Neural, Parallel, and Scientific Computations 18 (2010) 59 – 74

Codevelopment of Multi-Level Instruction Set Architecture and Hardware for an Efficient Matrix Processor

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

Abdulmajid F. Al-Junaid
Computer & System Section, Electrical Engineering Department, Faculty of Engineering, Assiut University, Egypt

Abstract
The instruction set architecture (ISA) is the part of the processor that is visible to the programmer or compiler writer. Multi-level ISA is proposed to explicitly communicate data parallelism to hardware (processor) in a compact way instead of the dynamic extraction using complex hardware or the static extraction using sophisticated compiler techniques. This paper presents the codevelopment of multi-level ISA and hardware for an efficient matrix processor called Mat-Core. Mat-Core extends a general-purpose scalar processor with a matrix unit for processing vector/matrix data. To hide memory latency, the extended matrix unit is decoupled into two components: address generation and data computation, which communicate through data queues. Like vector architectures, the data computation unit is organized in parallel lanes. However, on parallel lanes, Mat-Core can execute scalar-matrix, vector-matrix, and matrix-matrix instructions in addition to scalar-vector and vector-vector instructions. Mat-Core leads to a compiler model that is efficient both in terms of performance and executable code size. On four parallel lanes Mat-Core and matrix registers of size 8×4 or 32 elements, our results show performances of about 1.6, 2.1, 4.1, and 6.4 FLOPs per clock cycle achieved on scalar-vector multiplication, SAXPY, vector-matrix multiplication, and matrix-matrix multiplication, respectively.
Keywords - high performance computing, multi-level ISA, performance evaluation, SystemC implementation, vector/matrix processing.

1. INTRODUCTION

The presence of parallelism in applications is the key to achieving high performance with all modern microprocessors, for it allows the hardware to accelerate applications by executing multiple, independent operations concurrently [1]. Beyond simple pipelining, there are three major forms of parallelism, instruction-level parallelism (ILP), thread-level parallelism (TLP), and data-level parallelism (DLP), which are not mutually exclusive. The cheapest and the most prevalent form of parallelism available in many

Received January 11, 2010

applications is DLP. For example, in multimedia applications, which are widely used today, the computationally intensive kernels repeat the same set of operations over streams of input data [2].

The Mat-Core architecture [3] relies on the use of multi-level ISA to express and exploit the DLP in data parallel applications. Scalar/vector/matrix instruction sets can be executed on the Mat-Core hardware (see Figure 1). Scalar instruction set architecture (Level-1 ISA) is the fundamental instruction set for any general-purpose processor. It usually has an instruction for every scalar operation (see Figure 1a). Meaning that scalar ISA cannot express parallelism to hardware (processor); however, the performance can be improved only by processing more scalar instructions concurrently. These parallel instructions require complex superscalar architectures [4] for extracting them dynamically or wide VLIW architectures [5] relying on sophisticated compiler for extracting them statically.

Vector instruction sets (Level-2 ISA) have many fundamental advantages and deserve serious consideration for implementation on microprocessors [6-9]. Vector ISA packages multiple homogenous, independent operations into a single short instruction, which results in compact, expressive, and scalable code (see Figure 1b). Advantages of vector ISAs over scalar or VLIW ISAs can be placed in three broad categories [10]. First, semantic advantages; that is, vector ISAs tend to express programs in a more concise and efficient way. Second, explicit parallelism is encoded in each vector instruction, thus allowing for highly parallel implementations. Third, 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 [1].

To reduce the execution time, most vector processors use parallel pipelines per functional unit. On parallel pipelines, not only vector but also matrix data can be processed (see Figure 1c). Matrix instruction set (Level-3 ISA) can be executed on the same parallel pipelines of the extended matrix unit of Mat-Core processor. Matrix FastCrypto: Parallel

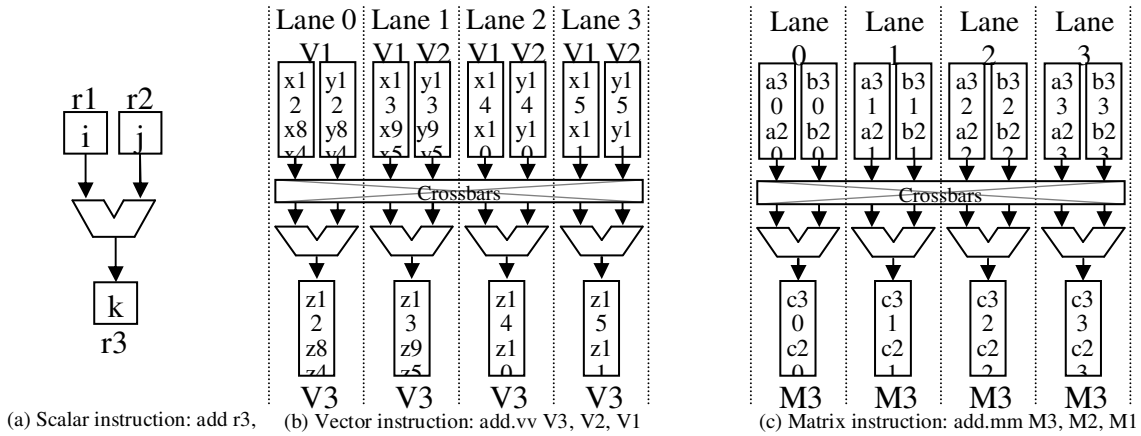

Figure 1: Mat-Core instruction sets

processing is the logical direct extension of vector processing. Matrix ISA further reduces both the semantic and the parallelism gaps between high-level languages and hardware. (See [11] for more detail about the semantic and the parallelism gaps.) Thus, high-level instructions, such as vector-scalar, vector-vector, matrix-scalar, matrix-vector, and matrix-matrix instructions, convoy up to 3-D data parallelism to Mat-Core processor, which results in reducing the complexity of hardware and compiler.

As Figure 2 shows, 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. The elements of vector data are distributed across the lanes in a round-robin, interleaved fashion (see Figure 1b). SystemC has been used to simulate the Mat-Core processor (see [12] for more detail).

There are several benefits to the modular, lane-based implementation [13]. A single lane must be designed and verified regardless of the number of lanes allocated in the processor. Scaling the processor for processing longer vectors or larger matrices by allocating the proper number of lanes leads to balanced addition of both register file and execution resources, without requiring redesign of functional units or their control. A four-lane processor, for example, can store vectors twice as long and execute twice as many element operations per cycle as a two-lane processor. Finally, the locality of communication in a lane-based processors, allows hardware scaling without implications due to the high latency of long, cross-chip wires [14-15]. Note that the use of crossbars across Mat-Core lanes reduces the scalability. Thus, small number of parallel lanes is used for processing vector/matrix data per a single core of Mat-Core processor. As the underlying semiconductor technology continues to improve significantly, more cores can

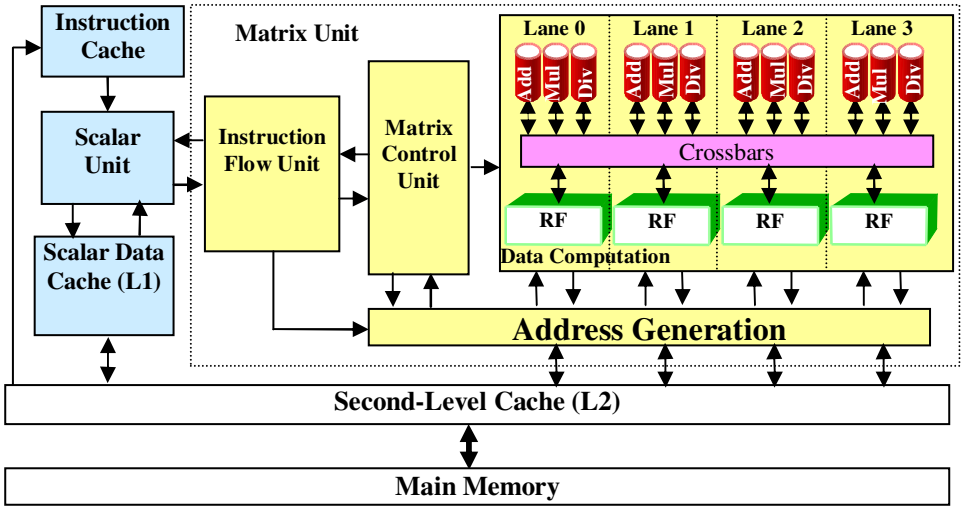

Figure 2: Block diagram of the Mat-Core processor

be fabricated on a single chip. Multi-threading techniques on Mat-Core having multi-core can be used to further scaling Mat-Core and improving the performance of data parallel applications.

This paper is organized as follows. Section 2 demonstrates the Mat-Core instruction sets. It presents the codevelopment of multi-level ISA and hardware for an efficient matrix processor (Mat-Core). Section 3 describes the architecture of the decoupled Mat-Core, which can hide memory latency by splitting extended matrix unit into address generation and data computation units. In Section 4 an assembler for Mat-Core architecture is demonstrated, which enables writing programs in assembly language instead of machine code. Moreover, Section 4 presents the performance evaluation of Mat-Core processor on vector kernels (scalar-vector multiplication and SAXPY: Single-precision scalar A times vector X Plus vector Y), on matrix-vector kernel (vector-matrix multiplication), and on matrix-matrix multiplication as a matrix kernel. Finally, Section 5 concludes this paper.

2. MAT-CORE INSTRUCTION SETS

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 (LDQ), processed on parallel execution datapaths, and then stored back from matrix registers to L2 cache through store data queue (SDQ).

According to Mat-Core ISA, matrix unit can contain up to 32 matrix registers, each can store up to $1024 \times P$ elements, where P is the number of parallel lanes. Each element can be 64-bit data or less. These matrix registers can store strips of vector data or blocks of matrices (see Figure 1b and 1c). Thus, each matrix register has two names (M0 through M31 or V0 through V31). For example, M7 and V7 refer to the same hardware register. This will make the code more readable for matrix/vector processing. The first implementation of Mat-Core provides only 8 matrix registers M0 through M7 or V0 through V7. Each of them has 32 elements (8×4) of 32-bit wide. Besides, the number of parallel lanes of the first implementation is only four ($P = 4$). Future implementations may provide larger number of lanes for processing longer strip vectors and larger block matrices.

Mat-Core instructions are divided into several classes:

- Scalar/vector/matrix load/store instructions that move 0-D/1-D/2-D data between registers and memory;

- Scalar/vector/matrix arithmetic/logical instructions that process 0-D/1-D/2-D data stored in Mat-Core registers;
- Compare/branch/jump scalar instructions.
- Control instructions that read/write control registers; and
- Move instructions, which move data between matrix registers, and between scalar registers and matrix registers.

Before executing high-level vector/matrix instruction(s), CPL (control parallel lanes) instruction should be executed first. The main purpose of the CPL control instruction is to adjust the number of parallel lanes used in the successive execution of vector/matrix instruction(s) and to tell the functional units about the number of elements per lane. Like any Mat-Core instruction, CPL has 32-bit with the following fields.

6-bit	10-bit	5-bit	5-bit	6-bit
Matrix class	*Strps*	*Wstrp*	*Dim*	CPL opcode

The first 6-bit (010010 like the code of coprocessor 2 in MIPS architectures) tells the scalar part that the execution of this instruction is on the extended matrix unit instead of on the scalar unit. *Strps* and *Wstrp* fields store the number of strips and the number of elements per strip, respectively, where $1 \leq Strps \leq 1024$ and $1 \leq Wstrp \leq 32$. The last 6-bit (000000) is the opcode of CPL control instruction Thus the second and the third fields are used to set the control registers *Strps* and *Wstrp*, 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 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* fields 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 LDQ/SDQ. The base address of all 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 LDQ/SDQ. Using *Strps* and *Wstrp*, a single-precision matrix block can be loaded into *Md* matrix register (`LM.S Md, rs, rt`), as the following pseudo code shows, where *rs* and *rt* are scalar registers holding the starting address and the number of bytes between two consecutive rows,

respectively. Note that 96-bit (`LM.S` instruction, *rs*, and *rt* scalar registers) are sent from scalar unit to matrix unit by the end of decode stage of the scalar pipeline.

```
address = rs;
for(i=0; i<Strps; i++){
  for(j=0; j<Wstrp; j++){
    Md[i][j] = Mem[address+j*4]
  }
  address += rt;
}
```

In addition, the format of the Mat-Core load instruction (32-bit) is as follows.

6-bit	5-bits	5-bits	5-bits	5-bits	6-bit
010010	rs	rt	Md	LM.S	load/store

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 (see Figure 1b). Said differently, vector data is a special case of matrix data. The CPL control instruction for processing *n*-element vector data is as follows.

6-bit	10-bit	5-bit	5-bit	6-bit
Matrix (010010)	*Strps* ($\lfloor n/P \rfloor$)	*Wstrp* (*P*)	*Dim* (00000)	CPL (000000)

The Mat-Core instruction format for loading *n*-element (unit-stride loading), four-byte each, into a matrix register is as follows.

6-bit	5-bits	5-bits	5-bits	5-bits	6-bit
Matrix (010010)	rs (base add.)	rt (Wstrp*4)	Md (dest. reg.)	LM.S	Load/store (000001)

Not only unit-stride vector load/store, but also stride version is available. We can think of the matrix block load as a stride vector load. The contents of CPL instruction for loading/storing stride data are as follows.

6-bit	10-bit	5-bit	5-bit	6-bit
Matrix (010010)	*Strps* (*n*)	*Wstrp* (00001)	*Dim* (00000)	CPL (000000)

The first lane is used for loading/storing stride data into a matrix register (`LM.S Md, rs, rt`), where *rt* stores the stride value and *rs* stores the address of the first element. The first version of Mat-Core does not support indexed addressing mode.

On 8×4 matrix registers ($1 \le Strps \le 8$ and $1 \le Wstrp \le 4$), the Mat-Core processor can process strips of vectors with a maximum length of 32 elements or blocks of matrices with size of 8×4. These maximum strip length and maximum block size are unlikely to match the real vector length and matrix size in a program, respectively. Moreover, the size of a particular vector or matrix data is often unknown until run time in real programs. To solve these problems both the strip mining technique [16-18] to process longer or unknown vectors, and the block mining based on the block notation technique [19] to process larger or unknown matrices, are used (see Figure 3).

In the strip mining technique, the compiler reads two read-only control registers called *MSR* and *MSW*, which holding the maximum number of strips and the maximum number width per strip, respectively. The compiler strips an *n*-elemet vectors into $\lceil n/(MSR*MSW) \rceil$ segments. The length of all segments is *MSR*MSW* except the last which has a length of the remainder of dividing *n* by *MSR*MSW*. This results in reducing the of iterations of a single loop from *n* (by using scalar ISA) to $\lceil n/(MSR*MSW) \rceil$ (by using vector ISA).

The logical extension of the strip mining for processing longer vectors is the block mining for processing larger matrices (*m*×*n* elements). The block mining technique is based on the block notation of existing matrix algorithms. The compiler reads the control registers *MSR* and *MSW* as well as *MDIM*, which holds the maximum number of elements for the third parameter needed for matrix-matrix multiplication. The compiler reads the necessarily control registers and strips the given *m*×*n* matrices horizontally into $\lceil m/MSR \rceil$ segments and vertically into $\lceil n/MSW \rceil$ segments for element-wise matrix instructions. In case of matrix-matrix multiplication (*m*×*n* times *n*×*w*), the compiler strips the first matrix horizontally into $\lceil m/MDIM \rceil$ segments and vertically into $\lceil n/MSR \rceil$ segments and the second matrix horizontally into $\lceil n/MSR \rceil$ segments and vertically into

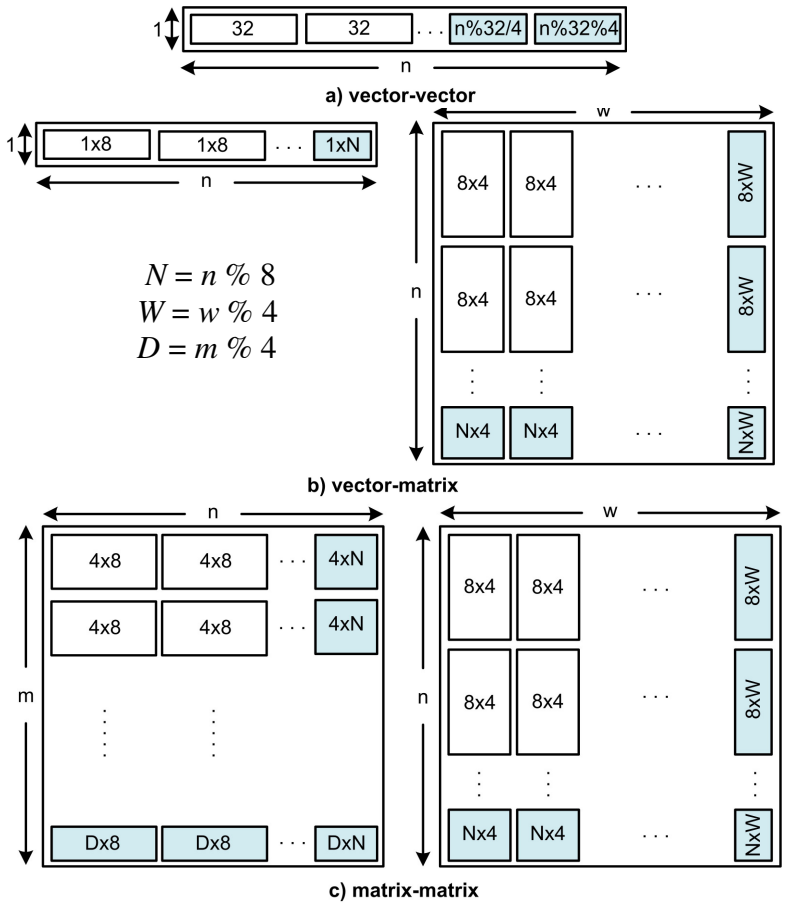

Figure 3: Strip/block mining for processing larger vectors/matrices

$\lceil w/MSW \rceil$ segments. Note that the *Strps*, *Wstrp*, and *Dim* control registers are set to *MSR*, *MSW*, and *MDIM*, respectively, to process the maximum vector strips or the maximum matrix blocks. To process shorter vector strips or smaller matrix blocks, these control registers are set to values less than *MSR*, *MSW*, and *MDIM*.

3. THE ARCHITECTURE OF THE DECOUPLED MAT-CORE

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 [20, 21]. The main advantage of decoupled architectures is the toleration of memory latency. In decoupled architectures, 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 [7].

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 4 shows. The SystemC implementation of the decoupled Mat-Core processor is described in detail in [12]. The address unit performs all address computations, addresses checking, and loads/stores data from/to memory to/from queues. 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.

High-level vector/matrix instructions are fetched, decoded, and then dispatched in-order by the scalar core to the pre-address queue called instruction and scalar operands queue 1 (ISQ1). Instruction flow controller in the matrix part takes memory/arithmetic vector/matrix instructions in-order from the head of ISQ1. Load/store instructions are split into two components: address generation and pseudo-move instruction. The first component generates a stream of addresses stored in LAQ (load address queue) or SAQ (store address queue) to fill LDQ (load data queue) or empty SDQ (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 unit inserts load addresses into the load address queue (LAQ) and store addresses into the store address queue (SAQ). 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 (LDQ/SDQ) to/from the matrix register file.

Arithmetic and pseudo-move instructions are passed to another queue called

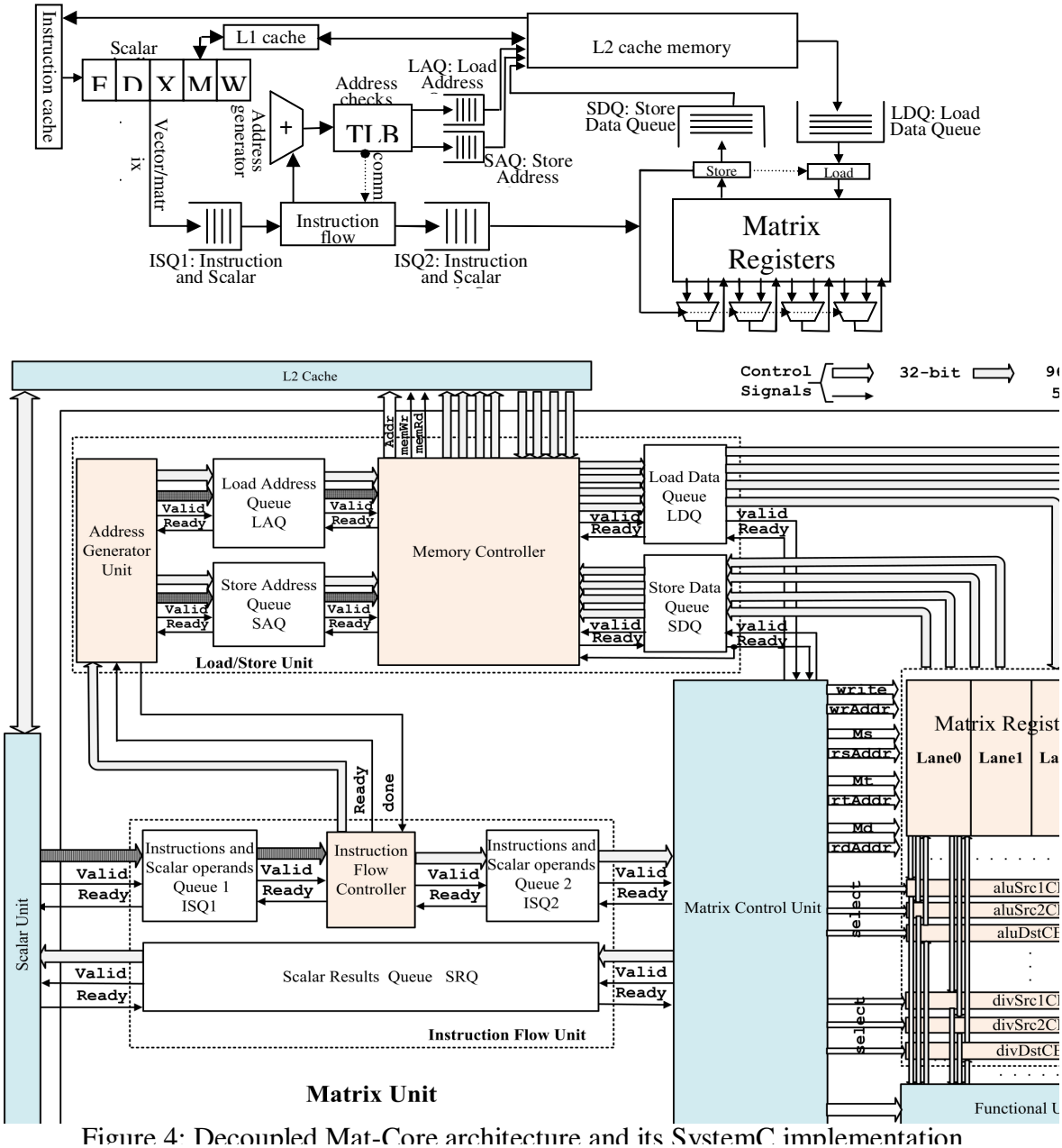

Figure 4: Decoupled Mat-Core architecture and its SystemC implementation

instruction and scalar operands queue 2 (ISQ2). Note that, the contents of ISQ1 is arithmetic/memory instructions, however, ISQ2 is keeping arithmetic and pseudo-move instructions. Once a pseudo-move is at the head of the ISQ2 and its operands are ready, the scoreboard control unit moves operands from/to LDQ/SDQ to/from matrix registers. Pseudo-move instructions move data from/to LDQ/SDQ to/from matrix registers,

however, other instructions perform arithmetic operations on data in matrix registers. The purpose of ISQ2 is to buffer pseudo-move/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, the contents of ISQ1, ISQ2, and the current instruction in instruction flow controller are needed to be stored.

No interconnections between parallel lanes are needed for element-wise vector/matrix instructions. However, 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 multiplications are based on reduction operations; however, outer-product 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. Pass, Rotate, and Broadcast are the main shuffle operations that can be done on Mat-Core crossbars. See [3] for more detail about using crossbars in the execution of matrix/vector operations.

4. ASSEMBLER FOR MAT-CORE ARCHITECTURE

For Mat-Core hardware to be useful, it is necessary to be able to compile applications, written in high-level languages such as MATLAB, MATHEMATICA, C++, etc., into sequences of scalar/vector/matrix instructions. Since MATLAB supports matrix notations, it can be considered as a suitable high-level language for programming Mat-Core applications. Traditionally, the Mat-Core compiler can be divided into three main stages. The first stage is the compiler front-end where syntax and semantic analysis are done. Loop blocking, strip/block mining, and independent optimization are the three main tasks of the second stage. The third stage is the compiler back-end, which generates machine code for Mat-Core architecture and schedules the generated code for optimal performance. It is known that writing such compiler is not an easy task, which represents our future work. In this section, a part of the last stage (compiler back-end), is demonstrated as an assembler for Mat-Core. This enables us to write programs in assembly language instead of machine code. Thus, the performance of Mat-Core is evaluated now on kernels instead of applications.

The Mat-Core assembler consists of three passes as shown in Figure 5. In the first pass (macro manipulation), each calling of a macro is replaced by its equivalent body.

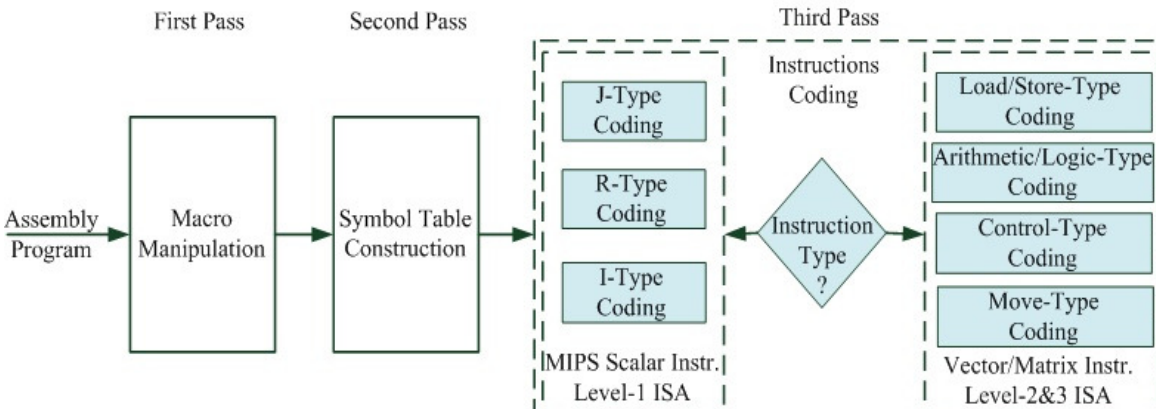

Figure 5: Three-pass Mat-Core assembler for converting assembly program to machine code

Macro is a group of instructions performs a task that is used repeatedly. Macros allow the programmer to write the task once and to invoke it whenever it is needed and wherever it is needed. To make macros more flexible, parameters are used. The assembler passes macro parameters to the macro body for substitution. Since a macro can be expanded more than once in a program, labels of the body of the macro must be renamed to avoid label replication. Otherwise an assembler error would be generated when the same label is encounter in two or more places.

The second pass is the symbol table construction for labels and addresses. A symbol table is a container that maps each label in the source program to its corresponding address in memory. Labels in assembly programs sometimes represent an address in the data area (a variable) and sometimes they represent an address in the program area (a location to which you want to jump). Symbol tables are typically implemented using hashing schemes because good efficiency for the lookup is needed. Thus the symbol table of the Mat-Core assembler is constructed as a hash table. Finally, the third stage (instructions coding) generates machine code depending on the type of instruction and the content of symbol table. The coding process includes two main processes: coding Level-1 ISA (MIPS scalar ISA) and coding high-level instructions (vector/matrix instructions). The vector/matrix instructions are coded into load/store, arithmetic/logic, control, and move types as discussed in Section 2. The same conventions used in MIPS assemblers like directives, register names, etc., are used in developing Mat-Core assembler.

As mentioned in Section 2, strip/block mining techniques are used for processing longer vectors and larger matrices. In this section, we show how these techniques are applied on SVmul, SAXPY, vector-matrix and matrix-matrix multiplications on four

parallel lanes, matrix registers of 8×4 elements, and the contents of *MSR*, *MSW*, and *MDIM* registers are 8, 4, and 4, respectively. Note that the CPL instruction is used to set

Table 1: Mat-Core instructions in high-level and assembly languages

Instruction set	High-Level Statement in Matlab	Mat-core assembly code for inner loop only

Scalar-vector multiplication	c(1:n) = a* b(1:n)	`LV.S $M4,$t0,$t #load vector of SP elements`
		`MULVS.S $M5,$M4,$t2 # vector-scalar multiply`
		`SV.S $M5,$t4,$t1 #store result vector`

SAXPY	c(1:n) = a*b(1:n) + c(1:n)	`LV.S $M3,$t0,$t1 #load vector`
		`LV.S $M4,$t6,$t1 #load vector`
		`MULVS.S $M5,$M3,$t4 #vector-scalar multiply`
		`ADDVV.S $M7,$M5,$M4 #add two vectors`
		`SV.S $M7,$s0,$t1 #store result vector`

Vector-matrix multiplication	c(1:n) = a(1:n) * b(1:n, 1:n)	`LRH.S $M3,$t0,$t1 #load row horizontal 1x8`
		`LM.S $M4,$t6,$t1 #load matrix 8x4`
		`RMAC #Reset MAC`
		`MULVM.S $M5,$M3,$M4 #vector-matrix multiply`
		`ADDR.S $M7,$M5,$M7 #accumulate row result`

Matrix-matrix multiplication	c(1:n, 1:n) = a(1:n,1:n) * b(1:n,1:n)	`LMH.S $M3,$t0,$t1 #load matrix horizontal 4x8`
		`LM.S $M4,$t7,$t1 #load matrix 8x4`
		`RMac #Reset MAC`
		`MULMM.S $M5,$M3,$M4 #matrix-matrix multiply`
		`ADDD.S $M7,$M5,$M7 #accumulate block result #and use DIM control register`

Strp, *Wstrp*, and *Dim* at the same time whereas SETN, SETW and SETD instructions individually set these control registers, respectively. In the strip mining technique (see Figure 3a), *n*-element vectors are divided into $\lceil n/32 \rceil$ segments (*MSR***MSW* = 8*4 = 32 elements). The remainder, which equals *n*%32, is divided into two segments, one segment with *Strp* = $\lfloor n\%32/4 \rfloor$ and *Wstrp* = 4 and the other segment with *Strp* = 1 and *Wstrp* = *n*%32%4. After strip mining vectors, the Mat-Core instructions shown in Table 1 is added in the inner loop. The inner loop of scalar-vector multiplication and SAXPY need only three and five vector/matrix instructions, respectively.

In vector-matrix multiplication (1×*n* times *n*×*w*), strip mining and block mining techniques are used (see Figure 3b). The input vector is divided into $\lceil n/8 \rceil$ strips each one has 1×8 elements, however, the last strip has 1×(*n*%8) elements. Moreover, matrix is divided into $\lceil w/4 \rceil$ columns, each column has $\lceil n/8 \rceil$ blocks, where $\lfloor w/4 \rfloor$*$\lfloor n/8 \rfloor$ blocks each has 8×4 elements. The last block in any column except the last has (*n*%8)×4 elements. The last column has $\lceil n/8 \rceil$ blocks of 8×(*w*%4) and the last block of this column

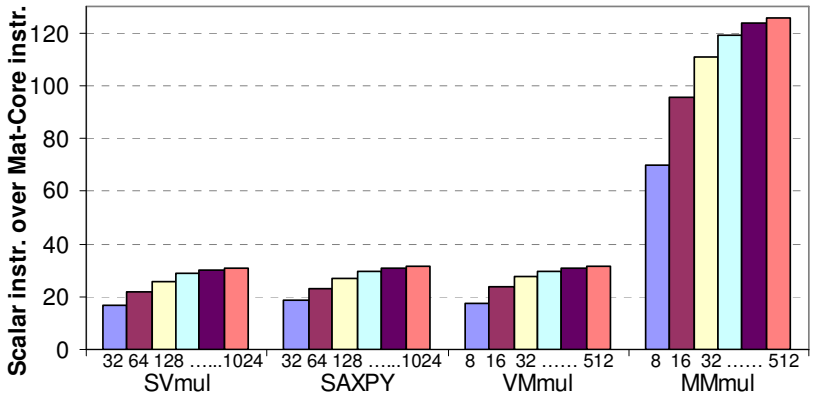

Figure 6: Ratio of scalar to Mat-Core instructions

has (n%8)×(w%4) elements. Vector strips are multiplied by each column blocks to produce a single strip of the result vector. Thus, vector-matrix multiplication needs two nested loops for strip/block mining. The inner loop needs only five vector/matrix instructions, as Table I shows. The first two instructions load a strip (8 elements or less) and a block (8×4 elements or less). The third instruction is used to reset the multiply-accumulate (MAC) functional unit. The fourth instruction does vector-matrix multiplication (1×8 strip times 8×4 block). The result vector-matrix multiplication is accumulated in a matrix register in the fifth instruction.

In matrix-matrix multiplications (m×n times n×w), block mining technique is used, which requires reading *MSR*, *MSW*, and *MDIM* control registers. The first matrix is divided into $\lceil m/4 \rceil$ rows, each row has $\lceil n/8 \rceil$ blocks, where $\lfloor m/4 \rfloor$*$\lfloor n/8 \rfloor$ blocks each has 4×8 elements. The last block in each row has 4×(n%8) elements. The last row has $\lceil n/8 \rceil$ blocks of (m%4)×8, however, the last block in this row has (m%4)×(n%8) elements. The second matrix is divided as the matrix in vector-matrix multiplication (see Figure 3c). Multiplying row blocks in the first matrix by column blocks in the second matrix and accumulating the results produce one block of the output matrix. Matrix-matrix multiplication needs the same number of Mat-Core instructions as vector-matrix multiplication; however, it requires three nested loops for block mining. This shows how the Mat-Core ISA reduces the semantic gap between high level language and hardware. It also shows that the data parallelism found in applications is exploited directly by Mat-Core ISA and convoyed to hardware in a compact form. As Figure 6 shows, the use of Mat-Core ISA results in reduction of the number of instructions by 15-30 times on applications dominated by scalar-vector, vector-vector, and vector-matrix kernels and by 60-125 times on applications dominated by matrix-matrix kernels. However, the use of

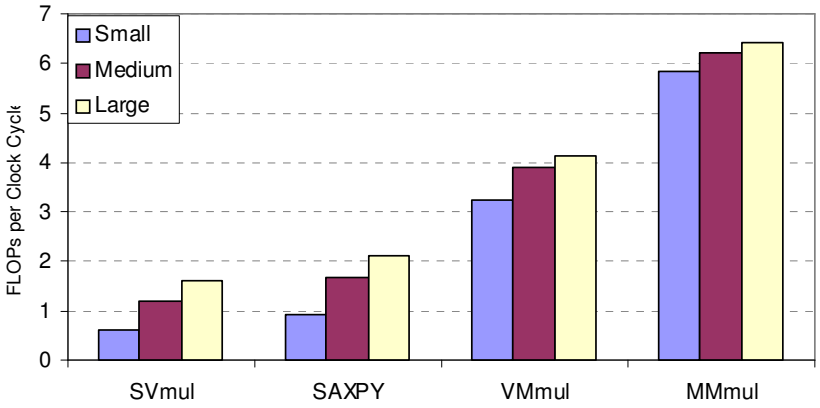

Figure 7: Performance evaluation of Mat-Core

scalar ISA results in scattering the data parallelism by compilers and then gathering it again using complex hardware.

Figure 7 shows the performance of Mat-Core processor with four parallel lanes and matrix registers of size 8×4 or 32 elements on scalar-vector, vector-vector, vector-matrix, and matrix-matrix kernels. The performance is evaluated on three categories of vector/matrix dimensions small (50-element vectors or 50×50-element matrices), medium (200-element vectors or 100×100-element matrices), and large (8K-element vectors or 300×300-element matrices). It is clear that the performance of Mat-Core processor is higher on computationally intensive kernels than on memory intensive kernels. Moreover, as the vector length increases, the loop overhead per element decreases, this increases the overall performance. A performance of about 1.6 and 2.1 FLOPs per clock cycle are achieved on SVmul and SAXPY, respectively, as shown in Figure 7. The maximum performances are four FLOPs per clock cycle on SVmul (four multiply operations can be processed in parallel in a clock cycle) and eight FLOPs per clock cycle on SAXPY because of chaining the results of the four multipliers and four adders. Note that as the number of memory references per FLOP decreases the performance increases (compare the performance of SAXPY, which has 3/2 memory references per FLOP, and SVmul, which has 2/1 memory references per FLOP).

Figure 7 shows also the performance of vector-matrix and matrix-matrix multiplications on four lanes Mat-Core processor. With 8×4 matrix registers, a performance of 4.1 and 6.4 FLOPs per clock cycle are achieved on vector-matrix and matrix-matrix multiplications, respectively. Due to reusing the loaded data $O(n)$ times in case of multiplying two $n \times n$ matrices, the performance of matrix-matrix multiplication is better than vector-matrix multiplication, which reuses the loaded data $O(1)$ times. The performance of matrix-matrix multiplication on Mat-Core represents 80% of the maximum performance; eight FLOPs can be executed in parallel on four parallel lanes.

5. CONCLUSION

This paper shows how the codevelopment of multi-level ISA and hardware results in an efficient matrix processor called Mat-Core. 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. Scalar/vector/matrix instructions can be executed on the Mat-Core hardware. These instructions can convoy up to 3-D data parallelism to hardware. Mat-Core ISA reduces the semantic gap between high level language and hardware. The data parallelism found in applications is exploited directly by Mat-Core ISA and convoyed to hardware in a compact form. However, the use of scalar ISA results in scattering the data parallelism by compilers and then gathering it again by complex hardware. The use of Mat-Core ISA results in reduction of the number of instructions by 15-30 times on applications dominated by scalar-vector, vector-vector, and vector-matrix kernels and by 60-125 times on applications dominated by matrix-matrix kernels.

Three passes assembler for Mat-Core is demonstrated for writing kernels in assembly language instead of machine code. The performance of Mat-Core processor is evaluated on scalar-vector, vector-vector, vector-matrix, and matrix-matrix kernels using SystemC. On four parallel lanes Mat-Core and matrix registers of size 8×4 or 32 elements, our results show performances of about 1.6, 2.1, 4.1, and 6.4 FLOPs per clock cycle achieved on scalar-vector multiplication, SAXPY, vector-matrix multiplication, and matrix-matrix multiplication, respectively.

REFERENCES

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

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

[3] M. Soliman, "Mat-Core: A Matrix Core Extension for General Purpose Processors," *Proc. The 2007 International Conference on Computer Engineering & Systems (ICCES'07),* Cairo, Egypt, pp. 304-310, *November* 2007.

[4] J. Smith and G. Sohi, "The Microarchitecture of Superscalar Processors," *Proceedings of the IEEE*, Vol. 83, No. 12, pp. 1609-1624, 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] C. Lee, *Code Optimizers and Register Organizations for Vector Architectures*, Ph.D. Thesis, Computer Science Division, University of California at Berkeley, 1992.

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

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

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

[10] R. Espasa, M. Valero, and J. Smith, "Vector Architectures: Past, Present and Future," *Proc. 2th International Conference on Supercomputing*, Melbourne, Australia, pp. 425-432, July 1998.

[11] J. Smith, "The Best Way to Achieve Vector-Like Performance?", *Proc. 21st International Symposium on Computer Architecture*, Denver, CO, June 1997, Slides in http://www.engr.wisc.edu/ece/faculty/smith_james.html.

[12] M. Soliman and A. Al-Junaid "SystemC Implementation of Mat-Core: A Matrix Core Extension for General-Purpose Processors," *Proc. 4th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era*, April 2009, Egypt.

[13] C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, and K. Yelick, "Hardware/Compiler Codevelopment for an Embedded Media Processor," *Proceedings of the IEEE*, Vol. 89, No. 11, pp. 1694-709, November 2001.

[14] R. Ho, K. Mai, and M. Horowitz, "The Future of Wires," *Proceedings of the IEEE*, Vol. 89, pp. 490-504, No. 4, April 2001.

[15] R. Ho, K. Mai, and M. Horowitz, "Efficient On-Chip Global Interconnects," *Proc. IEEE Symposium on VLSI Circuits*, pp. 271- 274, June 2003.

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

[17] 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.

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

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

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

[21] R. Espasa and M. Valero, "Decoupled Vector Architecture," *Proc. 2nd International Symposium on High-Performance Computer Architecture*, San Jose, CA, pp. 281-290, February 1996.