SAMS Multi-Layout Memory: Providing Multiple Views of Data to Boost SIMD Performance

Chunyang Gou, Georgi Kuzmanov, Georgi N. Gaydadjiev
Computer Engineering Lab
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, The Netherlands
{C.Gou, G.K.Kuzmanov, G.N.Gaydadjiev}@tudelft.nl

ABSTRACT

We propose to bridge the discrepancy between data representations in memory and those favored by the SIMD processor by *customizing the low-level address mapping*. To achieve this, we employ the extended Single-Affiliation Multiple-Stride (SAMS) parallel memory scheme at an appropriate level in the memory hierarchy. This level of memory provides both Array of Structures (AoS) and Structure of Arrays (SoA) views for the structured data to the processor, appearing to have maintained *multiple layouts* for the *same data*. With such multi-layout memory, optimal SIMDization can be achieved. Our synthesis results using TSMC 90nm CMOS technology indicate that the SAMS Multi-Layout Memory system has efficient hardware implementation, with a critical path delay of less than 1ns and moderate hardware overhead. Experimental evaluation based on a modified IBM Cell processor model suggests that our approach is able to decrease the dynamic instruction count by up to 49% for a selection of real applications and kernels. Under the same conditions, the total execution time can be reduced by up to 37%.

Categories and Subject Descriptors: C.1.2[Multiple Data Stream Architectures (Multiprocessors)]:SIMD; B.3.2[Design Styles]:Interleaved memories
General Terms: Design, Performance

1. INTRODUCTION

One of the most critical challenges in SIMD processing is imposed by the data representation. By exploiting explicitly expressed data parallelism, SIMD processors tend to provide higher performance for computationally intensive applications with lower control overhead compared to superscalar microprocessors. However, SIMDization suffers from the notorious problems of difficult data alignment and arrangement, which greatly undermine its potential performance[4, 26, 31].

In both scientific and commercial applications, data is usually organized in a structured way. A sequence of structured data units could be represented either in AoS (Array of Structures) or in SoA (Structure of Arrays) format. Such data representation predetermines, at the application level, the data layout and its continuity in the *linear memory address space*. It has been found that most SIMDized applications are in favor of operating on SoA format for better performance[18, 20]. However, data representation in the system memory is mostly in the form of AoS because of two reasons. First, AoS is the natural data representation in many scientific and engineering applications. Secondly, indirections to structured data, such as pointer or indexed array accesses, are in favor of the AoS format. Therefore, a pragmatic problem in the SIMDization arises: the need for dynamic data format transform between AoS and SoA, which results in significant performance degradation. To our best knowledge, no trivial solution for this problem has been previously proposed. Our SAMS Multi-Layout Memory system, presented in this paper, supports contiguous data access for both AoS and SoA formats. The specific contributions of our proposal are:

- Custom, low-level address mapping logic to manage individual internal data layout and provide efficient memory accesses for both AoS and SoA views;
- Novel hardware/software interface for improved programmer productivity and additional performance gains;
- The SAMS scheme implementation in TSMC 90nm CMOS technology with affordable critical path (<1ns) and its integration into the IBM Cell SPE model;
- Up to 49% improvement in dynamic instruction counts for real applications and kernels, which is translated into a 37% reduction of the overall execution time.

The remainder of the paper is organized as follows. In Section 2, we provide the motivation for this work. In Section 3, the original SAMS scheme and the proposed extensions are briefly described. The hardware implementation and synthesis results of the SAMS Multi-Layout Memory system and its integration to the IBM Cell SPE are presented in Section 4. Simulated performance of the SAMS memory in applications and kernels is evaluated in Section 5. The major differences between our proposal and related art are described in Section 6. Finally, Section 7 concludes the paper.

2. MOTIVATION

Motivation Example: We shall examine the SIMDization of vector-matrix multiplication, $\mathbf{c} = \mathbf{a} * \mathbf{B}$, where $\mathbf{a}$ and $\mathbf{c}$ are 1x3 vectors and $\mathbf{B}$ is a 3x3 matrix with column-major storage. Although the involved computations are quite simple, SIMDizing them to achieve optimal speedup is very difficult. Assuming

ICS'10, June 2–4, 2010, Tsukuba, Ibaraki, Japan.

```
struct vec3 {float a0[4], float a1[4],
float a2[4]};
struct matrix3 {float b0[4], float
b3[4], float b6[4], float b1[4], float
b4[4], float b7[4], float b2[4], float
b5[4], float b8[4]};
…
struct vec3 a, c;
struct matrix3 b;
vector float a0, a1, a2, b0, b1, b2,
b3, b4, b5, b6, b7, b8, c0, c1, c2;
…
// load a with SoA format
a0 = *(vector float*)a.a0;
a1 = *(vector float*)a.a1;
a2 = *(vector float*)a.a2;
...
// load b with SoA format
…
// do computation
c0 = a0*b0+a1*b3+a2*b6;
c1 = a0*b1+a1*b4+a2*b7;
c2 = a0*b2+a1*b5+a2*b8;
// store results to c with SoA format
*(vector float*)c.a0 = c0;
*(vector float*)c.a1 = c1;
*(vector float*)c.a2 = c2;
```

(a)
SoA storage +
SoA SIMDization

```
struct vec3 {float a0, float a1, float a2};
struct matrix3 {float b0, float b3, float b6, float b1, float b4,
float b7, float b2, float b5, float b8};
…
struct vec3 a[4], c[4];
struct matrix3 b[4];
vector float a0, a1, a2, b0, b1, b2, b3, b4, b5, b6, b7, b8,
c0, c1, c2;
…
// load a with AoS format, transform it to SoA with shuffles
vector float tmp0 = *(vector float*)a;
vector float tmp1 = *(vector float*)(&a[1].a1);
vector float tmp2 = *(vector float*)(&a[2].a2);
a0 = spu_shuffle(tmp0, tmp1, pattern_0360);
a0 = spu_shuffle(a0, tmp2, pattern_0125);
a1 = spu_shuffle(tmp1, tmp2, pattern_0360);
a1 = spu_shuffle(a0, tmp0, pattern_5012);
a2 = spu_shuffle(tmp2, tmp0, pattern_0360);
a2 = spu_shuffle(a0, tmp1, pattern_2501);
// load b with AoS format, transform it to SoA with shuffles
…
// do computation
c0=a0*b0+a1*b3+a2*b6
c1=a0*b1+a1*b4+a2*b7
c2=a0*b2+a1*b5+a2*b8
// transform results from SoA to AoS format with shuffles,
store them to c
...
```

(b)
AoS storage +
SoA SIMDization

```
struct vec3 {float a0, float a1, float a2};
struct matrix3 {float b0, float b3, float b6, float b1, float b4, float b7,
float b2, float b5, float b8};
…
struct vec3 a[4], c[4];
struct matrix3 b[4];
vector float a0, a1, a2, b0, b1, b2, b3, b4, b5, b6, b7, b8, c0, c1, c2;
// global addresses in main memory: gaa (for a), gab (b), gac (c)
…
// reading data from main memory to multi-layout memory
AOS_DMA_GET(a, gaa, 4*sizeof(vec3), tag, vec3);
AOS_DMA_GET(b, gab, 4*sizeof(matrix3), tag, matrix3);
// load a with SoA view
BEGIN_MULTI_VIEW(vec3);
a0 = SOA_GET(&a[0].a0);
a1 = SOA_GET(&a[0].a1);
a2 = SOA_GET(&a[0].a2);
// load b with SoA view
BEGIN_MULTI_VIEW(matrix3);
b0 = SOA_GET(&b[0].b0);
b1 = SOA_GET(&b[0].b1);
…
b8 = SOA_GET(&b[0].b8);
// do computation
c0=a0*b0+a1*b3+a2*b6
c1=a0*b1+a1*b4+a2*b7
c2=a0*b2+a1*b5+a2*b8
// store c with SoA view
BEGIN_MULTI_VIEW(vec3);
SOA_PUT(&c[0].a0, c0);
SOA_PUT(&c[0].a1, c1);
SOA_PUT(&c[0].a2, c2);
// writing data from multi-layout memory to main memory
AOS_DMA_PUT(c, gac, 4*sizeof(vec3), tag, vec3);
```

(c)
AoS storage + SoA SIMDization
+ Multi-layout memory

Figure 2: Sample vector-matrix multiplication code

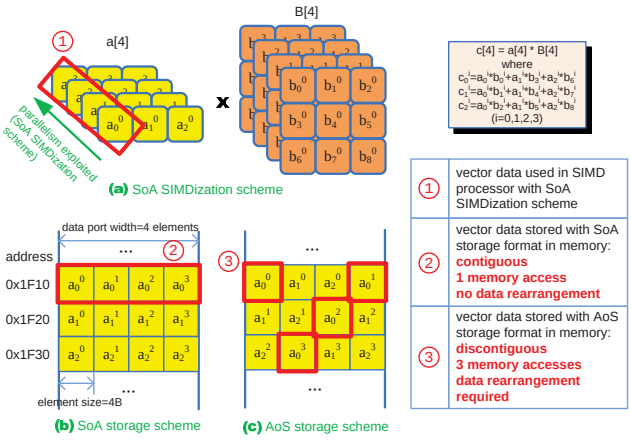

Figure 1: Vector-matrix multiplication: multiple working data sets

a 4-way SIMD processor, the first apparent drawback is that only 75% of the available bandwidth could be utilized during vector multiplications for the inner products. Afterwards, all 3 elements of the vector multiplication result have to be accumulated. However, this is not straightforward because the 3 elements are located in different vector lanes while a vector operation could be done in SIMD processors only when its operands are distributed in the same vector lane. Therefore, a sequence of data shuffle operations is necessary to rearrange the elements to be accumulated in the same lanes. Moreover, due to memory alignment restrictions in many practical SIMD systems, neither the second, nor the third column of **B** can be accessed within a single vector load; instead, they require additional load and shuffle instructions to fetch and rearrange the data into the right format. As a consequence of this rearrangement, performance is penalized. Zero-padding can be used in some applications to alleviate the data alignment problem, but at the cost of wasted memory space and additional memory bandwidth which can become prohibitively expensive for some applications (e.g. the Wilson-Dirac kernel).

Fortunately, it is common in applications with high data parallelism that the processing is to be operated upon multiple independent data sets, just as Figure 1a suggests. The SIMDization method which exploits data parallelism in single data set processing is still applicable, which will be referred to as "AoS SIMDization scheme" in the paper. However, we can also map each SIMD operation to a batch of data sets to exploit inter-dataset parallelism, which is referred to as "SoA SIMDization scheme"[1]. If the data storage scheme is SoA, illustrated in Figure 1b, optimal performance gain of four times speed-up could be potentially achieved. The example code for this case is shown in Figure 2a. However, if the data format in memory is AoS, as suggested in Figure 1c, the data rearrangement is inevitable, resulting in performance degradation. The example code for this case (Figure 2b) suggests that, e.g., 6 shuffles are required to rearrange 3 vector elements of **a** loaded from memory, which are apparently non-trivial overhead compared to the actual vector-matrix multiplication.

Problem Statement: From the above example, it can be observed that parallel processing of a batch of N data sets is more favorable for better utilization of SIMD parallel datapath and thus results in higher performance. Therefore, the SoA SIMDization scheme is preferable in most cases. Unfortunately, the most common data layout in the main memory is AoS (as briefly discussed in Section 1). This data representation discrepancy poses a significant overhead of dynamic data format conversions between the AoS and SoA.

Proposed Solution: The essential reason for the data format mismatch is that there is *no single optimal data layout*

[1]Also known as "outer-loop vectorization"[27].

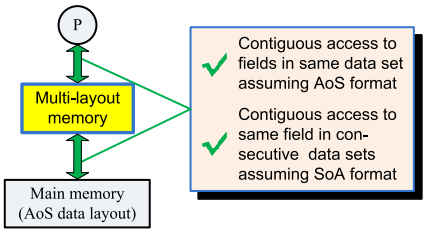

Figure 3: Proposed multi-layout memory

for different data access patterns. For operations based on indirections, the AoS storage scheme is preferable since access to fields inside a data set is contiguous; while for SIMD operations in most cases, the SoA storage scheme is favorable. To bridge the data representation gap, our idea is to design a memory system, which preserves the benefits of both AoS and SoA layouts. We call such a system "multi-layout memory" and its position and main functionalities are shown in Figure 3. In such a system, the multi-layout memory is deployed between the main memory and the SIMD processor, working as an intermediate data storage to provide contiguous data access to both data fields within the same data set (like the AoS layout) and the same field across consecutive data sets (like the SoA layout). Therefore, the penalty of dynamic conversion between the AoS and SoA data representations is completely avoided with the help of this multi-layout memory. The vector-matrix multiplication code for this case is shown in Figure 2c. It can be observed that, the programmer can easily express multiple views of the data arrays and the shuffle overhead is completely avoided.

It's worthy to mention that, assuming AoS layout in the linear address space, the AoS view of data requires unit-stride access, while the SoA view requires strided access, where the stride is determined by the size of the working data structure. We will address this issue in the following section.

3. THE EXTENDED SAMS SCHEME

3.1 Original SAMS Scheme

Given a specific physical memory organization and resources, parallel memory schemes determine the mapping from the linear address space to the physical locations, such as the module number and row address. Vector access is one of the most important memory reference patterns in SIMDized applications. Traditional parallel memory schemes in vector computers provide conflict-free access for a *single* stride family. To solve the module conflicts encountered with the cross stride family accesses, several enhancements have been previously proposed, e.g., the use of dynamic memory schemes[14, 15], use of buffers[13], use of more memory modules[13], and out-of-order vector access[32].

Recently, a parallel memory scheme, SAMS, was proposed to simultaneously support conflict-free unit-stride and strided memory accesses[19]. The idea of the SAMS hardware scheme is to use wide data line of memory modules (instead of using more modules) and make use of the wide data line to tolerate module conflicts, in order to support conflict-free access for both unit-stride and strided access patterns. More specifically, it uses 2^q memory modules with doubled memory module port width to support conflict-free vector loads/stores with vector length of 2^q. The SAMS scheme is mathematically described

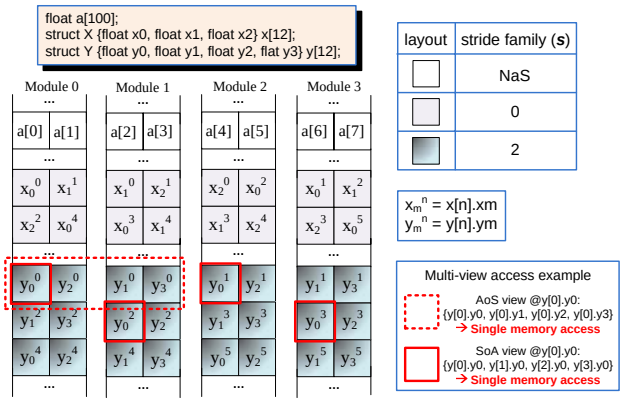

Figure 4: Internal data layouts in SAMS Multi-Layout Memory

by the following three functions[19]:

- *module assignment function:*

$$m(a) = \begin{cases} a\%2^q, & s = 0 \\ \left\langle a_q \cdots a_s,\ \left(a \otimes T_{H_{s-1,\,q+1}}\right)\%2^{s-1} \right\rangle, & 1 \le s \le q \\ \left(a \otimes T_{H_{q,s}}\right)\%2^q, & s > q \end{cases} \quad (s,\, q \in \mathbb{N})$$

- *row assignment function:*

$$r(a) = \begin{cases} \frac{a}{2^{q+1}}, & s = 0 \\ \frac{a}{2^{q+1}}, & 1 \le s \le q \\ \left(\left(\frac{a}{2^q} + 1\right)\%2^{n-q}\right)/2, & s > q \end{cases} \quad (s,\, q \in \mathbb{N})$$

- *offset assignment function:*

$$o(a) = \begin{cases} a_q, & s = 0 \\ a_{s-1}, & 1 \le s \le q \\ \overline{a_q}, & s > q \end{cases} \quad (s,\, q \in \mathbb{N})$$

3.2 Proposed Extensions

We have made two important extensions to the original SAMS scheme[19], in order to better meet the requirements and constraints of practical SIMD systems, related to (1)multi-layout support and (2)non-strided access.

(1)Multiple Data Layouts Support: In [19], it is assumed that the entire SAMS memory system adopts a single low-level address mapping (linear address↔module/row/offset) scheme and therefore manages unified internal data layout pattern. Although this simplifies the memory access since it doesn't need to indicate the stride family for which the accessed data is optimized (such information is maintained at the global scope), it significantly limits the SAMS applications, since there are many applications with multiple structured data, which require different internal data layouts for optimal access efficiency. The *Point* and *PointData* in streamcluster[5], and the *spinor* and *gauge_link* in Wilson-Dirac kernel[22] are examples[2] for such a requirement. Therefore, instead of maintaining a single low-level address mapping at the global scope for all data, our approach *customizes the address mapping logic* and manages an *individual internal data layout* for each application data, as illustrated in Figure 4. Figure 4 suggests that the *stride family* is an essential parameter in the SAMS scheme. Strided accesses with strides belonging to the stride

[2]See Section 5 for details.

family supported by the internal data layout could be accomplished in a single access; while accesses with strides from other stride families may cause module conflicts. Furthermore, it is the stride family that *configures* the low-level address mapping and the resulting internal data layout in the memory. On the other hand, the internal data layout/address mapping *determines* what stride family it supports, as illustrated in Figure 4.

The configuration granularity of the internal data layout/ address mapping is a complete 32 bytes data line. This equals $2\times sizeof$(vector register), which is determined by the SAMS hardware. Since most relevant applications tend to use large arrays, such granularity is well suited.

Obviously, with the extension to multiple data layouts, we have to keep track of the appropriate access strides and stride families for different data. Fortunately, this is not difficult as the information of the data structure and organization is static in most cases. Therefore, it is quite feasible to provide the programmer with some abstractions, e.g., C macros or library functions, to facilitate capturing such structural information, as Figure 2c illustrates. Furthermore, it is also possible for the compiler to automate the multi-layout memory usage with proper compile-time analysis and optimizations.

(2)Definition for *NaS*: By convention, the stride family $s \geq 0$. We extend this definition by introducing a special symbol NaS(not a stride), which indicates a special non-strided data storage pattern:

$$\begin{cases} m(a) & = & \frac{a}{2}\%2^q \\ r(a) & = & \frac{a}{2^{q+1}} \\ o(a) & = & a_0 \end{cases}$$

as demonstrated by the layout of **a** in Figure 4. The NaS pattern is a simple yet efficient layout for data not touched by any strided memory access. The intuition for this extension is the concern of power efficiency. For aligned and continuous accesses, it is unnecessary to invoke the majority of the AGU, ATU and In/Out Switch logic in Figure 5b[3]. Therefore, those components may be bypassed or even shutdown to save power, when the program doesn't need unaligned or strided access. In the particular case of the SAMS integration into the Cell SPE, the system further benefits from the NaS pattern. In the Cell SPE, the local store is responsible for feeding instructions as well as data to SPU, where the instruction fetch (IF) is always aligned and continuous - at the granularity of 64 bytes[17]. Therefore, the instruction fetch engine can use the NaS layout and completely remove the Out Switch in Figure 5b from the IF pipeline. The DMA engine can also benefit from this pattern for regular data accesses.

4. IMPLEMENTATION AND INTEGRATION

In this section, we investigate the implementation of the SAMS Multi-Layout Memory system and present its integration into the IBM Cell SPE.

4.1 SAMS Organization and Implementation

Figure 5a illustrates a typical memory system of a SIMD processor. To reduce hardware complexity, a logically monolithic memory module with wide data port is used to feed vector elements, which are contiguous in memory space, to the SIMD processor core. Figure 5b illustrates the organization of a multi-layout memory system based on the extended

[3]See Section 4.1 for details.

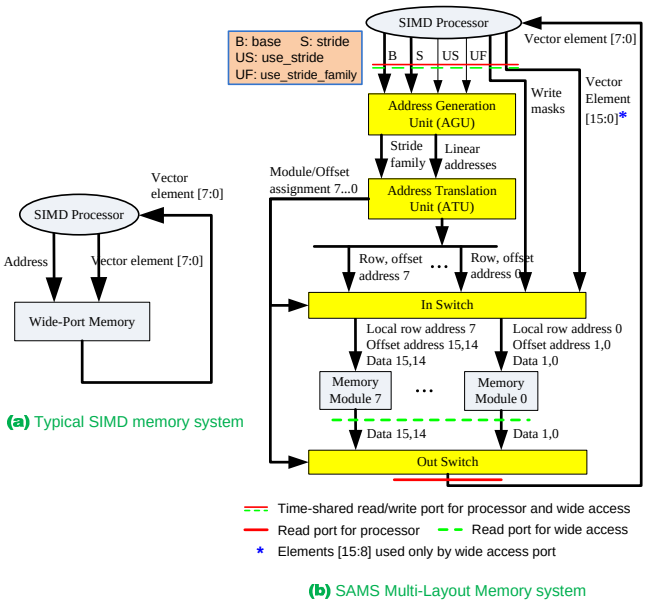

Figure 5: SIMD memory organizations

SAMS scheme. The vector processor core issues memory access commands, together with the base address and stride (note the vector length (VL)=8) to the Address Generation Unit(AGU). The eight linear addresses are generated in parallel in AGU and they are then resolved by the Address Translation Unit(ATU) into eight module assignments, eight row addresses and eight row offset addresses. Afterwards, the eight groups of row-offset pair and eight data elements from input data port (on a memory write) go to the In Switch and get routed to the proper memory modules according to their corresponding module assignments. In case of a memory read access, after the read latency of the memory modules[4], eight read data are fed back to the vector processor through the Out Switch at the bottom of Figure 5b. Two additional latencies are incurred by the integration of extended SAMS scheme to the original SIMD memory system of Figure 5a: a)"inbound path", which includes AGU, ATU and In Switch; b)"outbound path", which consists of the Out Switch only.

Address Generation Unit (AGU): is responsible for parallel generation of the addresses of the 2^q vector elements. Also, it computes the stride family.

Address Translation Unit (ATU): determines the internal data layout of the SAMS Multi-Layout Memory and input/output data permutation patterns used in In Switch and Out Switch. ATU consists of three independent components: the module assignment logic, the row assignment logic and the offset assignment logic. Therefore, the critical path of ATU is the longest of the three, which is the $n-q$ bit adder followed by a 2-to-1 multiplexor in the row assignment logic[19].

In Switch and Out Switch: In the SAMS Multi-Layout Memory system, the InSwitch is a 2^q by 2^{q+1} crossbar, while the OutSwitch is a 2^{q+1} by 2^q crossbar[19].

Unaligned Vector Access: Unaligned vector memory ac-

[4]The access latency of a memory module may be more than one clock cycle. In the paper, we assume the memory modules are fully pipelined.

Table 1: Synthesis results of SAMS Multi-Layout Memory system

	Critical Path Delay [ns]			Logic Complexity [# of gates]		
	$q=2$	$q=3$	$q=4$	$q=2$	$q=3$	$q=4$
SAMS memory logic	0.76	0.87	1.01	6,906	2,6784	82,538
equivalent # of 32-bit adders	2.0	2.3	2.7	1.5	5.9	18.1

cess is one of the critical problems in SIMD processing systems[26, 31]. The SAMS Multi-Layout Memory system supports unaligned unit-stride and strided vector loads and stores. Details of a similar technique can be found in[4].

Memory Store Granularity: With 2^q memory modules instead of a monolithic memory module, the store granularity of the SAMS Multi-Layout Memory system is reduced from an entire vector of 2^q elements to a single element. For example, the monolithic local store of IBM Cell SPE only supports loads/stores at the granularity of 128 bits; while with the SAMS scheme with 4 memory modules and element size of 32 bits, stores of 1, 2 or 4 32-bit elements are well supported.

Wide Port Support: The SAMS scheme utilizes wide data lines to tolerate module conflicts[19]. More specifically, each of the eight modules in Figure 5b has a data port width of two elements and the eight memory modules are capable of servicing 16 elements per access, under the condition that it is aligned to 16 elements boundary. To avoid additional hardware complexity, the wide access port in Figure 5b is not responsible for reordering the 16 data elements during a wide access. Indeed, the wide port behaves the same as an ordinary linear memory interface: it directly reads or writes all the 16 data elements from/to the 8 memory modules with the row address of $\frac{base}{32}$(assuming 4B element size), effectively bypassing all the SAMS logic. Therefore, for a read of a full data line of 16 elements from the SAMS Multi-Layout Memory, the external data consumer has to do a post read shuffle after reading the data. For a write, a pre-write shuffle is also necessary, since the internal data layout of the SAMS Multi-layout Memory has non-linear structure as indicated in Figure 4. For the external data provider/consumer of the SAMS memory, there is a trade-off between the bandwidth and hardware complexity. We shall further discuss this in Section 4.2.

Implementation and Synthesis Results: We have implemented the SAMS Multi-Layout Memory system using Verilog and synthesized it for TSMC 90nm Low-K process technology using Synopsis Design Compiler. Synthesis results are provided in Table 1 for SAMS memory systems with 4, 8 and 16 memory modules, i.e., $q = 2$, 3 and 4, which target 4-way, 8-way and 16-way SIMD processing systems respectively. The critical path delays in Table 1 actually present the inbound path. We also calculated the relative delay and area consumption of the SAMS system compared to a 32-bit adder synthesized on the same technology node. Further investigation into the synthesis results indicates that the ATU, which is the core of the SAMS scheme, has quite fast and compact hardware implementation: it only contributes to approximately $\frac{1}{5}$ of the entire critical path delay and its contribution to the overall area is even much smaller.

4.2 Integration into the Cell SPE

To validate the performance of the proposed SAMS Multi-Layout Memory in real applications, we implemented it in a model of the IBM Cell processor, aiming at computation intensive applications with high data parallelism[23, 30]. The local store of the Cell Synergistic Processing Element (SPE) is chosen for the deployment and implementation of the multi-layout memory system. Figure 6a depicts the original local store memory organization in the Cell SPE. The fully pipelined 256KB local store is composed of 4 m64k SRAM modules (running at the same speed as the SPU core[16]). Note, the SRAM arrays are themselves single-ported, therefore, the local store is accessed in a time-shared manner, as sketched in Figure 6a (only the load path is shown for simplicity).

The integration of SAMS Multi-Layout Memory is illustrated in Figure 6b. It is also referred to as "SAMS local store" in our experiments in Section 5. Note, although in Figure 6b each m64k module is split into four submodules, the total size of the SAMS local store is kept the same as the original one. The "splitting" of SRAM arrays may not incur additional engineering effort, since the original m64k is composed of 32 subarrays in the Cell local store physical implementation[16].

An important change in hardware due to the SAMS integration is on the 128B wide port buffers. As high bandwidth of the wide port is extraordinarily desirable for both instruction fetch and DMA in SPE, we choose to provide full bandwidth of the SAMS memory for the wide port. As discussed in Section 4.1, in this case the wide port data of each SAMS duplicate needs to be aligned to 32B boundary (this is guaranteed by the 128B access granularity of the original local store wide port), and the data format needs to be adjusted to the internal layout in SAMS memory modules. The latter requires two macros, the Post-Read Shuffle(PRS) and the Pre-Write Shuffle(PWS), to be added to the system, as suggested in Figure 6b. The critical path delay and area of PRS and PWS are comparable to those of the SAMS ATU, which has less than one cycle latency and trivial hardware consumption as discussed in 4.1.

The major impact on the SPU microarchitecture with the incorporation of the SAMS Multi-Layout Memory in the SPE local store is that the local store pipeline is lengthened, since the SAMS logic introduces additional delay. According to our synthesis results in Table 1, the critical path delay of the SAMS inbound path for 4 memory modules (each with 64-bit port width) is 0.76ns, which corresponds to two times the latency of a 32-bit adder. In the Cell SPU, 32-bit addition in the Vector Fixed Point Unit (FPU) fixed-point is accomplished in a single cycle[25]. Therefore, we project the deployment of the SAMS memory in SPE's local store will introduce 2(very stringent) or 3(considering pipeline latches and retiming costs) additional pipeline stages for the inbound path. The outbound path takes one additional cycle since it has a critical path delay of 0.35ns which is less than the 0.38ns latency of a 32-bit adder in our study. To summarize, 4 cycles for a load (inbound and outbound paths) and 3 cycles for a store (only inbound path involved) is a realistic estimation for the extra latency incurred by the integration of the SAMS Multi-Layout Memory logic inside the SPU pipeline in our study. As in the original SPE[30] the load and store instructions take 6 and 4 clock cy-

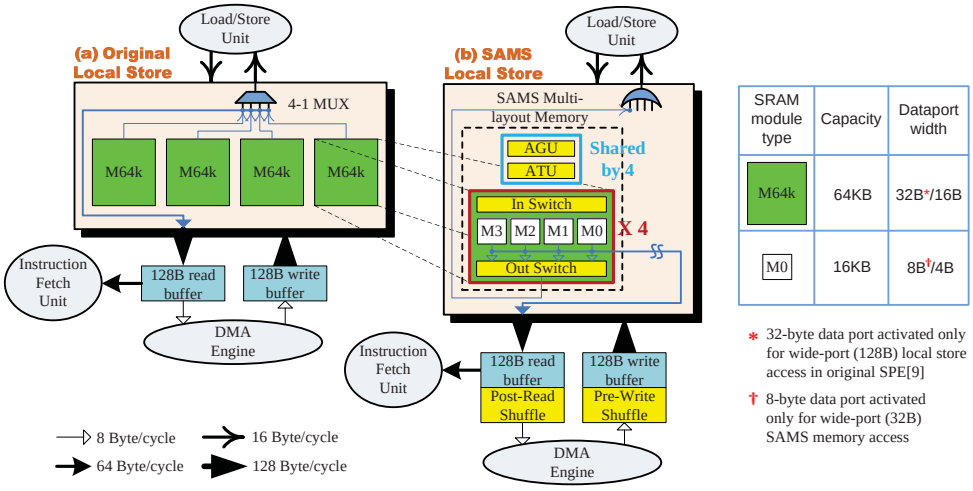

Figure 6: Integration of the SAMS Multi-Layout Memory into the Cell SPE

cles respectively, their costs will become 10 and 7 cycles in the modified SPE pipeline.

5. EXPERIMENTAL EVALUATION

Experimental Setup: We use CellSim developed at BSC[1], which is a cycle-accurate full system simulator for IBM Cell/BE processor. The benchmarks of our experiments consist of some full applications from PARSEC[5], the Wilson-Dirac kernel from INRIA[22], and some micro kernels from IBM Cell SDK[2]. These applications/kernels are selected since they are representative of application codes which operate heavily on array-based data structures. This type of code is widely used in scientific and engineering applications. Table 2 lists the major features of the selected benchmarks. Streamcluster from PARSEC is an online clustering kernel which takes streaming data points as input and uses a modified k-median algorithm to do the online clustering. The parameters of streamcluster workload in our study are set as follows: 1024 input points, block size 256 points, 10 point dimensions, 5-10 centers, up to 500 intermediate centers allowed. Fluidanimate is an Intel RMS application and it uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes[5]. The fluidanimate workload in our experiments uses the *sim-small* input set provided by PARSEC. The 3D working domain for one SPE has been shrunk to 9x9x30 cells and the maximal number of particles inside a cell has been reduced to 10. The simulation runs for one time step to compute one frame. Computing the actions of Wilson-Dirac operator contributes most of the CPU time in the simulation of Lattice Quantum Chromodynamics (Lattice QCD), which aims at understanding the strong interactions that bind quarks and gluons together to form hadrons[22]. The experiments in our study with the Wilson-Dirac kernel focus on single SPE for floating point data, with the 4-way runtime data fusion scheme proposed in [22]. The problem size for the Wilson-Dirac kernel for single SPE is 128 output spinors, with a computation intensity of 1608 FP (floating point operations) per output spinor. Besides full applications and large kernels, we also include some micro kernels, including complex number multiplication[21] with workload set to 10K multiplications and 4x4 matrix transpose[2] (with workload set to 10K transposes).

Table 2: Selected benchmark suit

Benchmark	Source	Type	Application Domain	Working Set
streamcluster	PARSEC	kernel	data mining	medium
fluidanimate	PARSEC	application	animation	large
Wilson-Dirac Operator	INRIA	kernel	quantum physics	medium
complex multiplication	Cell SDK	micro kernel	-	small
matrix transpose	Cell SDK	micro kernel	-	small

To compile the C code, we use two stand-alone compilers: PPU toolchain 2.3 (based on gcc 3.4.1) and SPU toolchain 3.3 (based on gcc 4.1.1). All benchmark applications and kernels are compiled with the -O1 option.

To make the functionalities of the SAMS Multi-Layout Memory available to software, we have extended the SPU ISA and the programming interface. Table 3 lists some of the new instructions, C intrinsics and macros for the enhanced SPE with SAMS integration (also referred to as "SAMS SPE"). To reflect the changes in the architecture, we have modified the spu-gcc backend to generate optimal code for the SAMS SPE, including automatic selection of appropriate instructions for unaligned memory access and flexible access granularity. The load latency and branch penalty have also been updated for proper instruction scheduling. Besides the compiler, the CellSim simulator has also been modified accordingly.

It should be noted that although there are eight SPEs available in the Cell processor, we only use a single one in our experiments, since we want to focus on the performance impact of the SAMS Multi-Layout Memory on SIMDization. Our techniques are orthogonal to those for efficient parallelization of data parallel applications on multiple processor cores.

Benchmarks SIMDization: In both applications of streamcluster and fluidanimate, the house keeping work (such as data preparation), scalar code dominated by branches (such as building the neighbor table in fluidanimate), and work suited to be done in the global scope (e.g., rebuilding the grid in

Table 3: New instructions, intrinsics and macros with the SAMS Multi-Layout Memory

Name	Type	Operation
setstride	instruction	set stride register
lqwsd/x/a	instruction	load a quad word with $stride = stride\,register$ and $s = log_2(stride)^\dagger$, d/x/a-form
stqwsd/x/a	instruction	store a quad word with $stride = stride\,register$ and $s = log_2(stride)^\dagger$, d/x/a-form
lqwsfd/x/a	instruction	load a quad word with $stride = 1$ and $s = log_2(stride\,register)^\dagger$, d/x/a-form
stqwsfd/x/a	instruction	store a quad word with $stride = 1$ and $s = log_2(stride\,register)^\dagger$, d/x/a-form
spu_sams_setstride (imm)	intrinsic	set stride register to value imm
spu_sams_lqws (a)	intrinsic	load a quad word at base address a with $stride = stride\,register$ and $s = log2(stride)^\dagger$ (unified form for lqwsd/x/a)
spu_sams_stqws (a,val)	intrinsic	store quad word val at base address a with with $stride = stride\,register$ and $s = log2(stride)^\dagger$ (unified form for stqwsd/x/a)
spu_sams_lqwsf (a)	intrinsic	load a quad word at base address a with $stride = 1$ and $s = log2(stride\,register)^\dagger$ (unified form for lqwsfd/x/a)
spu_sams_stqwsf (a,val)	intrinsic	store quad word val at base address a with with $stride = 1$ and $s = log2(stride\,register)^\dagger$ (unified form for stqwsfd/x/a)
BEGIN_MULTI_ VIEW(str)	macro	spu_sams_setstride(sizeof(str))
SOA_GET(a)	macro	spu_sams_lqws(a)
SOA_PUT(a,val)	macro	spu_sams_stqws(a,val)
AOS_GET(a)	macro	spu_sams_lqwsf(a)
AOS_PUT(a,val)	macro	spu_sams_stqwsf(a,val)
AOS_DMA_GET (la,ga,size,tag,str)	macro	spu_sams_mfcdma64(la,mfc_ea2h(ga),mfc_ea2l(ga),size,tag, MFC_GET_CMD,log2sizeof(str)‡)
AOS_DMA_PET (la,ga,size,tag,str)	macro	spu_sams_mfcdma64(la,mfc_ea2h(ga),mfc_ea2l(ga),size,tag, MFC_PUT_CMD,log2sizeof(str)‡)

†Stride family(s) calculation is done in AGU (see Figure 5b).

‡*log2sizeof* is a new C keyword we implemented in spu-gcc (log2sizeof(str)=log_2(sizeof(str)), where log_2 is done at compile time).

fluidanimate) is done by PPU, while the majority of the computation is offloaded to the SPU. When data are ready in the main memory, the PPU triggers SPU to start processing. SPU reads a portion of data into its local store by DMA transfers, processes them and writes (using DMA) the results back to main memory. The baseline Wilson-Dirac kernel is already SIMDized and heavily optimized on the original SPE using SoA SIMDization scheme. Therefore, our optimizations based on the SAMS SPE only involve the elimination of dynamic data format conversion overhead. The micro kernels of complex number multiplication and matrix transpose are normally used as one step in a sequence of data operations in the SPE local store. Therefore, DMA transfers are not invoked in our experiments with them.

It should be noted that for all experiments except streamcluster, the SoA SIMDization scheme is adopted since it gives better performance over AoS, in both the original and the SAMS SPEs. As for the memory access patterns, the AoS access of the SAMS local-store is used in all benchmarks for data transfers between the main memory and the local-store (illustrated in Figure 1c). During execution, however, four benchmarks access the local-store with the SoA access and one - streamcluster - uses both SoA/AoS.

Experimental Results: We evaluate the performance by measuring the application execution time on the SPU. Table 4 depicts the results of our experiments.

For the streamcluster benchmark, it is not obvious whether the AoS or the SoA SIMDization scheme gives better performance, since it depends on the input data, the load latency and the quality of spu-gcc instruction scheduling. Although the critical loop is SIMDized with the AoS SIMDization scheme, for the rest of the code, the loops involving distance calculation are SIMDized with SoA scheme, to achieve better SIMD datapath utilization. The major performance improvement of the SAMS SPE comes from the support for unaligned vector and scalar memory accesses in the SAMS local store, quantified by up to 19% reduction of memory instructions and 43% of total instructions. However, the two-level indirection has serious negative impact on performance especially in the SAMS SPE (it has longer load latency). Moreover, the large number of branches in the source code which could not be well handled by SPU also incur substantial performance overhead - around 14% execution time is on IF stall for the original SPE and 20% for the SAMS SPE since it has longer IF latency. Therefore, the overall performance gain is only 11%, as indicated in Table 4. For such applications, further effort to make major modifications to both the data representation and the control flow at algorithm level would pay-off for better SIMDization performance, on both the original and the SAMS SPEs. Nonetheless, streamcluster represents a class of applications where both AoS and SoA SIMDization schemes are applied on the same data at different application phases. In such cases, the SAMS Multi-Layout Memory's capability of providing multiple data views with high efficiency enables flexible choices of optimal SIMDization schemes in different scenarios.

In fluidanimate, most execution time is spent on computing the density and the acceleration for each particle, by accumulating densities and forces between the current particle and all neighboring particles in a valid range. Since the data parallelism in computing a single particle pair is very limited, the SoA SIMDization scheme is used to vectorize the code, so that in each batch of processing the interoperation between 4 particles in the current cell and a particle in a neighboring cell are evaluated. Although optimizations are equally deployed in both the original and the SAMS SPEs, the SAMS SPE gives

Table 4: Comparison of SPU dynamic instruction count and execution time

Benchmark	Memory Instruction Count			Total Instruction Count			Execution Time (cycles)		
	Original SPE (load/store)	SAMS SPE (load/store)	R	Original SPE	SAMS SPE	R	Original SPE	SAMS SPE	R
streamcluster	12,790,400 (11,084,211/ 1,706,189)	10,331,056 (9,079,100/ 1,251,956)	19%	78,850,693	45,117,363	43%	251,783,087	224,622,745	11%
fluidanimate	4,888,924 (3,043,109/ 1,845,815)	1,772,207 (680,436/ 1,091,771)	64%	33,106,575	22,515,460	32%	95,284,379	74,148,296	22%
Wilson-Dirac Operator	13,863 (13,077/786)	11,375 (10,551/824)	18%	77,038	63,510	18%	75,975	62,035	18%
complex multiplication	15,392 (10,256/5,136)	15,419 (10,272/5,147)	0%	42,008	26,693	36%	96,242	73,137	24%
matrix transpose	81,950 (40,974/40,976)	81,979 (40,992/40,987)	0%	167,766	85,893	49%	283,121	179,690	37%

superior performance for three major reasons. First, the 3D position, velocity and acceleration data are maintained in AoS format in main memory, therefore the dynamic format conversion overhead is incurred in the original SPE. Second, as the 3D particle data components are not aligned to 16B memory addresses, an alignment problem occurs in the original SPE. Third, frequent scalar memory access in the code incurs significant performance overhead in the original SPE. All three overheads are removed in the SAMS SPE, leading to the 22% reduction of execution time as presented in Table 4. Note, the number of stores (3rd column of Table 4) in our SoA SIMDized fluidanimate is significantly larger than the loads. The reason is explained as follows. As the actual number of particles in current cell (maximal 10) is known only at runtime, 3 batches of processing (dealing with 4, 4, and 2 particles in the current cell respectively) are necessary for computing one neighboring particle. Furthermore, each batch has to update the density and acceleration of the current particles and the neighboring particle. As a result, the stores of the particle data are doubled for current cells with $5 \sim 8$ particles and tripled for current cells with $9 \sim 10$ particles, compared to current cells with only $1 \sim 4$ particles, for each neighboring particle. In contrast to the duplicated stores, the loads of particles in the current cell are shared among all particles in all neighboring cells, and the load of a neighboring particle is invoked only once regardless of the number of particles in the current cell. This explains the unusual disparity between the number of dynamic loads and stores in the SAMS SPE. In the original SPE, the number of loads (2nd column in Table 4) are larger than the stores because each unaligned load incurs two aligned loads and each unaligned store incurs a load-modify-store sequence.

The main problem that prevents SIMDizing the computation of Wilson-Dirac operator efficiently is the multiplicity of patterns accessing the same *spinor* and *gauge link* data[22]. For efficient SIMDization, the authors introduce the runtime data fusion technique in [22], which is basically a SoA SIMDization scheme with rearrangement of data from AoS to SoA format at runtime. Consequently, it also suffers from the overhead of dynamic data format conversion. With our SAMS Multi-Layout Memory, the spinor and gauge link data are accessed in both AoS and SoA formats (for computation) with high efficiency, therefore the data rearrangement overhead is completely eliminated. Additionally, with the original SPE code there are 80 loads overhead per 4 output spinors (therefore 20*128=2560 loads overhead in total for 128 spinors) due to partial use of the loaded spinor and gauge filed data. The partial use of memory bandwidth also results from the mismatch between the data layout in the local store and the one used in the SPU. With the SAMS local store, such overhead is effectively removed, resulting the reduced number of executed load instructions. Altogether, the execution time is reduced by 18% in the SAMS SPE, as indicated in Table 4.

In complex number multiplication, with the common AoS representation of a complex number array, load and store of the real/imaginary part of consecutive complex numbers require a SoA view of the array. With the multiple view capability of the SAMS memory, the real vector and imaginary vector can be loaded directly with one single strided access, instead of loading the mixture of them and extracting the real and imaginary parts using shuffle instructions, as the source code in [21] did. Therefore, although the memory accesses count is the same for both the original and the SAMS SPEs, the kernel still achieved a significant performance gain in the proposed SPE as a result of glue instructions reduction.

In 4x4 matrix transpose, if each row (4 elements) of the matrix is treated as a basic structure, and the original matrix is stored in the AoS format, then accessing a row requires an AoS view of the row array, while accessing a column requires a SoA view. With the SAMS local store, the transpose procedure is accomplished in a simple manner: first load 4 columns of the input matrix directly (with the SoA view) into registers (transpose the matrix), and then store the transposed result to the output buffer. All shuffle instructions to pack the transposed rows from the original rows in the original SPE are completely eliminated. This explains the significant performance improvement of 37% by the SAMS SPE suggested in Table 4.

6. RELATED WORK

Vector supercomputers have traditionally relied on expensive, large number of SRAM memory banks used as main memory to achieve high memory bandwidth at low latency[10, 12]. Many of them are optimized for unit-stride[10], which lead to serious bank/module conflicts with other strided vector access patterns[6, 28]. To cope with module conflicts of vector accesses across stride families, several techniques have been proposed in the literature, including the use of buffers[13], dynamic memory schemes[14, 15], memory modules clustering[13], and intra-stream out-of-order access[32], just to name a few. These schemes have certain constraints or limits such as degraded performance for short vectors, data flush and reload

penalty for stride change, and waste of memory module resources[19]. In the Tarantula project, a conflict-free memory scheme is devised and deployed in the L2 cache to support strided vector access with strides from a range of stride families, by using the wide cache line[3]. The limit is that it also requires long vectors to achieve conflict-free access.

The SAMS parallel memory scheme was recently proposed as a pure hardware scheme, to simultaneously support conflict-free unit-stride and strided memory accesses for short vectors[19]. The general idea of this paper is independent of any specific hardware memory-mapping scheme. Nevertheless, we chose the SAMS scheme because of its capability to support both contiguous and strided memory access with atomicity, predictability, and low hardware complexity. Although we built our multi-layout memory upon the extended SAMS address mapping scheme, the idea of our work is that by conveying the static structural information of the data to the memory system, the memory access patterns to the structured data can be captured and handled with improved efficiency. To do so, we expose the configuration of the low-level address mapping logic of each linear memory region (at the granularity of $2\times sizeof$(vector register)) to the architecture level (with 2 sets of special load/store instructions and 1 special-purpose *stride register*). We further abstract such a configuration with C intrinsics/macros to create a simple yet easy-to-use programming interface. In such a way, the layout of each individual memory region can be customized with specific address mapping logic, in order to provide "conflict-free" accesses to the data for which the memory region is allocated.

There has been a lot of work in exploiting the memory access pattern information, by dynamically predicting or statically capturing the access patterns/strides, e.g., stream buffers[29] and software prefetching[7, 9]. Although existing prefetching techniques can do a good job in reducing memory latency for scalar processors, they are not as efficient for SIMD processors, since prefetched data elements may reside among multiple cache lines, causing non-trivial efforts to gather/rearrange. Our SAMS Multi-Layout Memory not only maintains the required data as a fast local store, but also guarantees the correct data format - data elements are distributed in the local store *without bank-conflicts* and can be accessed simultaneously, without any need for rearrangement in software.

Some recent research projects also consider manipulating the memory address to improve memory access performance. The Impulse memory [8] remaps physical addresses of discontinuous data to an alias contiguous in the shadow space. References to the discrete data through the alias are actually performed by the Impulse Memory Controller at the DRAM side. While improving the cache and memory bus utilization, it is not suitable for on-chip local stores, as data at the memory side still remain discontinuous and the efficiency of the data access remains low. Our work differs from Impulse in that we adjust physical locations of data among multiple memory banks/rows/offsets to exploit the memory access parallelism. We also avoid the coherence problem as the SAMS memory does not create any alias for discontinuous data. Similar to Impulse, the active memory system[24] uses the address remapping to create contiguous aliases for discontinuous data, and access the data with their aliases, to hijack the memory hierarchy for better cache behavior. Again, they are unable to improve the efficiency of the memory access at the memory side. The Command Vector Memory System[11] proposes broadcasting vector access commands to all memory modules/banks, instead of sending individual addresses/data. Despite its inherent support for strided access, the Command Vector Memory System does not consider special address mapping schemes to exploit memory access parallelism among multiple banks for strided access.

Regarding the data alignment problem in GPP SIMD extensions, studies have been done to improve the performance of SIMD devices by relieving the impact of non-contiguous and unaligned memory access from both the compiler and hardware point of view. For example, Ren et al proposed a compiler framework to optimize data permutation operations by reducing the number of permutations in the source code with techniques such as permutation propagation and reduction[31]. Nuzman et al [26] developed an auto-vectorization compilation scheme for interleaved data access with constant strides that are power of two. Alvarez et al [4] analyzed the performance of extending the Altivec SIMD ISA with unaligned memory access support on H.264/AVC codec applications. Our work also addresses the data rearrangement overhead in SIMD processors in general and the pragmatic problem of data format conversion between AoS and SoA in particular. However, our special focus is on the demonstration of a parallel memory system with configurable low-level address mapping logic at proper granularity and the accompanying software abstractions to improve the memory access efficiency.

7. CONCLUSIONS

In this paper, we proposed the SAMS Multi-Layout Memory to solve the data rearrangement problem in general and to reduce the dynamic data format conversion overhead in particular. The idea is to easily express the preferred view of the data structures in software and let the hardware customize the low-level address mapping logic for optimal data access using this information. Synthesis results for TSMC 90nm CMOS technology node suggest reasonable latency and area overhead of the proposed SAMS memory. To investigate the performance improvement gains, SAMS was integrated into the IBM Cell SPE model, and simulated using real applications. Experiments suggest that, for applications which require dynamic data format conversions between AoS and SoA, our multi-layout memory hardware together with the accompanying software abstractions improve the system performance by up to 37% and simplify the program SIMDization.

Acknowledgments

This research has been supported by the European Commission in the context of the FP6 project SARC (FET-27648) and by the Dutch Technology Foundation STW, applied science division of NWO and the Technology Program of the Dutch Ministry of Economic Affairs (project DCS.7533).

References

[1] http://pcsostres.ac.upc.edu/cellsim/doku.php.

[2] http://www.ibm.com/developerworks/power/cell/.

[3] A. Seznec and R. Espasa. Conflict-free accesses to strided vectors on a banked cache. *IEEE Trans. Computers*, 54(7):913–916, 2005.

[4] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. Performance impact of unaligned memory operations in SIMD extensions for video codec applications. In *ISPASS '07: Proceedings of the 2007 International Symposium on Performance Analysis of Systems and Software*, pages 62–71, 2007.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[6] P. Budnik and D. J. Kuck. The organization and use of parallel memories. *IEEE Trans. Comput.*, 20(12):1566–1569, 1971.

[7] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, 1991.

[8] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *HPCA '99: Proceedings of the 5th international symposium on High Performance Computer Architecture*, pages 70–79, 1999.

[9] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. In *ISCA '94: Proceedings of the 21st annual International Symposium on Computer Architecture*, pages 223–232, 1994.

[10] T. Cheung and J. E. Smith. A simulation study of the CRAY X-MP memory system. *IEEE Trans. Comput.*, 35(7):613–622, 1986.

[11] J. Corbal, R. Espasa, and M. Valero. Command vector memory systems: High performance at low cost. In *PACT '98: Proceedings of the 1998 international conference on Parallel Architectures and Compilation Techniques*, pages 68–77, 1998.

[12] Cray Research Inc. Cray Y-MP C90 system programmer reference manual. 1993.

[13] D. T. Harper III. Block, multistride vector and FFT accesses in parallel memory systems. *IEEE Trans. Parallel and Distributed Systems*, 2(1):43–51, 1991.

[14] D. T. Harper III. Increased memory performance during vector accesses through the use of linear address transformations. *IEEE Trans. Comput.*, 41(2):227–230, 1992.

[15] D. T. Harper III and D. A. Linebarger. Conflict-free vector access using a dynamic storage scheme. *IEEE Trans. Comput.*, 40(3):276–283, 1991.

[16] S. Dhong, O. Takahashi, M. White, T. Asano, T. Nakazato, J. Silberman, A. Kawasumi, and H. Yoshihara. A 4.8GHz fully pipelined embedded SRAM in the streaming processor of a Cell processor. In *Proceedings of IEEE Int'l Solid-State Circuits Conference 2005*, pages 486–612, 2005.

[17] B. Flachs, S. Asano, S. Dhong, P. Hotstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processing unit for a Cell processor. In *Proceedings of IEEE Int'l Solid-State Circuits Conference 2005*, pages 134–135, 2005.

[18] T. Forsyth. SIMD programming with Larrabee. *http://software.intel.com/file/15545*.

[19] C. Gou, G. K. Kuzmanov, and G. N. Gaydadjiev. SAMS: Single-Affiliation Multiple-Stride parallel memory scheme. In *MAW '08: Proceedings of the 2008 Workshop on Memory Access on future processors*, pages 359–367, 2008.

[20] IBM Systems and Technology Group. Cell BE programming tutorial v3.0. *http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788*.

[21] IBM Systems and Technology Group. Developing code for Cell - SIMD. *www.cc.gatech.edu/~bader/cell/Day1-06_DevelopingCodeforCell-SIMD.ppt*.

[22] K. Z. Ibrahim and F. Bodin. Implementing Wilson-Dirac operator on the Cell Broadband Engine. In *ICS '08: Proceedings of the 22nd annual International Conference on Supercomputing*, pages 4–14, 2008.

[23] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. & Dev.*, 49(4/5):589–604, 2005.

[24] D. Kim, M. Chaudhuri, M. Heinrich, and E. Speight. Architectural support for uniprocessor and multiprocessor active memory systems. *IEEE Trans. Comput.*, 53(3):288–307, 2004.

[25] N. Mäding, J. Leenstra, J. Pille, R. Sautter, S. Büttner, S. Ehrenreich, and W. Haller. The vector fixed point unit of the Synergistic Processor Element of the Cell architecture processor. In *DATE '06: Proceedings of the conference on Design, Automation and Test in Europe*, pages 244–248, 2006.

[26] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI '06: Proceedings of the 2006 conference on Programming Language Design and Implementation*, pages 132–143, 2006.

[27] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short simd architectures. In *PACT '08: Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques*, pages 2–11, 2008.

[28] W. Oed and O. Lange. On the effective bandwidth of interleaved memories in vector processor systems. *IEEE Trans. Comput.*, 34(10):949–957, 1985.

[29] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA '94: Proceedings of the 21st annual International Symposium on Computer Architecture*, pages 24–33, 1994.

[30] D. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, H. Harvey, P.M. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation Cell processor. *IEEE Journal of Solid-State Circuits*, 41:179–196, 2005.

[31] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 118–131, 2006.

[32] M. Valero, T. Lang, M. Peiron, and E. Ayguade. Conflict-free access for streams in multimodule memories. *IEEE Trans. Comput.*, 44:634–646, 1995.