Neural, Parallel, and Scientific Computations 19 (2011) 91-110

MAT-CORE: A DECOUPLED MATRIX CORE EXTENSION FOR GENERAL-PURPOSE PROCESSORS

MOSTAFA I. SOLIMAN
Computer & System Section, Electrical Engineering Department,
Aswan Faculty of Engineering,
South Valley University, Aswan, Egypt

Abstract. This paper proposes new processor architecture to exploit the increasingly number of transistors per integrated circuit and improve the performance of many applications on general-purpose processors. The proposed processor (called Mat-Core) is based on the use of multi-level ISA to explicitly communicate data parallelism to processor in a compact way instead of the dynamic extraction using complex hardware or the static extraction using sophisticated compiler techniques. Scalar-scalar (level-0), scalar-vector (level-1), vector-vector (level-1), vector-matrix (level-2), and matrix-matrix (level-3) instruction sets are used as a multi-level interface between hardware and software. Mat-Core extends a general-purpose scalar processor (for executing scalar instructions) with a matrix unit (for executing vector/matrix instructions). To tolerate the memory latency, the extended matrix unit is decoupled into two components: address generation and data computation. The data computation unit is organized in parallel lanes; each lane contains a pipeline of each functional unit and a slice of the matrix register file. On parallel lanes, the Mat-Core processor can effectively process not only vector but also matrix data. This paper explains the execution of vector/matrix instructions on the parallel lanes of Mat-Core. Moreover, the performances of element-wise vector-vector addition, vector-matrix multiplication, and matrix-matrix multiplication are estimated on the decoupled Mat-Core processor. The increasingly budget of transistors can be exploiting to scale the Mat-core processor by providing more cores in a physical package. On a Multi-Mat-Core processor, performance would be improved by parallel processing threads of codes using multi-threading techniques.

Keywords - high-performance computing, parallel architectures, vector/matrix processing, decoupled architectures, and multi-core computation.

1. INTRODUCTION

The key to achieving high performance with all modern microprocessors is the presence of parallelism in applications, for it allows the hardware to accelerate applications by executing multiple, independent operations concurrently [1]. Computer architects have employed various forms of parallelism to provide increases in performance above those made possible just by improvements in underlying circuit technologies. Beyond pipelining technique, which is now universally applied in all types of computing systems, there are several ways in which processor designs can exploit parallelism to improve performance. The three major forms are instruction-level parallelism (ILP), thread-level

Received April 25, 2010

parallelism (TLP), and data-level parallelism (DLP) [1, 2]. These various forms of machine parallelism are not mutually exclusive and can be combined to yield systems that can exploit all forms for application parallelism. For example, Intel multi-core processors are pipelined superscalar processor, which can exploit ILP, TLP, and DLP using superscalar techniques, multi-threading computations, and multimedia extension instruction sets, respectively [3].

Exploiting ILP was the primary focus of processor designs (superscalar [4] and VLIW [5]) for about 20 years starting in the mid-1980s to improve the processor performance by parallel processing multiple scalar instructions per clock cycle. Superscalar architectures have used the increasable chip resources to dynamically extracting and dispatching more independent scalar instructions in the same clock cycle. However, VLIW architectures have increased the number of decoders and the execution datapaths to process more parallel scalar instructions explicitly packed by the compiler into a very long instruction word. Recently, the limits of power, available ILP, and long memory latency have slowed uniprocessor performance from about 52% to about 20% per year [1]. Moreover, superscalar and VLIW microprocessors use scalar instruction set architecture (ISA) as an interface between hardware and software, which cannot express parallelism to hardware (processor).

On the software side, applications based on DLP are growing in importance and demanding increased performance from hardware [6, 7]. These applications include 3D graphics, image processing, signal processing, voice recognition, network processing, scientific and engineering applications, etc. To satisfy the performance demand, specialized hardware is commonplace for these applications. Otherwise, a general-purpose scalar processor needs to perform fetching, decoding, executing, and writing a result for each scalar instruction. This traditional way for processing data parallel applications using scalar ISA is not the best, even though multiple scalar instructions can be extracted easily by hardware or compiler [8]. Recently, major microprocessor vendors have announced extensions to their general-purpose microprocessors in an effort to process multiple data by using a single instruction (SIMD) to improve the performance data parallel applications [9, 10]. For example, Intel processors have been extended with MMX, SSE, SSE2, SSE3, SSSE3, SSE4, and AVX [11]. Moreover, Sun enhanced Sparc with VIS, Hewlett-Packard added MAX to its PA-RISC architecture, Silicon Graphics extended the MIPS architecture with MDMX, Motorola extended the PowerPC with AltiVec, etc.

Although these extensions are a good step toward incorporating vector architecture into microprocessor level, they have some disadvantages. They have limited vector instruction sets with fixed vector length and stride; one instruction may keep one datapath busy for a few clock cycles; wide datapaths can be used after either changing the ISA or the issue width; multiple instructions are needed to load and align a vector data, etc. See [12, 13] for more details.

On the other hand, vector instruction sets have many fundamental advantages and deserve serious consideration for implementation on microprocessors. Vector ISA packages multiple homogenous, independent operations into a single short instruction, which results in compact, expressive, and scalable code [14-20]. Thus, vector ISA have seen a renaissance, at least for use in graphics, digital signal processing, and multimedia applications, in addition to the traditional scientific and engineering applications [1]. The combination of regularity in each vector instruction and explicit parallelism allows for very aggressive design techniques, such as heavy pipelining, functional unit replication, and aggressive clocking. Recently, CMOS technology has enabled integration of a complete vector processor (scalar core and vector engine with parallel pipelines) on a single chip. Practically, the vector processor developed for the NEC SX-6 supercomputers has eight vector pipelines and a four-way superscalar unit on a single chip (60 million transistors) [21]. Moreover, the latest vector processor of the SX-9 system has 350 million transistors and a peak vector performance exceeding 100 GFLOPS per single core (eight vector pipelines and superscalar unit) [22].

As the underlying semiconductor technology continues to improve significantly since a single chip transistor counts double roughly every 18 months [23], more pipelines and more powerful scalar core can be fabricated on a single chip. On parallel pipelines, not only vector but also matrix data can be processed. This paper proposes a new processor called Mat-Core to exploit the increasingly number of transistors per integrated circuit and improve the performance of many applications on general-purpose processors. Mat-Core extends a general-purpose scalar processor (for executing scalar instructions) with a matrix engine (for executing vector/matrix instructions). One key point of the proposed Mat-Core is the use of multi-level (scalar/vector/matrix) ISA to provide a flexible and high-level interface between hardware and software. High-level instructions, such as scalar-vector, vector-vector, scalar-matrix, vector-matrix, and matrix-matrix instructions, convoy up to 3-D parallelism to the Mat-Core processor, instead of the dynamic extraction of parallelism using complex hardware or the static extraction of parallelism using sophisticated compiler techniques. Another key point of Mat-Core is the use of parallel pipelines for effectively executing both of vector and matrix instructions on the same hardware. Since the fundamental data structures for data parallel applications are scalar, vector, and matrix data [24, 25], which can be executed effectively on Mat-Core, our proposed matrix processor is an appropriate architecture for improving the performance of these applications. This paper describes in detail the organization of a single-core Mat-Core processor. The increasingly budget of transistors can be exploited by providing more processor cores in a physical package. The performance of many applications can be improved on Multi-Mat-Core processors by parallel processing threads of codes using multi-threading techniques [26, 27].

This paper is organized as follows. Section 2 depicts the microarchitecture of the proposed Mat-Core processor, which can execute a mixture of scalar, vector, and matrix instructions and can exploit up to 3-D data parallelism. To tolerate the memory latency, Section 3 describes the architecture of the decoupled Mat-Core processor. The Mat-Core executions of some vector/matrix instructions are explained in Section 4. Moreover, it estimates the performances of element-wise vector-vector addition, vector-matrix multiplication, and matrix-matrix multiplication on Mat-Core processor. Finally, Section 5 concludes this paper and gives directions for future work.

2. THE MICROARCHITECTURE OF THE MAT-CORE PROCESSOR

Figure 1 shows an overall block diagram of a matrix processor, which integrates a scalar processor (instruction cache, scalar functional units, and data cache), an extended unit (matrix unit) for executing high-level vector/matrix instructions, and an external memory interface (address and data buses) for loading/storing data. The scalar processor can be single-issue/multiple-issue, in-order/out-of-order architecture. It is responsible for executing scalar (unparallel) code and for supporting the execution of the high-level instructions on the extended matrix unit. The scalar processor, however, is not responsible for achieving high performance. The extended unit contains the matrix memory unit, matrix arithmetic unit, and matrix register file. The matrix memory unit is used for calculating the addresses and loading/storing the vector and matrix data from/to L2 cache memory to/from the matrix register file. The matrix arithmetic unit executes vector and matrix instructions on data stored in the matrix register file. The extended matrix unit is responsible for achieving high performance by executing vector/matrix instructions.

The straightforward organization of a matrix processor is to structure the extended

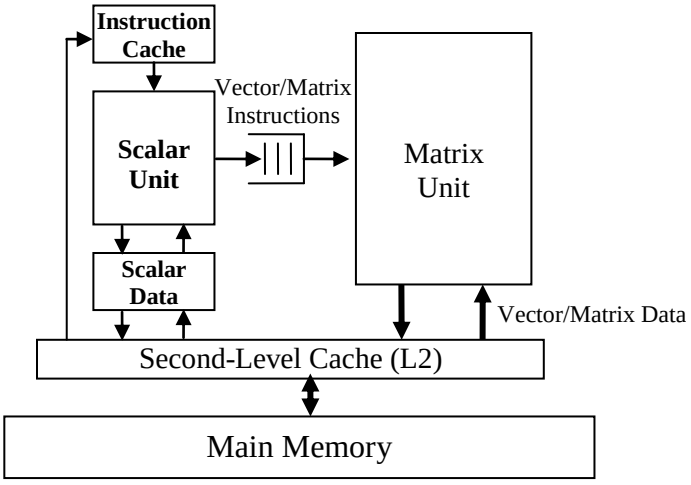

Figure 1: Block diagram of a matrix processor

matrix unit as n^2 cells in order to process $n \times n$ blocks of matrices and n^2-element strips of vectors. Each cell has a portion of the matrix register file, arithmetic functional unit, and load/store unit, as shown in Figure 2. Element-wise vector and matrix instructions can be done easily without intercommunications between the n^2 cells. Each cell loads one element from each input matrix using its load unit, performs the arithmetic instruction using its arithmetic unit, and then stores the result element using its store unit. Said differently, cells (1:n, 1:n) load a block of $A(x_1{:}x_n, y_1{:}y_n)$ into a matrix register ($n \times n$ elements) and a block of $B(x_1{:}x_n, y_1{:}y_n)$ into another matrix register, where (x_1, y_1) and (x_n, y_n) are the indices of the first and last elements of each block. The arithmetic units process these blocks of matrices loaded into matrix registers and then store the result $C(x_1{:}x_n, y_1{:}y_n)$ back to the destination matrix register. The final step is returning the final results stored in the destination matrix register to L2 cache memory by the store units. The same sequence of instructions can be done for vector processing, where the input vectors are loaded into matrix registers in round-robin fashion.

As we can see from Figure 2, each cell has a load/store unit, which is not scalable because of the memory wall problem [28, 29]. Due to the processor-memory performance gap, the main memory cannot sustain this quadratic increase in the number of loaded/stored elements. Let us consider decreasing only the number of load/store units from n^2 to n, where n is the number of cells. In this case, loading an $n \times n$-block of matrix or an n^2-strip of vector requires $O(n)$ time; however, processing these loaded data on n^2 arithmetic units requires only $O(1)$ time. This leads to keeping the arithmetic units idle for a long time while waiting for loading/storing vector/matrix data. In other words, the number of clock cycles needed for loading/storing data dominates the overall computational time. Thus, as the number of load/store units decreases from n^2 to n, the number of arithmetic units should also be decreased to n units, to make a balance between the loading/storing time and processing time. The organization of the matrix unit based on the previous discussion has a 2-D register file, but the load/store and arithmetic units are 1-D.

Processing vector/matrix data on multiple (1-D) execution units requires fetching multiple operands and storing multiple results per a clock cycle. This results in a load

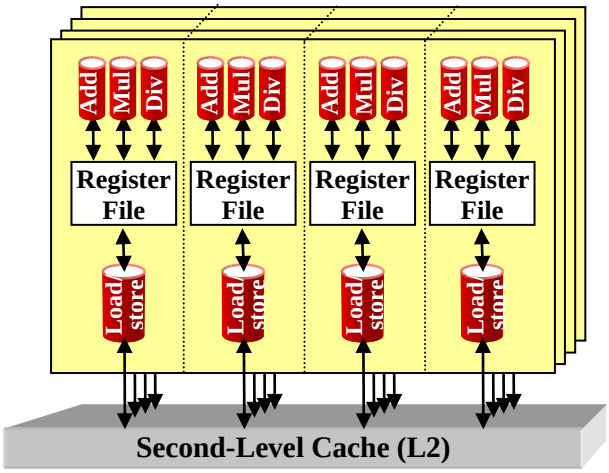

Figure 2: Straightforward organization of a matrix unit

store architecture with a multiple ports register file. A register file with *R* read ports and *W* write ports provides the capability of reading *R* registers and writing *W* registers during the same clock cycle. The most straightforward configuration for implementing a multi-port register file is the monolithic register file [15, 30]. It uses a register cell with multi-read and multi-write ports. Although the number of registers actually accessed is determined by the number of ports, all registers in such a monolithic register file are available simultaneously as a source or a destination for any processing unit or load/store unit. That is why the monolithic register file is also known as a shared register file. This means increasing the number of functional units increases the required number of read and write ports, which results in increasing the monolithic register file area and its time delay. To be specific, for *N* functional units, the area of the monolithic register file grows as N^3 and its delay grows as $N^{3/2}$ [31].

The monolithic register file, which is used by most superpipelined, superscalar, and VLIW designs, provides the ability of any functional unit to access any register randomly. However, in vector and matrix processing, data are accessed from the register file sequentially rather than randomly. In other words, not all registers are needed to be available simultaneously for vector/matrix processing. In addition, since the size of the matrix register file is much greater than the scalar register file, the monolithic register file is not an effective choice because it is not scalable. An alternative configuration for providing a multi-port register file is to partition the registers into banks (partitioned register file) [32, 33]. This configuration is more powerful in vector/matrix processing, where each register bank stores 1-D data. Each bank consists of many multi-port scalar registers and has its own read- and write-buses. Multiple banks give the appearance of a register file with multiple read and write ports needed for vector/matrix processing.

Compared to a monolithic register file, a partitioned one provides less connectivity between any individual register and any functional unit because it uses the features of

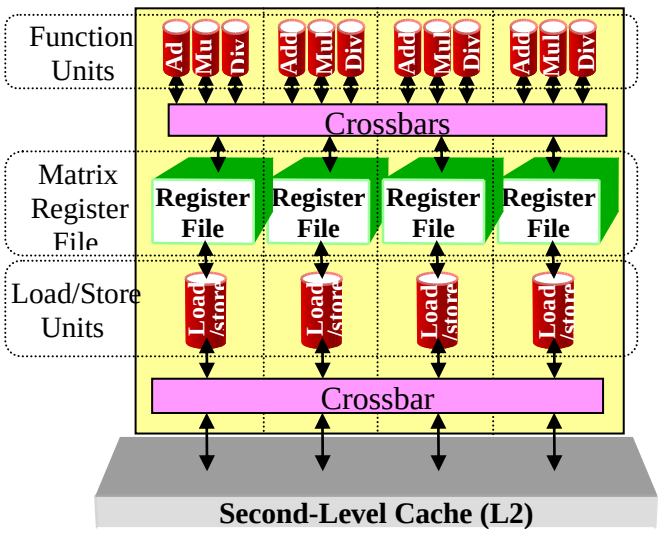

Figure 3: The extended matrix unit of the Mat-Core processor

vector/matrix processing. Obviously, not all elements per register bank are available simultaneously as an operand or as a result. Instead, only three elements per register bank are available (two for reading and one for writing) during each clock cycle. A register bank can be used concurrently at most three instructions as the destination for one and the source of the other two. Figure 3 shows the organization of our proposed matrix unit, which has *P* parallel lanes. Each lane contains a set of register banks based on partitioned register file and a pipeline of each functional unit. *P* register banks represent a matrix register (one register bank per lane), which can store vector/matrix data.

Even though no interconnections between parallel lanes are needed for element-wise vector/matrix instructions, not only element-wise instructions are needed for vector/matrix processing, but reduction and expansion instructions are also needed. Dot-product, vector-matrix, and matrix-matrix multiplication instructions are based on reduction operations; however, outer-product instruction is based on expansion operations. Executing reduction and expansion instructions needs interconnections between lanes. These interconnections can be local, global, bus, etc. It is known that all these types of interconnections are not scalable, except the local, because longer wires are needed to connect more lanes. However, for a small number of parallel lanes, the use of full crossbars is more efficient technique than the other techniques. Crossbars provide complete flexibility in connecting any register bank of the partition register file with any functional unit. Figure 4 shows the operations that can be performed on the crossbars of Mat-Core processor. Pass, Rotate, and Broadcast are the main shuffle operations that can be done on Mat-Core crossbars.

In this paper, we propose a single-core Mat-Core processor, which has a scalar unit

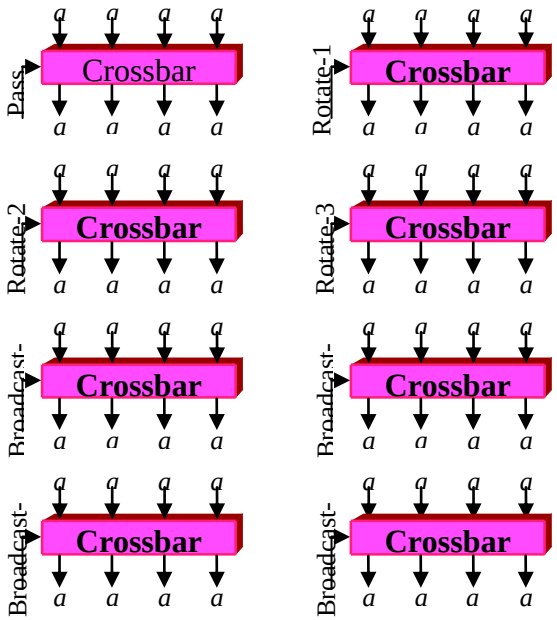

Figure 4: Full connection crossbar

extended by a four-lane matrix unit. As the underlying semiconductor technology continues to improve significantly, more cores can be fabricated on a single chip. Multi-threading techniques on Mat-Core having multi-core (Multi-Mat-Core) can be used to further improving the performance of data parallel applications.

3. DECOUPLED MAT-CORE ARCHITECTURE

Decoupled architectures are based on the observation that the execution of a program can be split into two different tasks: moving data to/from processor and executing arithmetic instructions that perform the program computations [34, 35]. Thus, a decoupled processor has two independent units: the address unit and the computation unit. The address unit performs all address computations, addresses checking, and loads/stores data from/to memory to/from queues in the computation unit. The computation unit moves data from/to queues to/from registers and executes all arithmetic instructions on data loaded into registers. These units are communicated through architectural queues which are used to temporary keep the loaded/stored data from/to memory to/from the register file. The main advantage of decoupled architectures is the toleration of memory latency. The arithmetic instructions waiting for memory operands do not block the issue stage. They are sent to an instruction queue freeing the issue stage to run ahead to find more memory instructions latter in the instruction stream. In other words, latency is tolerated because the address unit is able to slip ahead of the computation unit and loads data that will be needed soon by the computation unit early in time. This excess data produced by the address unit is stored in FIFO queue and stays there until it is retrieved by the computation unit [16].

The Mat-Core processor is based on decoupled architectures to hide memory latency. The extended matrix unit is split into two components: address generation and data computation, which communicate through data queues, as Figure 5 shows. High-level vector/matrix instructions are fetched, decoded, and then dispatched in-order by the scalar core to the pre-address instruction queue (Q1). The matrix unit takes memory/arithmetic vector/matrix instructions in-order from the head of Q1. Without checking, arithmetic instructions are passed directly to the second queue (Q2), which is called address check instruction queue. Load/store instructions are split into two components: address generation and pseudo-move instruction. The first component generates a stream of addresses stored in Q4 (load address queue) or Q5 (store address queue) to fill Q6 (load data queue) or empty Q7 (store data queue), respectively. In more detail, the address generation unit generates and checks the required addresses for loading/storing vector and matrix data. After checking, the address generation inserts load addresses into the load address queue (Q4) and store addresses into the store address

queue (Q5). When either a load or a store is ready (i.e., no dependence and the data is available in case of store instruction), it is sent over the address bus for execution. The pseudo-move instruction moves data from/to the load/store data queue (Q6/Q7) to/from the register files. After being checked, memory instructions are committed in-order from the address check instruction queue (Q2) to the final queue called committed instruction queue (Q3). However, arithmetic instructions are committed directly without checking to the Q3. Once an instruction (arithmetic or pseudo-move) is at the head of the Q3 and its operands are ready, it is dispatched to the appropriate functional unit. Pseudo-move instructions move data from/to Q6/Q7 to/from matrix registers, however, other instructions perform arithmetic operations on data in matrix registers. Note that, the purpose of Q2 is to buffer memory/arithmetic instructions that follow a memory instruction until it is known that the memory instruction will not generate a data page fault. On a page fault, only the content of Q1 and Q2 are needed to be stored.

Mat-Core is a load/store architecture, where memory can be accessed only with load/store instructions (data should be loaded into registers before processing). Scalar data are loaded from scalar data cache into scalar registers (integer or floating-point), processed (in-order or out-of-order) on scalar execution datapath, and then stored from scalar registers back to scalar data cache. Vector/matrix data are loaded directly from L2 cache into matrix registers through load data queue (Q6), processed in parallel on *P* execution datapaths, and then stored back from matrix registers to L2 cache through store data queue (Q7).

Control registers are needed to adjust the number of parallel lanes used to execute vector/matrix instructions and to tell the functional units about the number of elements per lane. *Strps* and *Wstrp* control registers store the number of strips and the number of elements per strip, respectively. *Strps*×*Wstrp* elements of blocks are processed using a vector/matrix instruction. For element-wise vector/matrix instructions, such as element-wise addition, subtraction, multiplication, etc., *Strps* and *Wstrp* are read by the control

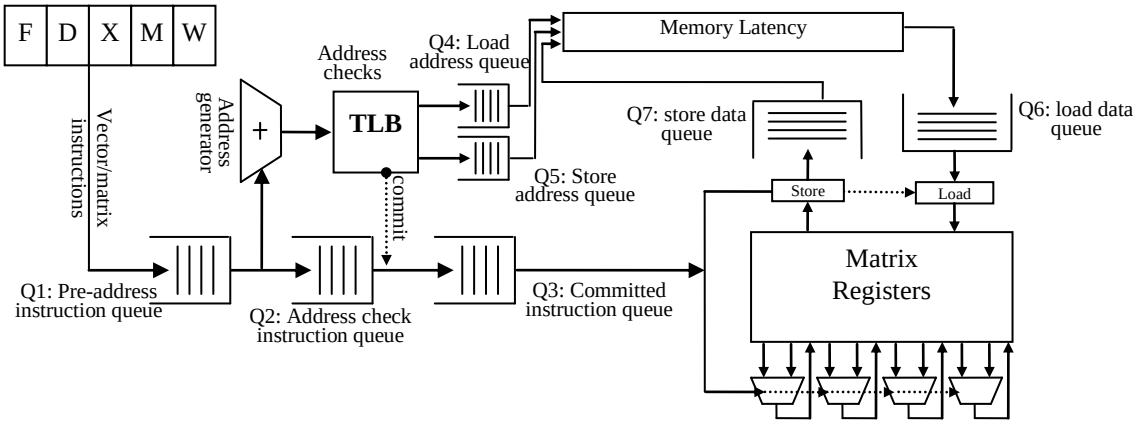

Figure 5: Decoupled Mat-core architecture

unit to generate the proper control signals to process *Strps*×*Wstrp* blocks of matrices or *Strps***Wstrp* strips of elements. Other instructions, such as matrix-matrix multiplications, need three parameters for processing blocks of data. The control register *Dim* is used for storing the third parameter. Depending on the opcode of the instruction being executed, the control unit uses *Strps*/*Wstrp* or *Strps*/*Wstrp*/*Dim* to generate the control signals.

Like vector ISA, memory instructions are divided into separate unit-stride, stride, and indexed classes. The simplest and effective form of loading/storing a block of data is the unit-stride form, which transfers a set of elements ($1 \leq Wstrp \leq P$ elements, where P is the number of lanes) between contiguous memory locations and register file through Q6/Q7. The base address of these *Wstrp* contiguous elements is specified by the contents of a scalar register passed to the matrix unit by the scalar core. The address unit generates a series of memory addresses (only one address per clock cycle); each address moves $1 \leq Wstrp \leq P$ elements from/to L2 cache memory to/from Q6/Q7. On the Mat-Core processor, vector data (1-D arrays) are loaded into matrix registers (2-D arrays) in round-robin fashion, and then processed on P execution datapaths as a matrix data. Said differently, vector data is a special case of matrix data, as explained in more detail in below.

Unit-stride accesses are obviously just a special case of stride accesses. A stride load/store instruction transfers memory elements that are separated by a constant stride. The number of elements between two consecutive elements should be loaded into a scalar register and sent to the matrix unit. Moreover, *Strps*, *Wstrp*, and *Dim* are set to n, 1, and 0, respectively, where n is the vector length.

As unit-stride is a special case from stride memory access, stride load/store is a special case from indexed memory access. Indexed load/store instructions allow elements to be collected into a matrix register from arbitrary locations in memory. An indexed load/store instruction uses another matrix register to supply a set of element indices. For an indexed load or gather, the register of indices is added to a scalar base register to give the effective addresses from which individual elements are gathered. An indexed store, or scatter, inverts the process and scatters elements from a densely packed data from the register file into memory locations specified by the effective addresses.

To effectively support stride and index accesses, the address unit should generate P addresses per clock cycle for loading/storing P elements from/to L2 cache. This means each lane should have an address generator and TLB for generating and checking an address per clock cycle. As the number of parallel lanes increases, the address bandwidth (the number of non-consecutive memory requests that can be transferred per unit time) should also be increased. This results in sophisticated and unscalable architecture because of memory wall problem. For simplicity and scalability, the Mat-Core processor has only a single address port, which can accept a single address per clock cycle. In this case,

strided and indexed loads and stores move at most a single element per clock cycle regardless of operand size.

In addition to *Strps*, *Wstrp*, and *Dim*, Mat-Core has a set of read only control registers: *MSR*, *MSW*, and *MDIM*, which are holding the maximum number of strips, the maximum width per strip, and maximum number of elements for the third parameter, respectively. These control registers are needed for the strip mining technique [36-38] to process longer or unknown vectors, and for the block mining technique based on the block notation technique [24] to process larger or unknown matrices.

4. VECTOR/MATRIX OPERATIONS ON DECOUPLED MAT-CORE PROCESSOR

The Mat-Core ISA extends a scalar ISA with vector and matrix instruction sets. The following instruction sets can be executed on the Mat-Core processor: scalar-scalar (level 0), scalar-vector (level 1), vector-vector (level 1), scalar-matrix (level 2), vector-matrix (level 2), and matrix-matrix (level 3). Up to 3-D data parallelism can be communicated explicitly to the Mat-Core processor through multi-level ISA. This section describes the execution of element-wise vector-vector addition, vector-matrix multiplication, and matrix-matrix multiplication on Mat-Core with 12 clock cycles memory latency and four clock cycles latency for any floating-point operation (FLOP). Besides, the performances of these operations are estimated on the decoupled Mat-Core processor.

4.1. Element-Wise Vector-Vector Addition on Mat-Core

To process vector data on the extended matrix unit of the Mat-Core processor, the input vectors should be loaded into matrix registers in round-robin fashion. After loading vector data into matrix registers, element-wise vector-vector operations can be performed easily like matrix-matrix operations. Let us explain in details the execution of element-wise vector-vector addition on four-lane Mat-Core processor, where the same procedure is done for element-wise matrix-matrix operations. In general, element-wise vector instructions ($Z = X$ op Y), such as vector addition, subtraction, multiplication, division, etc., can be processed on multiple execution datapaths without cross-lane communications (see Figure 6). The input vectors X and Y are distributed across matrix registers in a round-robin fashion. An execution datapath within a lane can process data stored in matrix registers at the rate of one element per cycle. Each datapath receives identical control but different input elements in each clock cycle. Besides, crossbars of source 1 and 2 receive the control signal “Pass” that allows passing the input data of source 1 ($x0$, $x1$, $x2$, and $x3$) and source 2 ($y0$, $y1$, $y2$, and $y3$) without shuffling to the output terminals of the source crossbars.

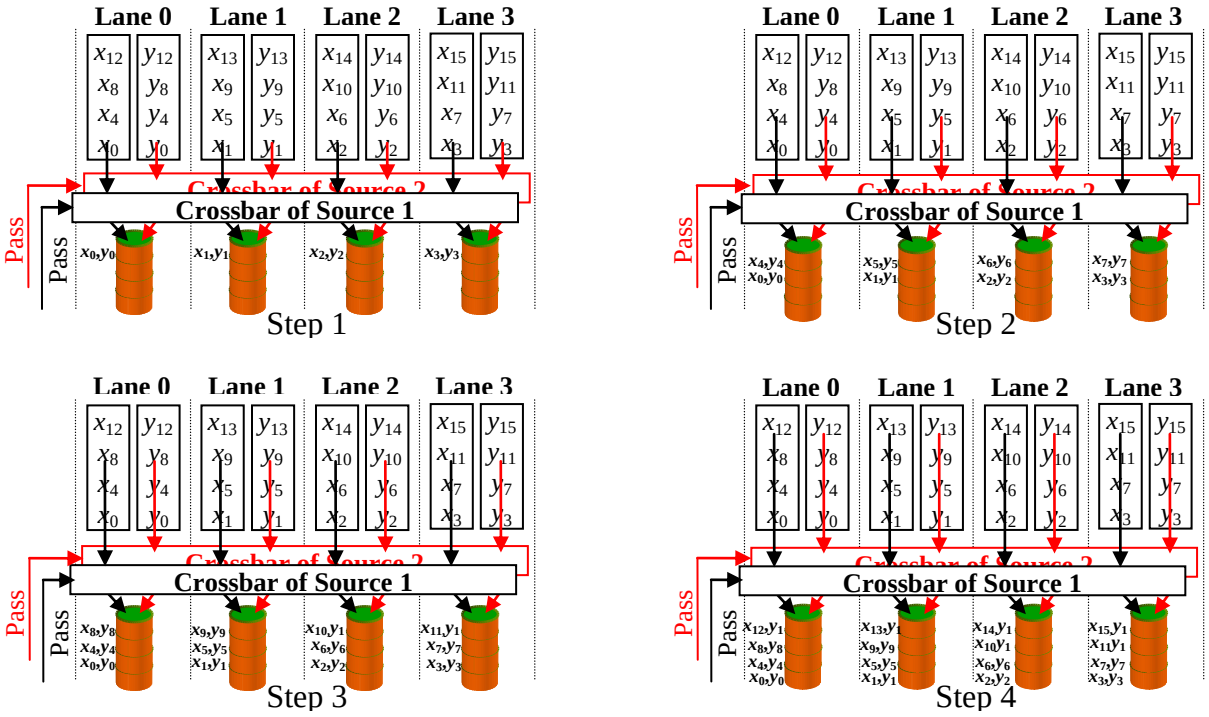

Figure 6: Element-wise vector operation on Mat-Core processor

The performance of element-wise vector-vector addition on the decoupled Mat-Core processor is dominated by memory clock cycles rather than the arithmetic clock cycles. As shown in Figure 7, the computation time can be overlapped by loading/storing time. Thus, on long vectors the number of clock cycles per FLOP is almost three. Besides, Table 1 shows the time line of execution of two iterations of vector addition on decoupled Mat-Core. The scalar unit fetches vector instructions (`LDus`: load with unit-stride, `ADDvv`: add two vectors, and `SRus`: store with unit-stride) and issues them to Q1 (pre-address instruction queue) in the matrix unit. Provided the Q1 is not full the scalar core can continue execution. This results in overlapping the execution time of the scalar instructions with vector instructions because a high-level vector instruction takes many clock cycles for execution.

The matrix unit takes instructions in-order from the head of the Q1. When the head of Q1 is `LDus` or `SRus` (memory) instruction, two actions are done in parallel: sending

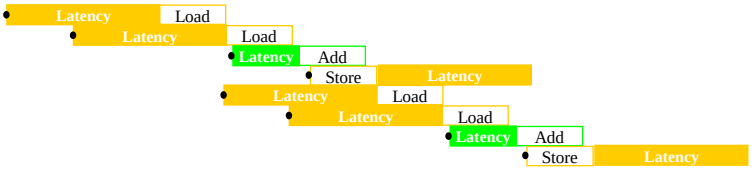

Figure 7: The execution of vector addition

Table 1: Time line of execution of vector addition on decoupled Mat-Core

Instruction	Issue to Q1	Issue to Q2	Issue to Q3	Dispatch from Q3	Complete
`LDus M1,X(R1)`	1	2	6	14	18
`LDus M2,Y(R1)`	2	6	10	18	22
`ADDvv M3,M1,M2`	3	7	10	19	27
`SRus Z(R1),M3`	4	10	14	24	28
`LDus M1,X(R1)`	5	14	18	26	30
`LDus M2,Y(R1)`	6	18	22	30	34
`ADDvv M3,M1,M2`	7	19	22	31	39
`SRus Z(R1),M3`	8	22	26	36	40

the memory instruction to the address unit for generating and checking addresses, and passing the pseudo-move instruction into Q2. However, `ADDvv` instruction is passed to the Q2 without checking when it reaches the head of Q1. The address unit generates, checks, and sends addresses to Q4 (load address queue) or Q5 (store address queue). When an address reaches the head of Q4/Q5, it is sent to the memory for loading/storing data even though the remaining addresses are under generation and checking. The pseudo-move instruction must wait in Q2 until last element address is generated and checked. After checking the last element address, the pseudo-move instruction is allowed to be passed to Q3. Any non memory instructions are at head of the Q2 is simply passed along to the Q3. Finally, arithmetic and pseudo-move instructions in Q3 are executed in-order in the computation unit when all operands are ready. According to Table 1, the performance of element-wise vector-vector addition on long vectors can be estimated as about 1.3 FLOPs per clock cycle (16 FLOPs /12 cycles).

As special case of element-wise vector/matrix operations, scalar-vector and scalar-matrix operations can be easily implemented on Mat-Core processor by performing the same operation on vector/matrix data stored in a matrix register and a scalar datum broadcasted to parallel lanes using crossbar. The crossbar of the source 1 (scalar datum) is controlled by the control signal called “Broadcast-0”; however, the crossbar of source 2 (vector data) is controlled by “Pass” (see Figure 4).

4.2. Vector-Matrix Multiplication on Mat-Core

There are two implementations of the vector-matrix multiplication ($y = x A + y$) due to the two nested loops [24]. One of these implementations is based on dot-product

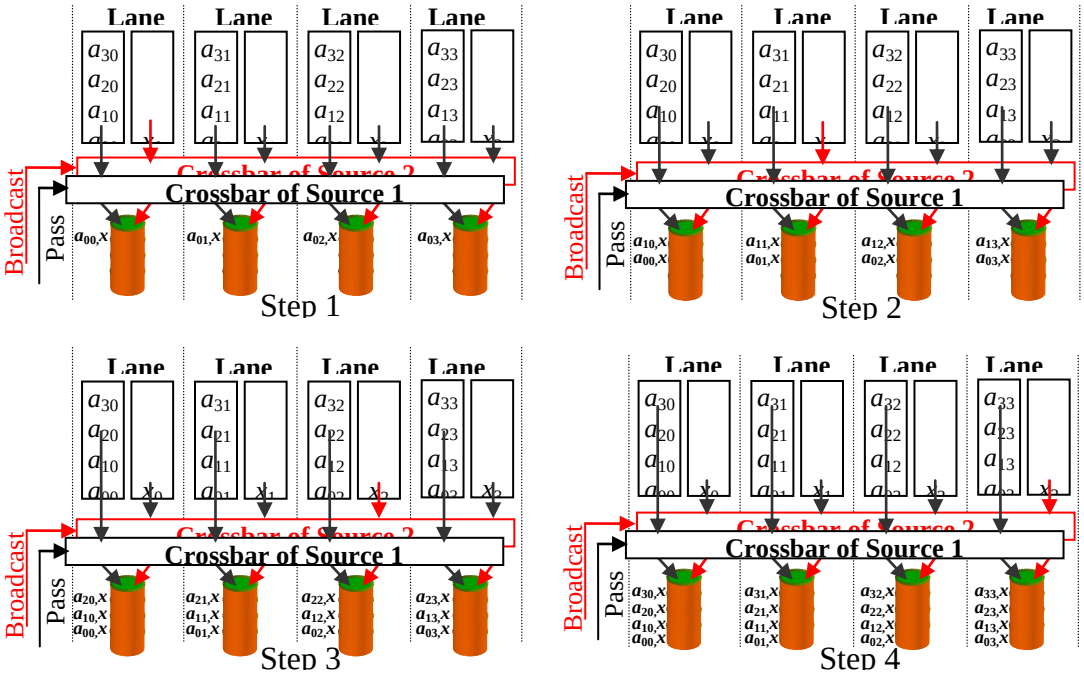

Figure 8: Vector-matrix multiplication on Mat-Core processor

and the other implementation is based on SAXPY (scalar *a* times vector *x* plus vector *y*). Each variant involves the same amount of FLOPs but accesses the operands data differently.

Obviously, loading a block of the input matrix into a matrix register row-by-row is better than loading it column-by-column, assuming the input matrix is stored in the main memory in row major. The former needs unit-stride accesses for loading the input matrix; however, the later needs stride accesses, which is more expensive.

Figure 8 shows the Mat-Core implementation of vector-matrix multiplication based on SAXPY on four parallel lanes. A 4×4 block of the input matrix *A* is loaded into a matrix register say M1 and a 4-element strip of input vector *x* is loaded into another matrix register (the first row of M2; four elements). In addition, a 4-element strip of the vector *y* is loaded into the first row of a matrix register (say M3) for accumulating the result. As shown in Figure 8, the source 2 crossbar broadcasts the elements of the input vector *x*; one element is broadcasted to all lanes each step. Element-wise multiply operations are performed on the contents of the matrix registers M1 and M2. The matrix data stored in M1 are passed through the source 1 crossbar; however, the vector data stored in the first row of M2 are broadcasted through the source 2 crossbar. The result of multiplication is accumulated with the data stored in the matrix register M3. To improve the performance, MAC (multiply-accumulate) operation would be used instead of chaining the multiplier and adder. MAC operation is a one of the fundamental operations

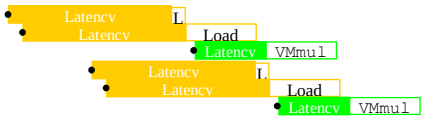

Figure 9: The execution of vector-matrix multiplication

Table 2: Time line of execution of vector-matrix multiplication on decoupled Mat-Core

Instruction	Issue to Q1	Issue to Q2	Issue to Q3	Dispatch from Q3	Complet
`LDus M1,X(R1)`	1	2	3	14	15
`LDus M2,Y(R1)`	2	3	7	15	19
`VMmul M3,M1,M2`	3	4	7	16	24
`LDus M1,X(R1)`	4	7	8	19	20
`LDus M2,Y(R1)`	5	8	12	20	24

of digital signal processors and is a key to dot-product operations for vector and matrix multiplies.

The performance of vector-matrix multiplication on Mat-Core is better than the performance of element-wise vector/matrix operations. As Figure 9 shows, a single loop iteration requires loading five elements and performing four MAC operations (eight FLOPs). However, a single iteration of an element-wise vector/matrix operation requires loading/storing 12 elements and performing only four FLOPs. As shown from the time line of the execution of vector-matrix multiplication on decoupled Mat-Core (see Table 2), the total execution time equals the time needed for loading data. Thus, on large matrices the number of FLOPs per clock cycle is estimated as about 6.4 (32 FLOPs / 5 cycles).

4.3. Matrix-Matrix Multiplication on Mat-Core

Multiplying two $A_{m\times w}$ and $B_{w\times n}$ matrices and accumulating the result with $C_{m\times n}$ matrix ($C = C + A \times B$) can be effectively implemented on the Mat-Core processor as shown in Figure10. The three nested loops (*i*, *j*, *k*) in the matrix-matrix multiplication can be arbitrarily ordered giving six variations (see [24] for more detail). Each of the six possibilities (*ijk*, *jik*, *ikj*, *jki*, *kij*, *kji*) features an inner loop operation (dot-product or SAXPY) and middle loop operation (vector-matrix, matrix-vector, row outer-product, and column outer-product). Additionally, each variant has its own pattern of data accessing (unit-stride or stride), while all of them have the same amount of floating-point operations (2*mwn* FLOPs).

Among the six implementations of the matrix-matrix multiplication, the *ikj* variant, which is based on vector-matrix multiplication and its inner loop is based on SAXPY, is the best because it accesses each block of *A*, *B*, and *C* row-by-row, again assuming *A*, *B*,

and *C* matrices are stored in row major. As Figure 10 shows, the input matrices are loaded into matrix registers (4×4 elements). Sixteen steps are needed to perform four vector-matrix multiplications; each requires four steps.

In contrast of element-wise vector/matrix operations and vector-matrix multiplication, the execution time of matrix-matrix multiplication is dominated by arithmetic (MAC) operations rather than by memory operations. As shown in Figure 11, loading time is overlapped with arithmetic operations. This is good for freeing the memory busses to prefetch more instructions into instruction cache. From the time line of decoupled execution of matrix-matrix multiplication shown in Table 3, the number of FLOPs per clock cycles is estimated as about eight on large matrices (128 FLOPs / 16 clock cycles).

5. CONCLUSION

Different subtasks of an application usually have different computational, memory, and I/O requirements that result in different needs for computer capabilities. Thus, a more appropriate approach for both a high performance and simple programming model is to design a processor having a multi-level instruction set architecture (ISA). Each level has instructions executed on a different data structure and precisely tells the processor what the application needs to be performed in compact form. This leads to high performance and a minimum executable code size.

Data parallel applications, which include scientific, engineering, multimedia, etc., are growing in importance and demanding increased performance from hardware. Since the fundamental data structures for a wide variety of data parallel applications are scalar, vector, and matrix, our proposed Mat-Core processor has a multi-level ISA (scalar-scalar (level-0), scalar-vector (level-1), vector-vector (level-1), vector-matrix (level-2), and matrix-matrix (level-3)) executed on zero-, one-, and two-dimensional arrays of data. These instruction sets are used to express a great amount of fine-grain parallelism (up to 3-D data parallelism) to a processor instead of the dynamical extraction by a complicated logic (superscalar approach) or statically with sophisticated compilers (VLIW approach). This reduces the design complexity and provides a high-level programming interface to hardware.

The proposed Mat-Core processor extends a general-purpose scalar processor (for executing scalar instructions) with a matrix unit (for executing vector/matrix instructions). To tolerate the memory latency, the extended matrix unit is decoupled into two components: address generation and data computation. Like vector microarchitectures, the data computation unit is organized in parallel lanes; each lane

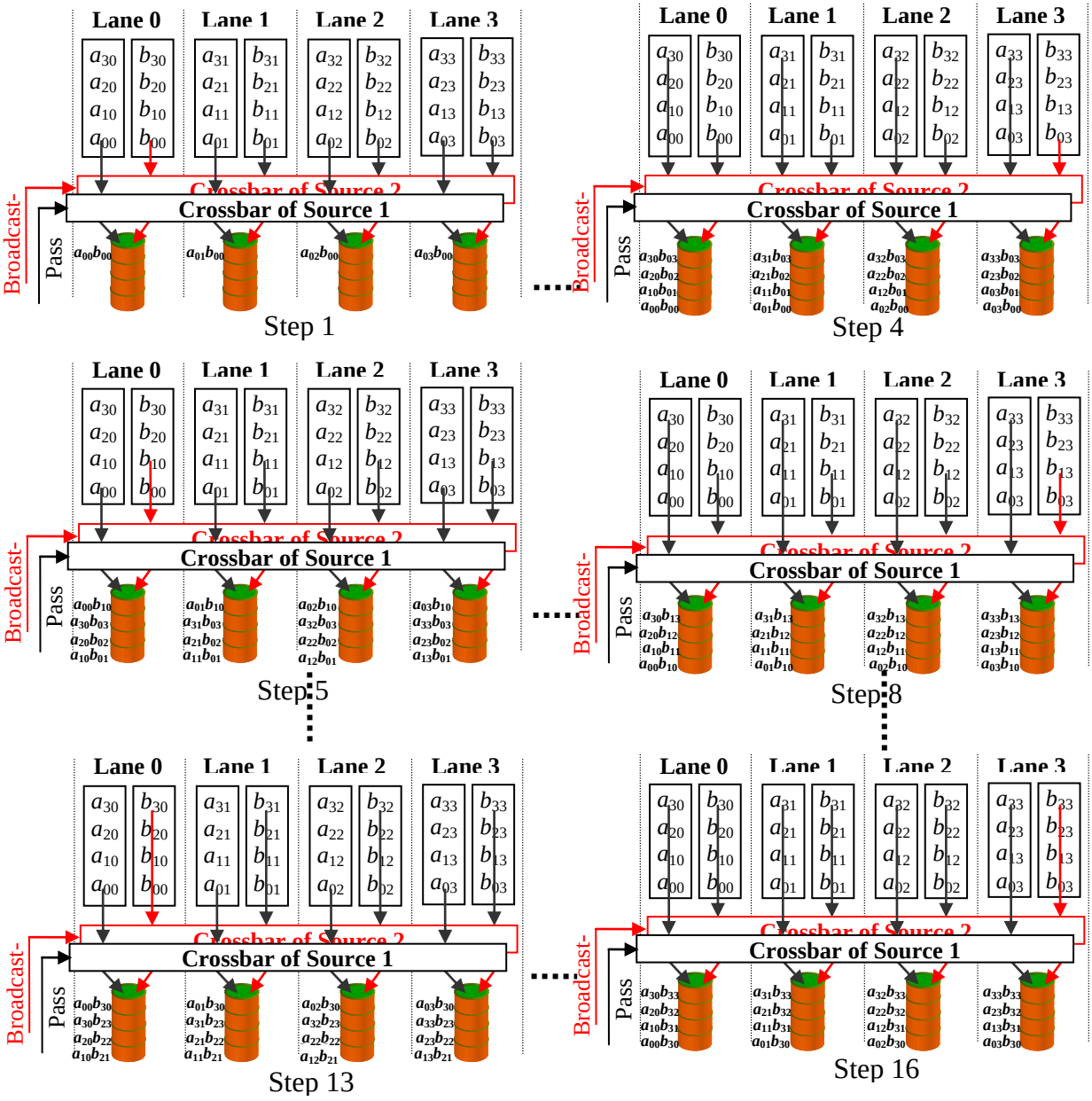

Figure 10: Matrix-matrix multiplication on Mat-Core processor

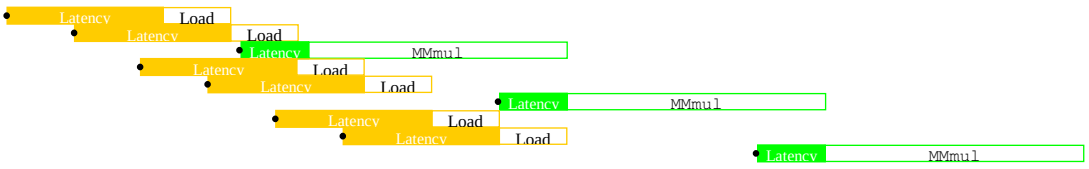

Figure 11: The execution of matrix-matrix multiplication

Table 3: Time line of decoupled execution of matrix-matrix multiplication

Instruction	Issue to Q1	Issue to Q2	Issue to Q3	Dispatch from Q3	Complete
`LDus M1,A(R1)`	1	2	6	14	18
`LDus M2,B(R1)`	2	6	10	18	22
`MMmul M3,M1,M2`	3	7	10	19	39
`LDus M1,A(R1)`	4	10	14	22	26
`LDus M2,B(R1)`	5	14	18	26	30
`MMmul M3,M1,M2`	6	15	18	35	55
`LDus M1,A(R1)`	7	18	22	30	34
`LDus M2,B(R1)`	8	22	26	34	38
`MMmul M3,M1,M2`	9	23	26	51	71

contains a pipeline of each functional unit and a slice of the matrix register file. On parallel lanes, the Mat-Core processor can effectively process not only vector but also matrix data. The executions of element-wise vector-vector addition, vector-matrix multiplication, and matrix-matrix multiplication are explained in detail in this paper. Moreover, their performances are estimated on the decoupled Mat-Core processor as about 1.3, 6.4, and 8 FLOPs per clock cycle, respectively.

As the underlying semiconductor technology continues to improve significantly, the increasingly budget of transistors can be exploited by providing more processor cores in a physical package. The performance of many applications can be improved on the multi-core version of Mat-Core processor (Multi-Mat-Core) by parallel processing threads of codes using multi-threading techniques. In the future, both of Mat-Core and Multi-Mat-Core will be implemented and evaluated on kernels and whole applications.

REFERENCES

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, 4th Edition, ISBN 1558605967, 2007.

[2] J. Ahn, M. Erez, and W. Dally, "Tradeoff between Data-, Instruction-, and Thread-level Parallelism in Stream Processors" *Proc. 21st ACM International Conference on Supercomputing (ICS'07)*, pp. 126-137, 2007.

[3] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Order Number: 248966-020, Available at www.intel.com/products/processor/manuals/index.htm, November 2009.

[4] J. Smith and G. Sohi, "The Microarchitecture of Superscalar Processors," *Proceedings of the IEEE*, Vol. 83, No. 12, pp. 1609-24, December 1995.

[5] J. Fisher, "VLIW Architectures and the ELI-512," *Proc. 10th International Symposium on Computer Architecture*, Stockholm, Sweden, pp. 140-150, June 1983.

[6] W. Hillis and G. Steele, "Data Parallel Algorithms," *Communications of the ACM*, Vol. 29, No. 12, pp. 1170-1183, December 1986.

[7] K. Sankaralingam, S. Keckler, W. Mark, and D. Burger, "Universal Mechanisms for Data-Parallel Architectures," *36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, San Diego, California, pp. 303-314, December 2003.

[8] C. Lee and D. DeVries, "Initial Results on the Performance and Cost of Vector Microprocessors," *Proc. 30th Annual International Symposium on Microarchitecture*, December 1997.

[9] K. Diefendorff and *P*. Dubey, "How Multimedia Workloads Will Change Processor Design," *IEEE Computer*, Vol. 30, No. 9, pp. 43-45, September 1997.

[10] N. Slingerland and A. Smith, "Multimedia Extensions for General Purpose Microprocessors: A Survey" *Microprocessors and Microsystems*, Vol. 29, Issue 5, pp. 225-246, June 2005.

[11] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Vol. 1: Basic Architecture, Order Number: 253665-033US, Available at www.intel.com/products/processor/manuals/index.htm, December 2009.

[12] M. Stoodley and C. Lee, "Vector Microprocessors for Desktop Computing," *Proc. 26th Annual International Symposium on Computer Architecture*, 1999.

[13] C. Lantwin, "NEC at the SC98 Conference: The 10 Past and the 10 Future Years of HPC," *SX WORLD*, No. 24, Spring 1999.

[14] W. Schonauer, *Scientific Computing on Vector Computers*, North-Holland, Amsterdam, 1987.

[15] C. Lee, *Code Optimizers and Register Organizations for Vector Architectures*, Ph.D Thesis, Computer Science Division, University of California at Berkeley, 1992.

[16] R. Espasa, *Advanced Vector Architectures*, Ph.D. Thesis, Department of Computer Architecture, Universitat Politecnica de Catalunya, Barcelona, Spain, February 1997.

[17] K. Asanovic, *Vector Microprocessors*, Ph.D. Thesis, Computer Science Division, University of California at Berkeley, 1998.

[18] C. Kozyrakis, *Scalable Vector Media-processors for Embedded Systems*, Ph.D. Thesis, Computer Science Division, University of California at Berkeley, 2002.

[19] R. Krashinsky, *Vector-Thread Architecture And Implementation*, Ph.D. Thesis, Massachusetts Institute Of Technology, 2007.

[20] J. Gebis, *Low-complexity Vector Microprocessor Extensions*, Ph.D. thesis, University of California at Berkeley, 2008.

[21] K. Kitagawa, S. Tagaya, Y. Hagihara, and Y. Kanoh, "A Hardware Overview of SX-6 and SX-7 Supercomputer," *NEC Research & Development*, Vol. 44, No. 1, pp. 2-7, January 2003.

[22] A. Musa, *High Performance Memory Architecture for Vector Processors*, Ph.D Thesis, Department of Computer and Mathematical Sciences, Graduate School of Information Sciences, Tohoku University, January 2009.

[23] G. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, Vol. 38, No. 8, pp. 114-117, April 1965.

[24] G. Golub and C. Van Loan, *Matrix Computations*. John Hopkins University Press, Baltimore and London, 3rd Edition, 1996.

[25] J. Dongarra, I. Foster, G. Fox, K. Kennedy, A. White, L. Torczon, and W. Gropp, *The Sourcebook of Parallel Computing*, Morgan Kaufmann, ISBN 1558608710, November 2002.

[26] A. Binstock and R. Gerber, *Programming with Hyper-Threading Technology: How to Write Multithreaded Software for Intel IA-32 Processors*, Intel PRESS, ISBN 0970284691, 2003.

[27] S. Akhter and J. RobertsIntel, *Multi-Core Programming: Increasing Performance through Software Multithreading*, Intel PRESS, ISBN 0976483246, 2006.

[28] W. Wulf and S. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, Vol. 23, No. 1, pp. 20-24, March 1995.

[29] P. Machanick, "Approaches to Addressing the Memory Wall," *Technical Report*, No. 6, The University of Queensland, Australia, November 2002.

[30] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Reducing the Complexity of the Register File in Dynamic Superscalar Processors," *Proc. 34th Annual International Symposium on Microarchitecture (MICRO'01)*, Austin, Texas, December 2001.

[31] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens, "Register Organization for Media Processing," *Proc. 6th International Symposium on High-Performance Computer Architecture (HPCA 6)*, pp. 375-386, 2000.

[32] C. Lee, and J. Smith, "A Study of Partitioned Vector Register Files," *Proc. Supercomputing'92*, Minneapolis, MN, USA, December 1992.

[33] J. Tseng and K. Asanovic, "Banked Multiported Register Files for High-Frequency Superscalar Microprocessors," *Proc. 30th Annual International Symposium on Computer Architecture*, San Diego, California, pp. 62-71, June 2003.

[34] J. Smith, "Decoupled Access/Execute Computer Architectures," *ACM Transactions on Computer Systems*, Vol. 2, No. 4, pp. 289-308, November 1984.

[35] W. Ro, S. Crago, A. Despain, and J. Gaudiot, "Design and Evaluation of a Hierarchical Decoupled Architecture," *The Journal of Supercomputing*, Vol. 38, Issue 3, pp. 237-259, December 2006.

[36] M. Weiss, "Strip Mining on SIMD Architectures," *Proc. 5th International Conference on Supercomputing*, Cologne, West Germany, pp. 234-243, June 1991.

[37] D. Bacon, S. Graham, and O. Sharp, "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys*, Vol. 26, No. 4, pp. 345-420, December 1994.

[38] D. DeVries, *A Vectorizing SUIF Compiler: Implementation and Performance*, Master Thesis, Department of Electrical and Computer Engineering, University of Toronto, June 1997.