcopy vr0 m0[ar0+=8% + cm[car0+=1%]].vw
```

The access pattern of m0 is specified in the instruction, which has two parts, the *ar0+=8%* controls the scalar branch in the vector AGU, and the *cm[car0+=1%]* configures the vector branch. The *cm* in the instruction refers to a constant memory which stores the permutation table. The table contents are defined in the kernel program using an assembly intrinsic, and are loaded to SIMD coprocessor's constant memory during program loading.

Fig. 6. Data allocation in main memory for the *syrk* example

V. AN IMPLEMENTATION EXAMPLE OF THE BLAS *syrk* ROUTINE

The Basic Linear Algebra Subprograms (BLAS) standard [10], [11] defines a software interface to perform basic linear algebra operations such as vector and matrix multiplication. As a fundamental component of the core math functions, BLAS is heavily used in high performance computing. Most of today's hardware vendors have developed optimized implementations of the BLAS operations, for example, Intel Math Kernel Library (MKL), AMD Core Math Library (ACML), and Nvidia CUBLAS library.

In this section, we implement a BLAS level-3 routine *syrk* as an example to show the memory access flexibility offered by ePUMA memory subsystem. The *syrk* function performs a rank- k update of a symmetric matrix. Its operation, for the case $trans = T$, is defined in the equation below.

$$C = \alpha * A^T * A + \beta * C, \text{ for } trans = 'T' \quad (1)$$

C is an n -by- n symmetric matrix, A is a k -by- n matrix, α and β are scalar coefficients.

We choose $n = 4$ and $k = 8$ for demonstration. In this small size example, we assume that a complete matrix C is stored in the system main memory in a sequential order. Matrix A is also stored sequentially in the main memory, as shown in Figure 6.

One SIMD coprocessor is used for *syrk* kernel computing. Before the execution, host processor will issue two DMA transactions to transfer matrix A and C to the SIMD's local memory. The DMA controller has the same data width as the local multi-bank memory, so eight data can be transferred per cycle. Because of the limit of the memory controller interface and the data allocation in off-chip memory, the eight data transfer in one burst should come from continuous locations in the main memory.

The computing $A^T * A$ requires a column-wise access of matrix A , as highlighted by the grey blocks in Figure 6. The SIMD coprocessor loads column vectors from A and then computes vector-vector multiplication with one SIMD instruction. To maximally reduce the vector load overhead, the column vector should be allocated in different memory

Fig. 7. Data allocation in SIMD local memory for the *syrk* example

banks during DMA transaction. The VAGU-1 is programmed by software to achieve this. The configuration of VAGU-1 is illustrated in Listing 3. Matrix C has normal row-wise access so its VAGU-1 is programmed for simple sequential address pattern. The data allocation in the SIMD local memory is illustrated in Figure 7. It can be seen that all the column vectors of matrix A are allocated in different banks. The SIMD core can access any one column vector by one SIMD load/store instruction without bank conflict.

Listing 3. VAGU-1 configuration in DMA function

```

/* VAGU configuration for DMA matrix A */
VAGU {
    SC = {
        INIT = 16; // base address
        INCR = 0; // no increase
    };
    PT = {
        VAL_SEL = 1; // enable the LUT
        INIT = 0;
        TOP = 3;
        BOT = 0;
        INCR = 1;
    };
    LUT = {
        0,1,2,3,4,5,6,7;
        9,10,11,12,13,14,15,8;
        18,19,20,21,22,23,16,17;
        27,28,29,30,31,24,25,26;
    };
};

```

The matrix data are allocated in SIMD local memory for conflict free parallel access by DMA transfer. The SIMD processor needs to be programmed to support the access of the re-ordered memory data. The assembly code listed below illustrates the column vector access in the *syrk* SIMD program. Two *vcopy* instructions load the first two columns of matrix A to vector register *vr0* and *vr1*, using a permutation table defined after the intrinsic *.cm*. The SIMD computing is done by the *vmac* instruction for vector-vector multiplication and accumulation.

Listing 4. SIMD program for the *syrk* example

```

.code
set ar0 $16

```

```

set car0 pt
;; load matrix A using permutation table
vcopy vr0 m0[ar0 + cm[car0+=1]].vw ;
vcopy vr1 m0[ar0 + cm[car0+=1]].vw ;
...

.cm
pt:
0 4 9 13 18 22 27 31
1 5 10 14 19 23 28 24
2 6 11 15 20 16 29 25
3 7 12 8 21 17 30 26

```

The above example shows how the ePUMA SIMD local memory system supports software programmable data allocation. The SIMD data access overhead is minimized by allocating column vector data in different memory banks. For larger size and more complex applications, ePUMA platform's advantages will become more significant. Some selected evaluation results are presented in the following section.

VI. EVALUATION RESULTS

The current evaluation programs are executed on the ePUMA cycle-true and pipeline-accurate simulator. We have developed a fully functional simulator including components as the host processor, SIMD coprocessor, on-chip interconnection and memory subsystem. The ePUMA SIMD coprocessor with the novel memory subsystem is evaluated using the ratio R as a measurement of efficiency, where

$$R = \frac{\text{Total execution cycles}}{\text{Arithmetic instructions}} \quad (2)$$

Table III lists the evaluation results of five selected DSP kernels. In the table, the theoretical limit shows the number of SIMD operations (8-way) required for the kernel computing. The SIMD processor is capable of executing one such operation per cycle. So this number equals to the arithmetic instructions in Equation 2. The total cycle count lists the kernel execution time in simulator. In addition to the computing instructions, this cycle count also includes SIMD execution overheads as data load/store instructions, branch latency, pipeline synchronization slots, etc. The ratio R in the table shows that the SIMD processing overhead is about 10% to 30%, which implies a high processing efficiency. In our previous publication [12], we have evaluated the ratio R in a non-optimized 8-way SIMD processor architecture, where the multi-bank memory only supports sequential addressing as in most existing SIMD processors and the programmer has to use data permutation instructions to shuffle vector data for SIMD computing. The ratio R is 7.47 for 64×64 matrix multiplication in the non-optimized architecture because of the execution overhead on data shuffling.

There exists potential complexity overhead on ePUMA software programmable multi-bank memory system. The permutation table size varies for different applications. Loading permutation table consumes DMA time. However, all current implemented benchmarks require very small size tables. We believe that a careful kernel design will not need large size permutation table.

TABLE III
ePUMA EVALUATION RESULTS

Algorithm	Theoretical limit	Total cycles	R
64×64 complex matrix mul.	65536	73737	1.13
1k FFT	1280	1710	1.33
4k FFT	6144	7635	1.24
8k 8-tap real FIR	8192	8212	1.00
2k 48-tap complex FIR	26679	24576	1.09

VII. CONCLUSION

In this paper we present a software programmable SIMD local memory subsystem in the ePUMA parallel DSP platform, both the hardware architecture and the programming model are described. The proposed solution has low implementation cost, low power consumption, and high data access flexibility. It enables the software programmer to explicitly control the data allocation in multi-bank memory and access vector for SIMD computing in a conflict free manner. The ePUMA architecture is optimized for predictable embedded signal processing. The evaluation results show that the ePUMA memory subsystem can effectively reduce the SIMD data access overhead and achieve a high computing efficiency. Many projects in cooperation with other algorithm groups at the university are in progress to implement more applications on this platform.

ACKNOWLEDGEMENT

The authors would like to thank SSF, Swedish Foundation for Strategic Research, for the support of this project.

REFERENCES

- [1] *Wireless Terminals Solution Guide*, Texas Instruments, 2007.
- [2] *Low cost HDTV set-top box decoder for H.264/AVC and MPEG-2*, ST Microelectronics, 12 2006.
- [3] M. Gschwind, "The Cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor," *Int. J. Parallel Program.*, vol. 35, no. 3, pp. 233–262, 2007.
- [4] H. Chang, J. Cho, and W. Sung, "Compiler-based performance evaluation of an SIMD processor with a multi-bank memory unit," *Signal Processing Systems*, vol. 56, no. 2-3, pp. 249–260, 2009.
- [5] D. Naishlos, M. Biberstein, S. Ben-David, and A. Zaks, "Vectorizing for a SIMD DSP architecture," in *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. New York, NY, USA: ACM, 2003, pp. 2–11.
- [6] P. Raghavan, S. Munaga, E. R. Ramos, A. Lambrechts, M. Jayapala, F. Catthoor, and D. Verkest, "A customized cross-bar for data-shuffling in domain-specific SIMD processors," in *ARCS*, ser. Lecture Notes in Computer Science, vol. 4415. Springer, 2007, pp. 57–68.
- [7] "ePUMA: embedded parallel DSP architecture with unique memory access," <http://www.da.isy.liu.se/research/scratchpad/>.
- [8] D. Liu, *Embedded DSP Processor Design, Volume 2: Application Specific Instruction Set Processors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008, ch. 20.
- [9] M. Gossel, B. Rebel, and R. Creutzburg, *Memory Architecture and Parallel Access*. New York, NY, USA: Elsevier Science Inc., 1994.
- [10] "BLAS (Basic Linear Algebra Subprograms)," <http://www.netlib.org/blas/>.
- [11] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.
- [12] J. Sohl, J. Wang, and D. Liu, "Large matrix multiplication on a novel heterogeneous parallel DSP architecture," in *APPT*, 2009, pp. 408–419.