Neural, Parallel, and Scientific Computations 21 (2013) 87-110

CYCLE ACCURATE MODELS FOR INVESTIGATING THE SCALABILITY OF MAT-CORE PROCESSOR

MOSTAFA I. SOLIMAN[1], MOUMEN T. EL-MELEGY[2], AND ABDULMAJID F. AL-JUNAID[3]

[1]Computers & Systems Section, Electrical Engineering Department, Faculty of Engineering, Aswan University, Aswan 81542, Egypt,

[2]Computers & Systems Section, Electrical Engineering Department, Faculty of Engineering, Assiut University, Assiut, Egypt

[3]Electronic & Communication Engineering Department, Faculty of Engineering and Architecture, Ibb University, Ibb, Yemen

Abstract. Mat-Core is a matrix processor aiming at exploiting the increasingly number of transistors per IC to improve the performance of a wide range of applications. It 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. In this paper, four cycle accurate models are implemented using SystemC (system level modeling language) to investigate the scalability of Mat-Core. They include Mat-Core having (1) 1-lane with 8×1 vector registers, (2) 4-lane with 4×4 matrix registers, (3) 4-lane with 8×4 matrix registers, and (4) 8-lane with 8×8 matrix registers. The first model exploits scalar/vector ISAs, however, the remaining three models exploit scalar/vector/matrix ISAs. Moreover, this paper describes in detail the implementation of some kernels on the Mat-Core processor and discusses its scalability. Our results show that increasing the number of parallel lanes from one to four and from one to eight speedup the execution of kernels by factors of 3.6x-4.8x and 7.94x-10.6x, respectively, which indicates the scalability of Mat-Core architecture. In addition, the maximum performance of the Mat-Core processor on math intensive kernels represents 90% of the ideal value.

Keywords - scalable architecture, high performance computing, performance evaluation, vector/matrix processing.

1. INTRODUCTION

Scalability problem is considered as a major challenge for processor designers. Architecture scalability simply means that a very large computer can be built from a large number of basic components (computers, processors or processing elements, memories, and switches) with no single bottleneck component. Thus, the computer can be increasingly expanded over its designed scaling range, delivering linear incremental performance for a well-defined set of applications. This paper investigates the scalability of Mat-Core architecture with different number of parallel lanes (one, four, and eight) on some kernels of linear algebra (scalar-vector multiplication, SAXPY: single-precision scalar A times vector X plus vector Y, Givens rotation, rank-1 update, vector-matrix multiplication, and matrix-matrix multiplication), DCT/IDCT, and image registration.

Received February 5, 2012

Mat-Core is a matrix processor aiming at exploiting the increasingly number of transistors per IC to improve the performance of a wide range of applications [1]-[3]. As Figure 1 shows, Mat-Core extends a general-purpose scalar processor with a matrix unit for processing vector/matrix data. The extended matrix unit is decoupled into two components to hide memory latency: data computation and address generation, which communicate through data queues. The matrix unit consists of five units: instruction flow, load/store (address generation), matrix control, matrix register file (RF), and functional units. As in conventional processors, instructions and data are loaded from L2 cache into instruction and L1 data caches, respectively. Each instruction (scalar/vector/matrix) is fetched from instruction cache and sent in-order to the decode stage. If the fetched instruction is scalar, it completes remaining cycle of execution on the scalar pipeline stages (read operand(s), execute, memory access, and write-back). However, vector/matrix instructions are fetched in-order from instruction cache and sent to the matrix unit for execution on parallel lanes. When the result of executing a vector/matrix instruction is a scalar value, the matrix unit sends it back to the scalar unit for storing in the scalar register file. Otherwise, vector/matrix results are stored in the matrix register file inside the matrix unit.

As in vector processors [4]-[9], 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. However, on these parallel lanes not only vectors but also matrix data can be processed. Hence, Mat-Core processor inherits from a vector processor design the relatively straightforward means to scale performance. By increasing the number of parallel lanes, designer can easily increase the amount of data-level parallelism exploited. This also allows designers to easily scale the processor design to exploit the increased number of transistors that continue to grow according to Moore's law [10].

In this paper, four versions of Mat-Core processor are implemented by SystemC (system level modeling language) and evaluated to show its scalability. These versions are different in the number of parallel lanes and the size of registers in the matrix unit of

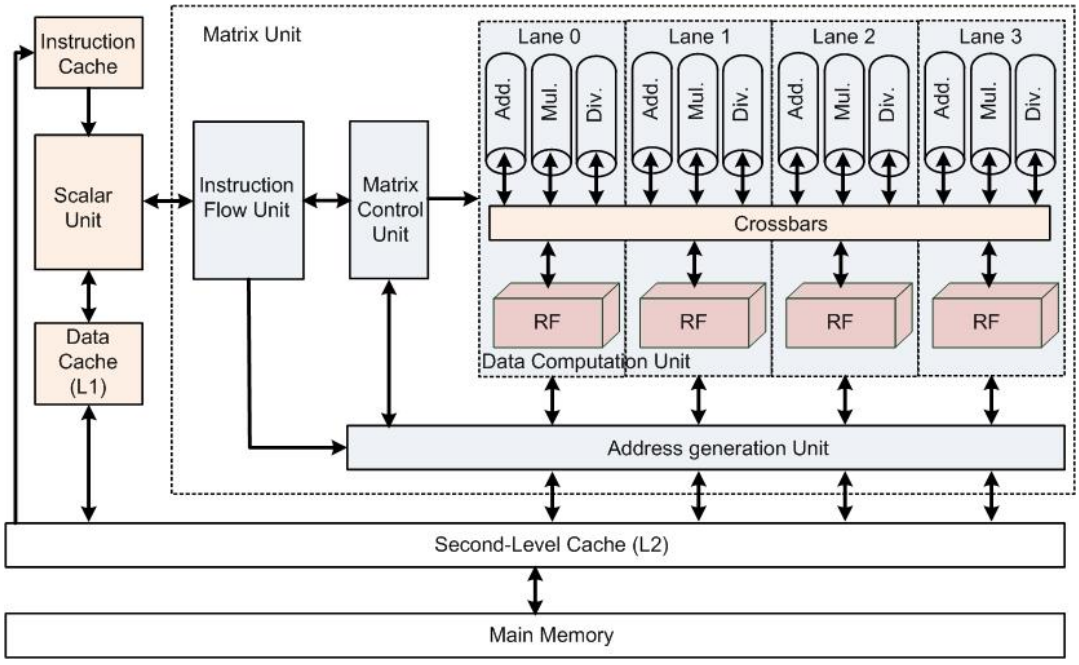


Figure 1: The block diagram of Mat-Core processor.

Mat-Core architecture. The first version contains one lane with vector register length of eight elements (Mat-Core-8×1). It exploits only scalar and vector instruction set architectures (ISAs). However, the remaining versions can exploit the three levels (scalar/vector/matrix) of Mat-Core ISA [11]. The second and third versions contain four lanes but they are different in the size of matrix registers (4×4 for Mat-Core-4×4 and 8×4 for Mat-Core-8×4). These versions show that scaling the matrix register size results in improving the performance of Mat-Core with the same number of parallel lanes. This is because larger matrix register size amortizes the pipeline latency of functional units. The last version has eight lanes with matrix registers of size 8×8 (Mat-Core-8×8).

This paper is organized as follows. The hardware scalability of Mat-Core is discussed in detail in Section 2. Section 3 describes the implementation of DCT/IDCT using Mat-Core instructions. The implementations of 3D affine transformation and SAD (sum of absolute differences) are described in Section 4. Section 5 evaluates the performance of some kernels including linear algebra, DCT/IDCT, and image registration to investigate the scalability of Mat-Core architecture with variable number of lanes. Finally, Section 6 concludes this paper.

2. HARDWARE SCALABILITY OF MAT-CORE PROCESSOR

To reduce the execution time, most vector processors use parallel pipelines per functional unit [12]. Thus, a vector unit can be structured as parallel lanes, where each lane contains a portion of the vector register file and one pipeline for each vector functional unit. The concept of parallel lanes is fundamental for the vector microarchitecture, as it leads to advantages in performance, design complexity, and scalability.

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 the lane-based processors allows hardware scaling without implications due to the high latency of long, cross-chip wires [14],[15]. On parallel lanes, Mat-Core can execute matrix-scalar, matrix-vector, and matrix-matrix instructions in addition to vector-scalar and vector-vector instructions.

The block diagram of Mat-Core version with a single lane (Mat-Core-8×1) is shown in Figure 2. The lane contains eight vector registers; each one has eight 32-bit elements. The lane also contains five arithmetic pipelines for executing vector instructions: ALU, FP adder, FP multiplier, FP MAC (multiply-accumulate), and FP divider. The multiplexers select the two input vector registers to the specified arithmetic pipeline. Moreover, demultiplexer selects the output of a specified arithmetic pipeline to be written in the destination vector register. This version of Mat-Core processor can exploit only scalar and vector ISAs.

To investigate the scalability of Mat-Core processor, a scalar core is extended with *n* lanes matrix unit (see Figure 3). Like the first version, there are five functional units for executing vector/matrix instructions. However, each functional unit has *n* parallel pipelines (one pipeline per lane), where two 32-bit data are needed per pipeline for processing and 32-bit is produced as a result. These functional units operate under the control of the matrix control unit. Each lane contains eight banks (one bank for each matrix register). *n* banks distributed among *n* lanes construct a matrix register. Each bank has two read ports and one write port. To show the scalability of Mat-Core, the number of lanes *n* is varied from one to four (Mat-Core-4×4 and Mat-Core-8×4) and from one to eight (Mat-Core-8×8).

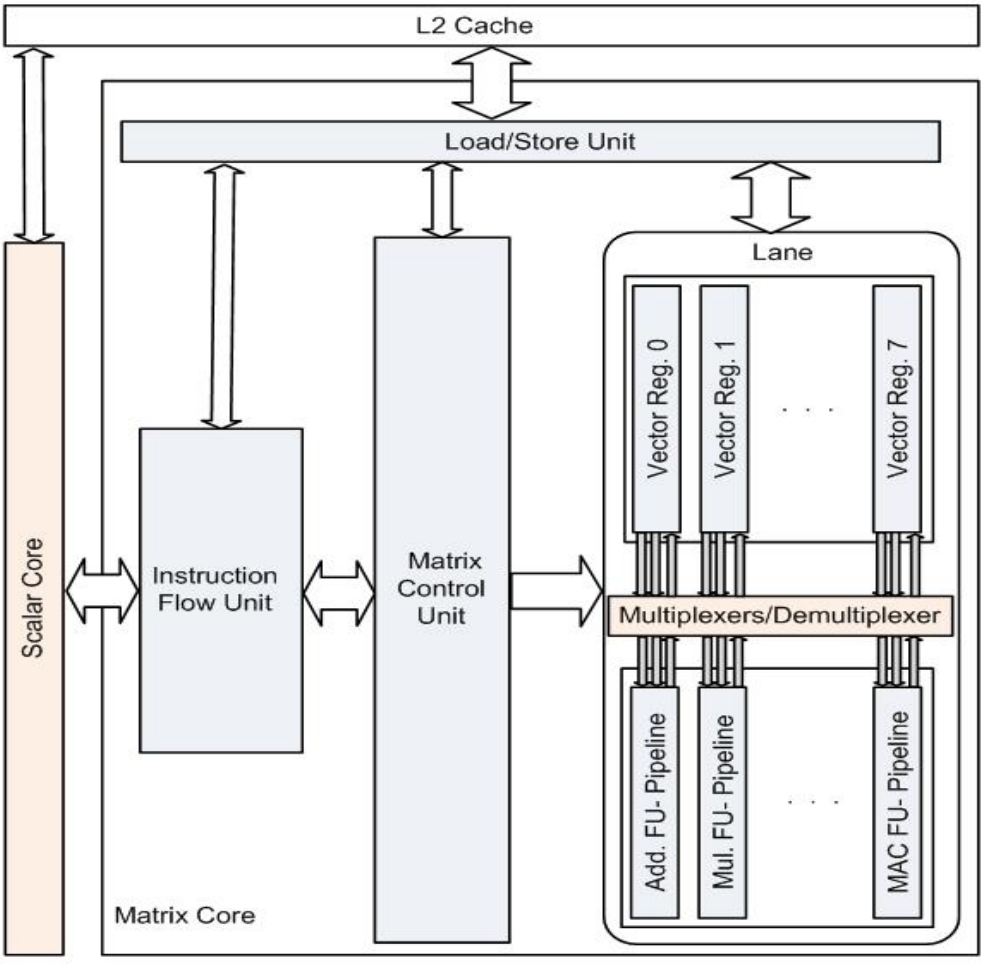


Figure 2: Mat-Core processor with one lane architecture.

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 others. 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 [2] for more detail about using crossbars in the execution of matrix/vector instructions. The use of crossbar in connecting Mat-Core lanes will limit its hardware scalability coming from increasing number of parallel lanes. Thus, in this paper the scalability of Mat-Core will be investigated when the number of parallel lanes varies from one to eight, which represent a small number of lanes. Extending the hardware scalability of Mat-Core can be achieved by providing more processor cores in a physical package as a multi-core processor (Multi-Mat-Core). On the Multi-Mat-Core, the performance could be scaled by parallel processing threads of codes using multi-threading techniques.

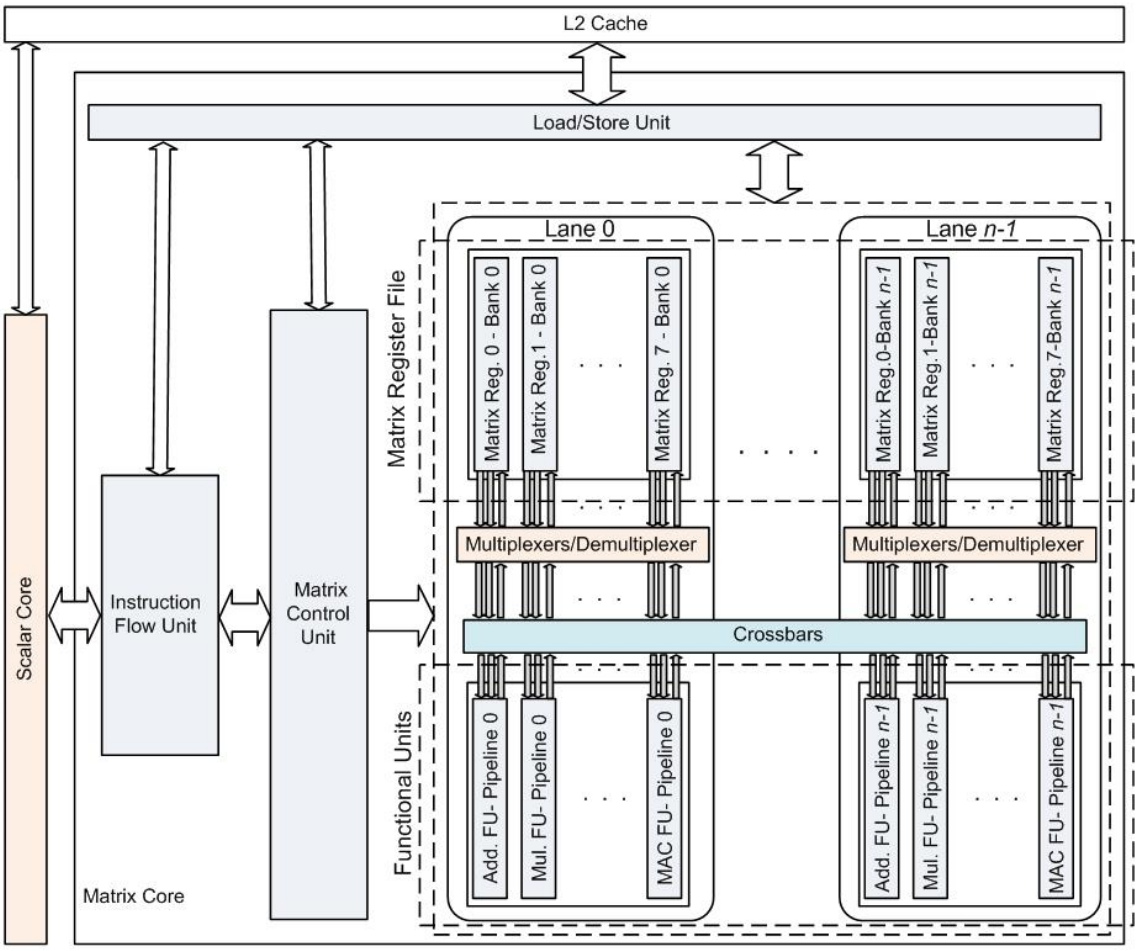


Figure 3: Mat-Core processor with n lane architecture.

3. DCT IMPLEMENTATION ON MAT-CORE

The discrete cosine transform (DCT) is a member of the sinusoidal family of unitary transforms. It has found applications in digital signal and image processing and particularly in transform coding systems for data compression/decompression. Besides being real, orthogonal, and separable, its properties are relevant to data compression and fast algorithms for its computation have proved to be of practical value. Recently, DCT has been employed as the main processing tool for data compression/decompression in international image and video coding standards—JPEG (Joint Photographic Experts Group) and MPEG (Moving Picture Experts Group) [16],[17].

The DCT gets its name from the fact that the rows of the $N\times N$ transform matrix M are obtained as a function of cosines. The conventional DCT in floating-point domain is implemented on $N\times N$ block of the image using the following equation: $B = M\times A\times M^{\mathrm{T}}$, where B is the transformed matrix, A is the input matrix, and M is the transformation matrix. The $(i, j)^{\text{th}}$ element of the transformation matrix M is given by

$$M_{ij} = \begin{cases} 1/\sqrt{N} & \text{if} \quad i = 0 \\ \sqrt{2/N}\cos((2j+1)i\pi/2N) & \text{if} \quad i>0 \end{cases} \qquad (1)$$

where $i, j = 0, 1, 2, \ldots, N$. Since M is orthogonal matrix, the IDCT (inverse DCT) could be computed as $A = M^{\mathrm{T}}\times B\times M$ [18].

The DCT and IDCT implementations require three types of matrix-matrix multiplications: $A\times M$, $A\times M^{T}$, and $M^{T}\times A$. In this section the implementation of $A\times M^{T}$ and $M^{T}\times A$ are discussed in details. However, the implementation of matrix-matrix multiplication ($A\times M$) on Mat-Core processor was described in [2].

The matrix register file (MRF) in Mat-Core allows accessing multiple data (one data item per lane), concurrently, to be processed on parallel lanes. These multiple data could be accessed from the same row of a matrix block loaded into a matrix register or from different rows, but no more than one element can be accessed from the same column (lane). By exploiting this ability of parallel accessing MRF and the shuffle operations (pass, rotate, and broadcast), the matrix transposition can be avoided in $A\times M^{\mathrm{T}}$ and $M^{\mathrm{T}}\times A$. It is well known that transposing a matrix is a very expensive operation since it takes $O(n^2)$ operations on $n\times n$ matrix. Thus, avoiding matrix transposition represents one of the main advantages of Mat-Core architecture. To carry out the matrix operation $A\times M^{T}$, the input matrix M should be loaded from memory to MRF in skewed form as shown in Figure 4. However, Mat-Core processes the input blocks of matrices without the need for skewing. The access abilities to MRF and the shuffle operations of crossbars substitute the skewing process.

Figure 5 shows the processing of $A\times M^{T}$ for two 4×4 matrix blocks stored in 4×4 matrix registers. The two matrix blocks are loaded from memory to MRF row by row without the need for skewing. The first row of the result matrix is produced after four steps (step1-step4). In each step, the first row (four elements) of the matrix A and the skewed row of matrix M (see Figure 4-b) are fed to four MAC (multiply-accumulate) pipelines. Feeding matrix M to the four MAC pipelines in the skewed form is achieved via accessing different rows of MRF at the same cycle and rotating them by crossbars. The shuffle operations for crossbars are *pass*, *rotate-1*, *rotate-2*, and *rotate-3* for step1,

step2, step3 and step4, respectively. The same shuffle operations applied to matrix *M* are also applied to matrix *A*. In the same manner, the second row of the result matrix is produced after four steps (step5-step8) and so on. Thus, the multiplication of two 4×4 matrix blocks requires 16 clock cycles in addition to the MAC pipelines latency.

Figure 6 shows the processing of $M^T \times A$ on two 4×4 matrix blocks stored in matrix registers. The two matrix blocks are loaded into MRF row by row without the need for transposition. The multiplication of two blocks is based on vector-matrix multiplication and its inner loop is based on SAXPY. However, the exception here is that the columns (instead of rows) of the first block *M* are taken as vectors to be multiplied by the rows of matrix *A*.

To improve the performance of Mat-Core without drastically increasing the complexity, 8×4 registers are used instead of 4×4. On Mat-Core with matrix register size of 8×4, the implementation of $A \times M^T$ and $M^T \times A$ is somehow different. This version of Mat-Core has two types of load instructions. One for loading matrix blocks with size 4×8 and the other for loading matrix blocks with size 8×4. Figure 7 shows the arrangement of 8×4 and 4×8 matrix blocks in matrix registers after executing *LMB* (Load Matrix Block) and *LMBH* (Load Matrix Block Horizontally), respectively.

To process $A \times M^T$ on this version of Mat-Core, it is required to load a block of 4×8 from matrix *A* and a block of 4×8 (instead of 8×4 because of the transposition) from matrix *M*. Then, the block 4×8 from matrix *A* is multiplied by the transpose of the block 4×8 from matrix *M*. Using the access abilities to MRF and the shuffle operations of crossbars substitute transposing the block 4×8 from matrix *M*. As shown in Figure 8-a, the production of each row of the result matrix block needs eight steps. For example, producing the first row of the result matrix block takes four steps for multiplying the first half of the first row R_{11} by the first block B_{11} and another four steps for multiplying the second half of the first row R_{12} by the second block B_{12}. Multiplying R_{11} by B_{11} or R_{12} by B_{12} is done in the same way shown in Figure 5 (step1-step4). The results are accumulated in the MAC pipelines through the eight steps to produce the first row of the result matrix block. Also, the second row of the result matrix block is produced by multiplying R_{21} by B_{11} and R_{22} by B_{12} and so on. Thus, processing $A \times M^T$ on two 4×8 matrix blocks requires 32 cycles in addition to the MAC pipelines latency.

m_{00}	m_{01}	m_{02}	m_{03}
m_{10}	m_{11}	m_{12}	m_{13}
m_{20}	m_{21}	m_{22}	m_{23}
m_{30}	m_{31}	m_{32}	m_{33}

(a) Before skewing

m_{00}	m_{11}	m_{22}	m_{33}
m_{01}	m_{12}	m_{23}	m_{30}
m_{02}	m_{13}	m_{20}	m_{31}
m_{03}	m_{10}	m_{21}	m_{32}

(b) After skewing

Figure 4: Skewing the input matrix *M* for $A \times M^T$.

To process $M^T \times A$ on Mat-Core, it is required to load a block of 8×4 from matrix M (instead of 4×8 because of the transposition of M) and a block of 8×4 form matrix A. The two blocks are multiplied based on vector-matrix multiplication and the inner loop is based on SAXPY in the same manner as mentioned before in Figure 6. The two blocks are loaded to matrix registers as shown in Figure 8-b. The first row of the result matrix block is produced after eight steps. In the first step, m_{00} from the first column of M is broadcast to the four MAC pipelines with the first row of A. Then, the second element, m_{10}, from the first column of M is broadcast to the four MAC pipelines with the second row of A. After that, the process continues in the same way, broadcasting one element from the first column of M with the corresponding row of A. The second row of the result matrix block produces from the second column of M and the rows of A in the same

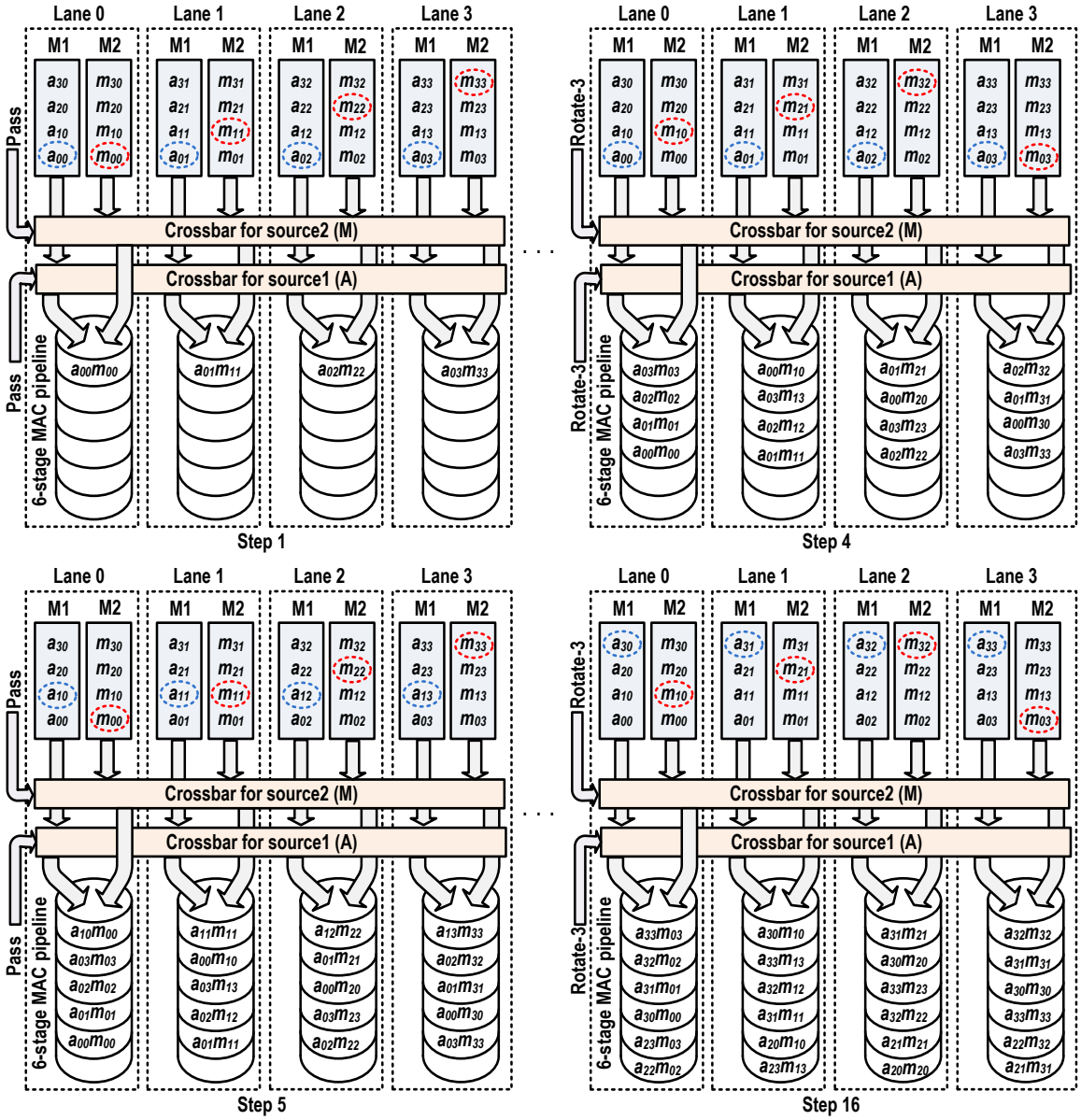


Figure 5: Processing $A \times M^T$ on Mat-Core processor for two 4×4 matrix blocks.

manner mentioned above for the first row and also takes eight steps. Thus, processing $M^T \times A$ on two 8×4 matrix blocks requires 32 steps.

In Mat-Core version with one lane, processing the operations $A \times M$, $A \times M^T$, and $M^T \times A$ is based on vector operations (scalar-vector multiplication and dot product). In addition, the implementation of DCT/IDCT on Mat-Core version with eight lanes is similar to what is discussed above for four lanes and matrix register size 4×4. However, multiplying two 8×8 blocks ($A \times M$, $A \times M^T$, and $M^T \times A$) on eight lanes requires 64 steps instead of 16 step in case of multiplying two 4×4 blocks.

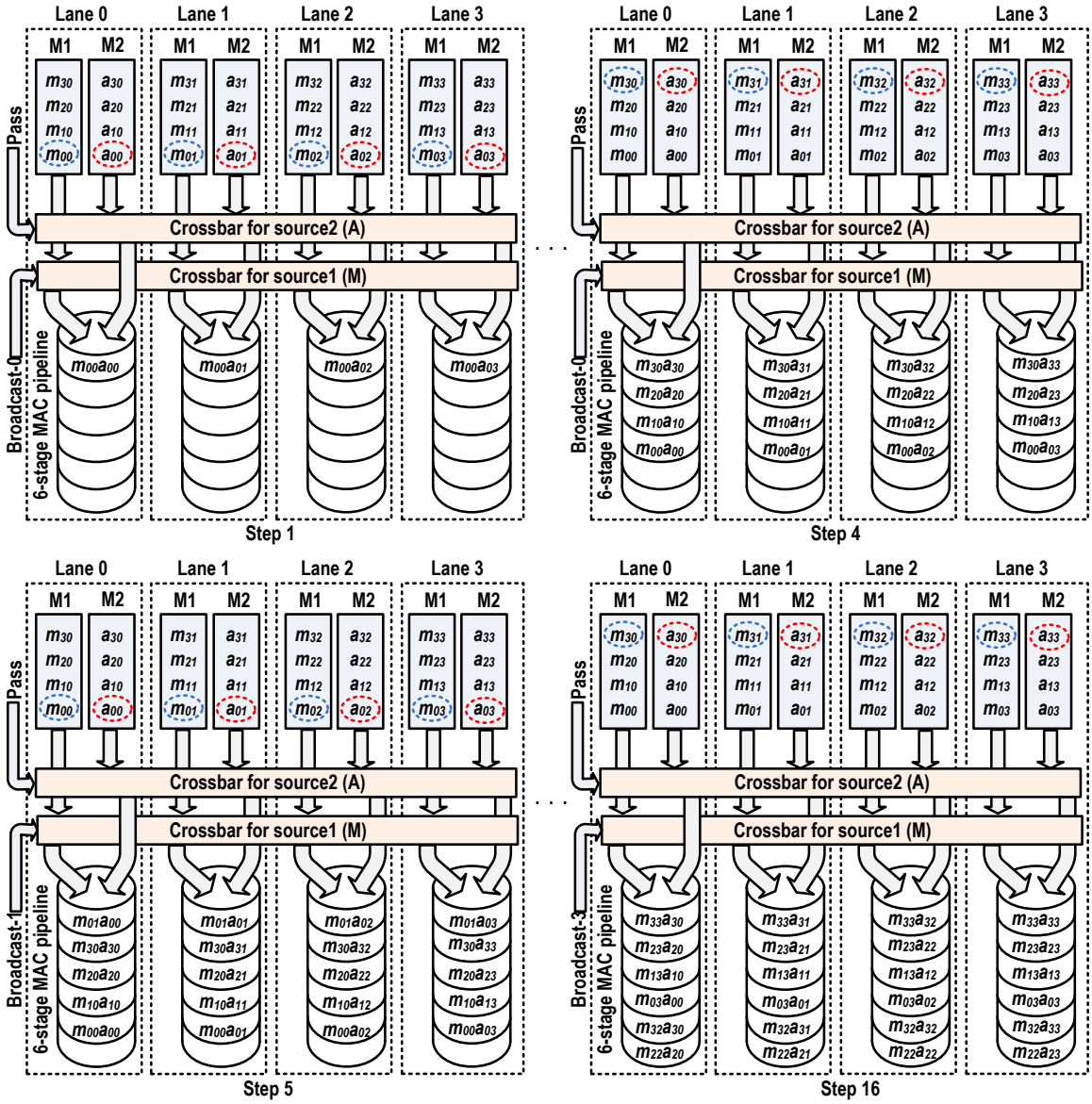


Figure 6: Processing $M^T \times A$ on Mat-Core processor for two 4×4 matrix blocks.

Lane0	Lane1	Lane2	Lane3
m_{70}	m_{71}	m_{72}	m_{73}
m_{60}	m_{61}	m_{62}	m_{63}
m_{50}	m_{51}	m_{52}	m_{53}
m_{40}	m_{41}	m_{42}	m_{43}
m_{30}	m_{31}	m_{32}	m_{33}
m_{20}	m_{21}	m_{22}	m_{23}
m_{10}	m_{11}	m_{12}	m_{13}
m_{00}	m_{01}	m_{02}	m_{03}

(a) Block of 8×4

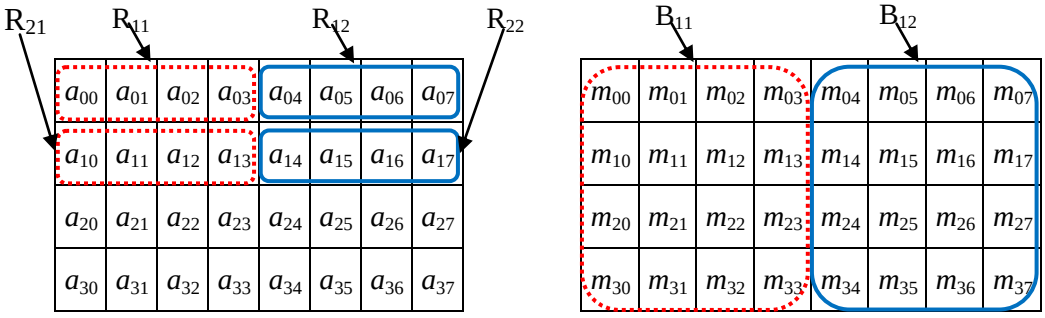
Address 7

Address 0

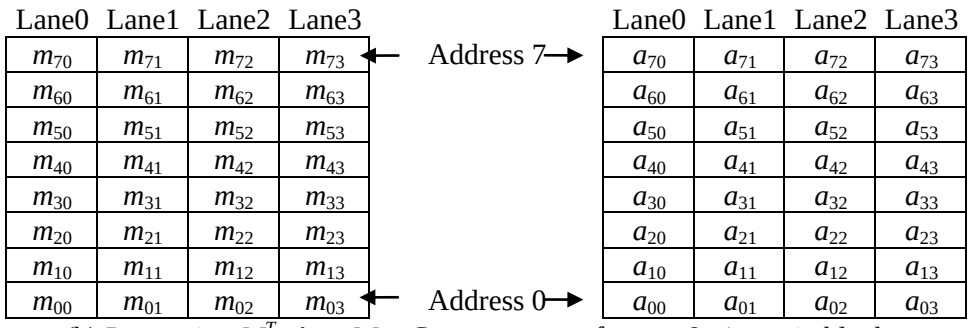
Lane0	Lane1	Lane2	Lane3
m_{34}	m_{35}	m_{36}	m_{37}
m_{30}	m_{31}	m_{32}	m_{33}
m_{24}	m_{25}	m_{26}	m_{27}
m_{20}	m_{21}	m_{22}	m_{23}
m_{14}	m_{15}	m_{16}	m_{17}
m_{10}	m_{11}	m_{12}	m_{13}
m_{04}	m_{05}	m_{06}	m_{07}
m_{00}	m_{01}	m_{02}	m_{03}

(b) Block of 4×8

Figure 7: The arrangement of 8×4 and 4×8 matrix blocks in matrix registers.



(a) Processing $A \times M^T$ on Mat-Core processor for two 4×8 matrix blocks.



(b) Processing $M^T \times A$ on Mat-Core processor for two 8×4 matrix blocks.

Figure 8: Processing $A \times M^T$ and $M^T \times A$ on Mat-Core processor with matrix registers of 8×4.

4. 3D AFFINE TRANSFORMATION & SAD IMPLEMENTATION ON MAT-CORE

Registration is an important problem and a fundamental task in image processing and computer vision [19]-[22]. Image registration is the process of overlaying two or more images of the same scene taken at different times, from different viewpoints, and/or by different sensors. Registration of medical images is becoming an important tool for

medical treatment and medical analysis. By finding spatial relations between two or more images, it combines their information, which is useful for observing changes in anatomy and/or function during time, for comparing subjects, and for merging information of multiple images. Several techniques are proposed to find a geometrical transformation that relates the points of an image to their corresponding points of another image [19],[21].

A typical image registration algorithm consists of three coupled components:

- An alignment measure (also known as similarity measure, registration objective function, etc.) that quantifies the quality of alignment. There are many similarity measures used in image registration, such as sum of absolute differences (SAD), sum of squared intensity differences (SSD), correlation coefficient (CC), and mutual information (MI) [19]-[21];
- A class of admissible geometric transformations that can be applied to the image(s). For 2D/2D or 3D/3D registration problems, the spatial transformation can be rigid, affine or deformable. In a rigid transformation, only rotations and translations are allowed. Affine transformations allow skewing and scaling in addition to rotation and translation. Deformable transformations define free-form mappings and are typically used with a regularization constraint to limit the allowable solution space; and
- An optimizer that seeks the transformation that maximizes the similarity as quantified by the alignment measure [20],[22],[23].

The focus of this section is on the implementation of the first two components of image registration: the similarity measure and geometric transformation. That is because these two components are computationally intensive kernels. SAD similarity measure is selected due to its popularity, simplicity and suitability to hardware implementation. 3D affine transformation is selected as a moderate complexity between rigid and deformable transformations. Also, affine transformation is an important computationally intensive kernel in computer graphics [24]-[26].

3D affine transformations are the transformations that involve rotation, scaling, shear and translation. A matrix can represent an affine transformation and a set of affine transformations can be combined into a single overall affine transformation. Technically, it can be said that an affine transformation is made up of any combination of linear transformations (rotation, scaling and shear) followed by translation (technically, translation is not a linear transformation) [27]. In homogeneous co-ordinates it is possible to describe any transformation in a matrix notation:

$$\begin{bmatrix} x^* \\ y^* \\ z^* \\ 1 \end{bmatrix} = \left[\begin{array}{cccc} a & d & g & j \\ b & e & h & k \\ c & f & i & l \\ \hline 0 & 0 & 0 & 1 \end{array}\right] * \left[\begin{array}{c} x \\ y \\ z \\ \hline 1 \end{array}\right] \qquad (2)$$

where the vertex $(x\ y\ z\ 1)^T$ is transformed to $(x^*\ y^*\ z^*\ 1)^T$ and the 4×4 square matrix is the transformation matrix *T*. This universal matrix for transformations can be divided into four functional blocks:

$$\left[\begin{array}{c|c} \text{scaling and rotation} & \text{translation} \\ \text{part of the homogeneous representation} & 1 \end{array}\right]$$

Consider an object represented with N vertices. The new position (NP) of the object when applying a transformation can be calculated as follows:

$$NP = T * OP, \qquad (3)$$

where T is the matrix transform, OP is a $4\times N$ matrix contains the old vertices position and NP is a $4\times N$ matrix containing the new vertices position. Equation (3) could be written as:

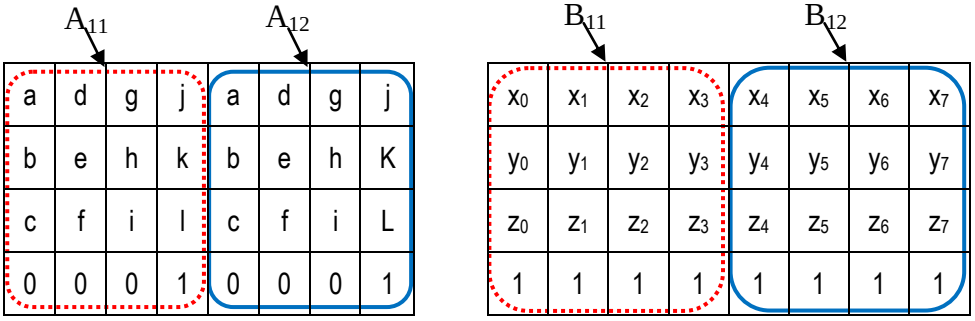
$$\begin{bmatrix} x_0^* & x_1^* & \dots & x_{N-1}^* \\ y_0^* & y_1^* & \dots & y_{N-1}^* \\ z_0^* & z_1^* & \dots & z_{N-1}^* \\ 1 & 1 & \dots & 1 \end{bmatrix} = \begin{bmatrix} a & d & g & j \\ b & e & h & k \\ c & f & i & l \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x_0 & x_1 & \dots & x_{N-1} \\ y_0 & y_1 & \dots & y_{N-1} \\ z_0 & z_1 & \dots & z_{N-1} \\ 1 & 1 & \dots & 1 \end{bmatrix} \qquad (4)$$

To execute the affine transformation (Equation 4) on Mat-Core version with 4×4 matrix registers, the 4×4 constant block (matrix T) is loaded into matrix register *M1* and four vertices (4×4 block) of the OP matrix are loaded into another matrix register *M2*. Then a matrix-matrix multiplication instruction is executed on the two blocks stored in matrix registers. After that, the next four vertices (4×4 block) of the OP matrix are loaded into *M2* and a matrix-matrix multiplication between the two matrix registers is performed and so on until finishing vertices. In each block multiplication, four vertices (4×4 block) of the NP matrix are produced and stored in memory.

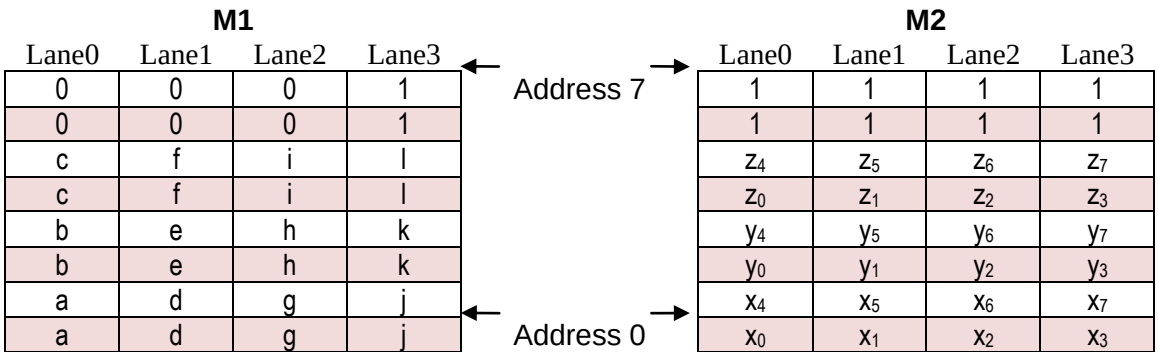
On the Mat-Core version with 8×4 matrix registers, two copies of 4×4 matrix T are loaded into matrix register *M1* using instruction LMBH (Load Matrix Block Horizontally) as 4×8 block (see the previous section) and eight vertices of the OP matrix are loaded into another 8×4 matrix register *M2* in the same manner (see Figure 9). A block matrix-matrix multiplication is done on the two matrix registers *M1* and *M2* using instruction *BMUL*. The *BMUL* instruction performs block-wise multiplication between 4×4 blocks. As shown in Figure 9-a, *BMUL* instruction multiplies block A_{11} by B_{11} (as matrix-matrix multiplication) and A_{12} by B_{12}. This instruction consumes 16 clock cycles for each two-block multiplication, so the total number of clock cycles that *BMUL* consumes is 32 plus the arithmetic pipelines latency. Figure 9-b shows the distribution of the blocks A_{11}, B_{11}, A_{12} and B_{12} in the matrix registers *M1* and *M2* after execution of *LMBH* instruction. Block A_{11} is in even rows (0, 2, 4, and 6) of *M1* (dark rows) and block A_{12} is in odd rows (1, 3, 5, and 7) of *M1* (white rows) and the blocks B_{11} and B_{12} are stored in the same manner in *M2*.

On the Mat-Core version with 8×8 matrix registers, four copies of 4×4 matrix *T* are loaded into matrix register *M1* using instruction *LMB* (Load Matrix Block) as 8×8 block and 16 vertices of the *OP* matrix are loaded into another matrix register *M2* using instruction *LMBH*, as shown in Figure 10. The four copies of 4×4 matrix *T* are distributed in *M1* as follows. The first and second copies are in even rows (0, 2, 4, and 6) in lanes L0-L3 and lanes L4-L7, respectively (dark rows). The third and fourth copies are in odd rows (1, 3, 5, and 7) in lanes L0-L3 and lanes L4-L7, respectively (white rows). In the same manner, the four groups of vertices (each group consists of four vertices) are distributed in *M2* (see Figure 10). Note that the eight lanes are communicated with each other via two levels of crossbars. The first level (4-input/4-output crossbars) connects each four lanes as a group and the second level (8-input/8-output crossbars) connects the two groups as shown in Figure 10. This two-level crossbar gives us two selections; the first one is to use the eight lanes as two separate 4-lane groups, which is useful for block-wise multiplication as an example. The second selection is to use the lanes as one 8-lane group, which is useful in matrix-matrix multiplication of 8×8 blocks as an example. The first selection can be performed by controlling the first crossbars level and applying *pass* signal on the second crossbars level. The opposite thing is done for the second selection by applying *pass* signal on the first crossbars level and controlling the second crossbars level.

On the Mat-Core version with 8×8 matrix registers, the *BMUL* instruction performs



(a) Block-wise multiplication of 4×4 blocks.



(b) Distribution of 4×4 blocks on matrix registers.

Figure 9: Processing 3D affine transformation on Mat-Core processor with 8×4 matrix registers.

four block matrix-matrix multiplication; each of them is element-wise multiplication between 4×4 blocks. *BMUL* instruction execution takes 32 clock cycles plus the arithmetic pipelines latency. In the first half of *BMUL* execution, a block-wise multiplication is performed between the first and second copies of matrix *T* in *M1* and the first and second groups of vertices in *M2*, respectively. These two matrix-matrix multiplications between the 4×4 blocks are performed simultaneously; each one of them is on 4-lane group. In the second half of *BMUL* execution, a block-wise multiplication is performed between the third and fourth copies of matrix *T* in *M1* and the third and fourth groups of vertices in *M2*, respectively. Also, they are performed simultaneously, each one of them on 4-lane group.

Sum of Absolute Differences (SAD) criteria is a popular intra-modality alignment measure [28],[29]:

$$SAD = \sum_{i=1}^{n} \left| R(\mathrm{x}_i) - I(T(\mathrm{x}_i)) \right|, \qquad (5)$$

where $R(x_i)$, $I(T(x_i))$ are the intensity values at the corresponding voxel x_i in the reference image *R* and the target image *I*, respectively. *n* is the total points of the image or the number of vertices. *T* is the transformation model.

To execute SAD (Equation 5) on Mat-Core version with 4×4 matrix registers, two 4×4 blocks from the reference image *R* and the target image *I* are loaded into matrix registers *M1* and *M2*, respectively. Then the instruction *ABD.MM* (stands for Absolute Difference Matrix-Matrix) is used to calculate the absolute difference between the two blocks of data stored in matrix registers *M1* and *M2*. The result of *ABD.MM* instruction is accumulated in matrix register *M3* using the instruction *Add.MM* (stands for Add Matrix-Matrix). The process continues until reaching the end of the two images. At that point,

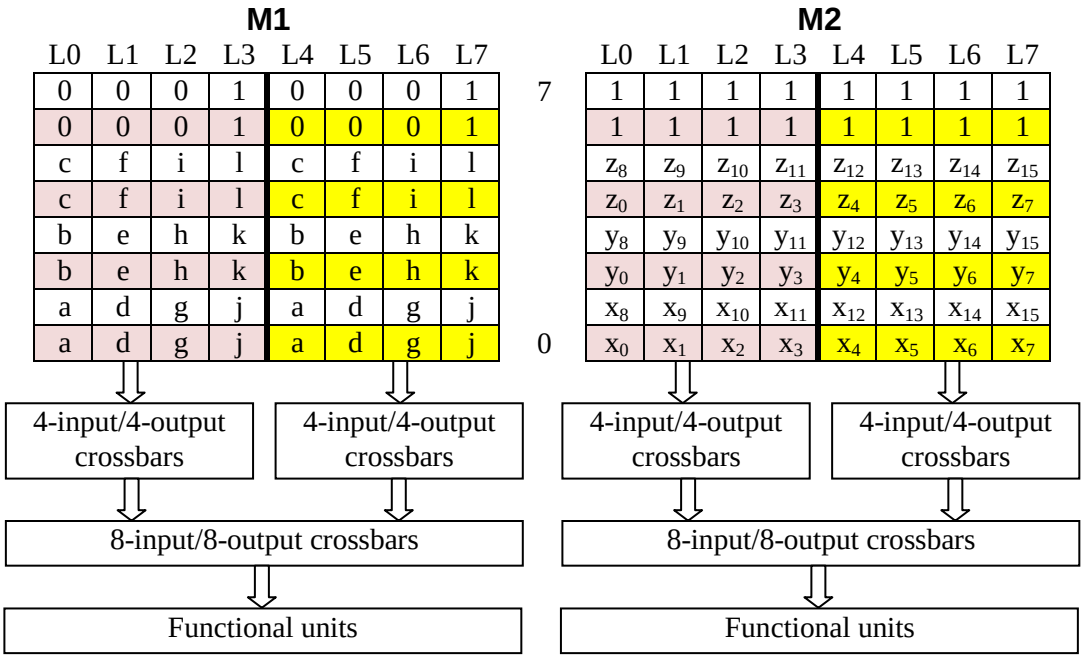


Figure 10: Processing 3D affine transformation on Mat-Core processor with 8×8 matrix registers.

the result is 4×4 block in *M3*. Accumulating the elements of *M3* gives the final scalar result which can be done by using two instructions *MAC.MM* (multiply-accumulate) and *PACC* (partial-accumulate). The first instruction *MAC.MM* does a vertical reduction by element-wise multiplication of the rows of *M3* by the corresponding rows of *M4* and accumulating the result in the *MAC* pipelines. Note that M4 is loaded by ones. The second instruction *PACC* performs a horizontal reduction of the vertical accumulated results in the *MAC* pipelines.

The execution of SAD on Mat-Core8×4 and Mat-Core8×8 is similar to what is mentioned above on Mat-Core4×4. The difference is that larger blocks are processed in each iteration (8×4 and 8×8 blocks for Mat-Core8×4 and Mat-Core8×8, respectively).

5. SCALING PERFORMANCE ON MAT-CORE PROCESSOR

This section studies the performance scalability of Mat-Core processor on linear algebra kernels, DCT/IDCT, and image registration. Four versions of Mat-Core processor are used for performance evaluation of these kernels to show the scalability of Mat-Core processor. The number of lanes is varied from one to four and from one to eight. The number of registers is constant (eight registers), however, the size of registers is varied from 8×1 to 4×4 to 8×4 to 8×8.

5.1 Performance of Linear Algebra Kernels on Mat-Core Processor

Table 1 shows the floating-point operations (FLOPs) and memory references (load/store operations) for some linear algebra kernels used for performance evaluation of Mat-Core. It is clear that the ratio of FLOPs to memory references is different. For example, Givens rotation is more computationally intensive than vector-scalar multiplication, since the ratio of FLOPs to memory references in the former is higher than the later. In general, the performance of Mat-Core processor is higher on computationally intensive kernels than on memory intensive kernels, as will be shown in this section.

Figure 11(a-c) shows the performance of vector kernels on Mat-Core processor with different number of lanes. Scaling the matrix unit from one lane to four lanes with 4×4 matrix registers improves the performance of scalar-vector multiplication, SAXPY, and Givens by factors of 3.1, 3.2, and 2.75, respectively. In addition, scaling the matrix registers from 4×4 to 8×4 without changing the number of parallel lanes (four) results in scaling the performance by factors of 1.2, 1.3, and 1.4, respectively. Moreover, scaling the matrix unit from four lanes with 4×4 matrix registers to eight lanes with 8×8 matrix registers improves the performance of vector kernels by factors of 2.6, 2.5, and 2.9, respectively. The speedup due to scaling one lane to eight lanes is around eight, which indicates the scalability of the Mat-Core architecture.

Figure 11(d, e) shows the performance of matrix-vector kernels on Mat-Core processor with one, four, and eight lanes. Increasing the number of lanes from one lane to four lanes with 4×4 matrix registers improves the performance of rank-one update and vector-matrix multiplication by factors of 3.5 and 2.75, respectively. Moreover, scaling the matrix registers from 4×4 to 8×4 in Mat-Core with four parallel lanes results in improving the performance by factors of 1.4 and 1.6, respectively. In addition, scaling the matrix unit from four lanes with 4×4 matrix registers to eight lanes with 8×8 matrix registers improves the performance of matrix-vector kernels by factors of 2.7 and 3.2, respectively. The speedups due to scaling one lane to eight lanes are 9.6 and 8.8, respectively. Reusing the loaded data in matrix register by using matrix ISA is greater than reusing the loaded data in vector registers using vector ISA, which results in super-linear scaling.

Matrix-matrix multiplication is one of the most fundamental operations in numerical linear algebra [30]. Although this problem is simple mathematically, it is very rich from the computational point of view. Accumulating $C_{m\times n}$ matrix with the multiplication of $A_{m\times w}$ matrix by $B_{w\times n}$ matrix ($C_{m\times n} = C_{m\times n} + A_{m\times w} \times B_{w\times n}$) needs $2mwn$ FLOPs while at least ($2mn + mw + wn$) memory operations being needed. The worst case (zero-reusing data) leads to needing ($2mn + 2mwn$) memory operations because the matrix $B_{w\times n}$ should be loaded m times or the matrix $A_{m\times w}$ should be loaded n times. This means ($2mn + mw + wn$) ≤ (memory operations for matrix-matrix multiplication) ≤ ($2mn + 2mwn$). On P Mat-Core lanes, the required number of memory operations is ($2mn + 2mwn/P$) because the matrix $B_{w\times n}$ should be loaded m/P times or the matrix $A_{m\times w}$ should be loaded n/P times. This results in ($1/w + 1/P$) memory operations per FLOP, where the optimum value is ($1/w + 1/2m + 1/2n$) memory operations per FLOP. Increasing the number of parallel lanes P leads the required number of memory operations to be closer to the optimal value. In other words, when $m = n = w = P$, the Mat-Core processor performs ideally since the matrices can be loaded into matrix registers and the highest reuse of data occurs.

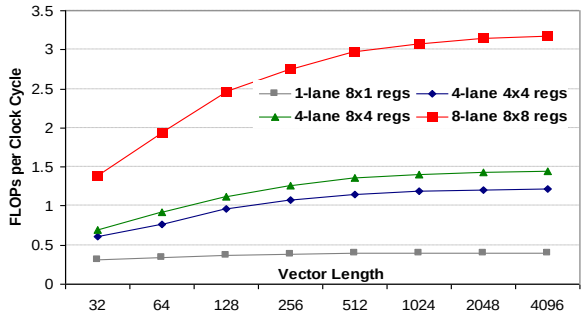
Figure 11-f shows the performance of matrix-matrix multiplication on Mat-Core processor with one, four, and eight lanes. Increasing the number of lanes from one lane to

Table 1: Kernels for scaling performance on Mat-Core processor.

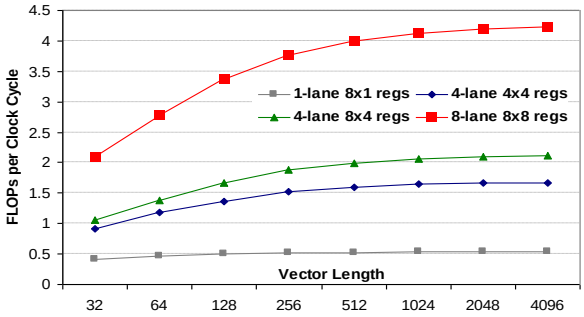
Kernel	Semantic	FLOPs/Memory References
Scalar-Vector multiplication	$x_i = a*x_i$, $1 \le i \le n$	n / $2n$
SAXPY	$y_i = a*x_i + y_i$, $1 \le i \le n$	$2n$ / $3n$
Givens rotation	$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^t \begin{bmatrix} x \\ y \end{bmatrix}$	$6n$ / $4n$
Rank-1 update (outer-product)	$A_{(i,j)} = A_{(i,j)} + x_i y_j$	$2n^2$ / from $2n^2+2n$ to $3n^2+n$
Vector-matrix multiplication	$y_j = y_j + \sum_{i=1}^{n} x_i A_{(i,j)}$	$2n^2$ / from n^2+3n to $2n^2+2n$
Matrix-matrix multiplication	$C_{n\times n}$ += $A_{n\times n}\times B_{n\times n}$	$2n^3$ / $O(n^2)$

four lanes with 4×4 matrix registers and to eight lanes with 8×8 matrix registers improves the performance by factors of 4.1 and 10.6, respectively. Moreover, scaling the matrix registers from 4×4 to 8×4 of Mat-Core with four parallel lanes results in improving the performance by a factor of 1.2.

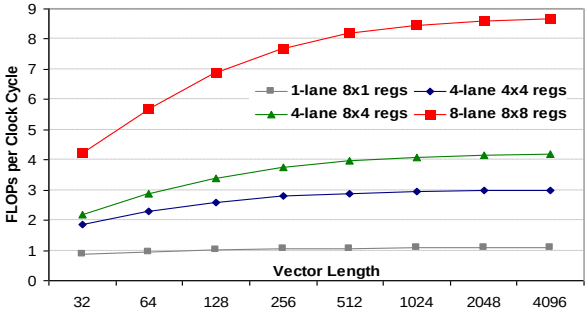
Figure 12 summaries the scalability of Mat-Core architecture on some linear algebra kernels. Moreover, Figure 13 shows the speedup of multiple lanes Mat-Core over one lane. The speedup due to scaling Mat-Core from one lane with 8-element vector registers to four lanes with 4×4 matrix registers is 2.75-4.1. Increasing the number of parallel lanes from one lane with 8-element vector registers to eight lanes with 8 x 8 matrix registers speeds up the execution of the six kernels by a factor of 7.94-10.6. Scaling the matrix registers from 4×4 to 8×4 improves the speedup by a factor of 1.2-1.6. Moreover, a speedup of 3.6-4.8 is achieved in four lanes with 8 x 4 matrix registers over one lane.



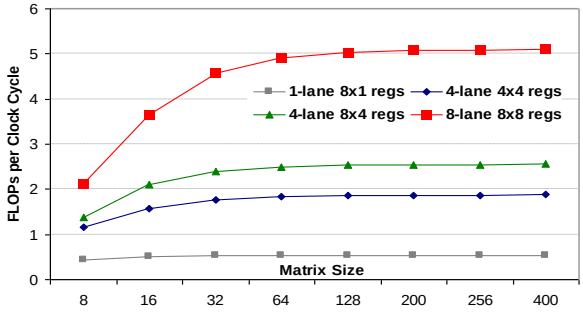
a) Scalar-vector multiplication



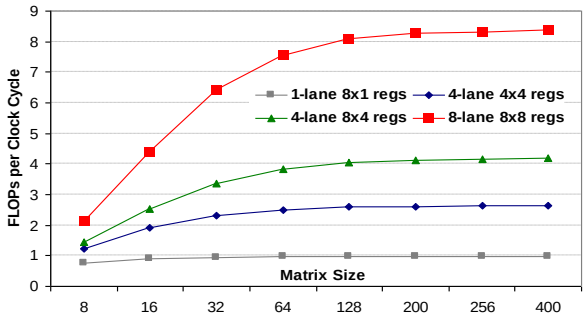
b) SAXPY



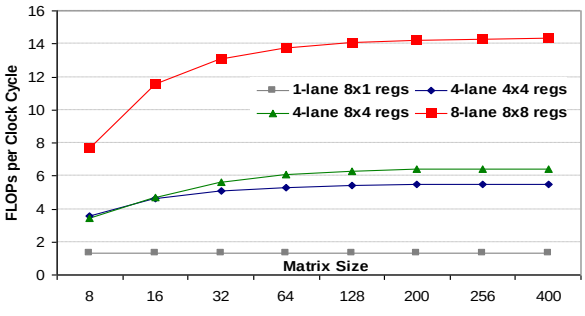
c) Givens



d) Rank-one update



e) Vector-matrix multiplication



f) Matrix-matrix multiplication

Figure 11: Performance scalability of linear algebra kernels on Mat-Core.

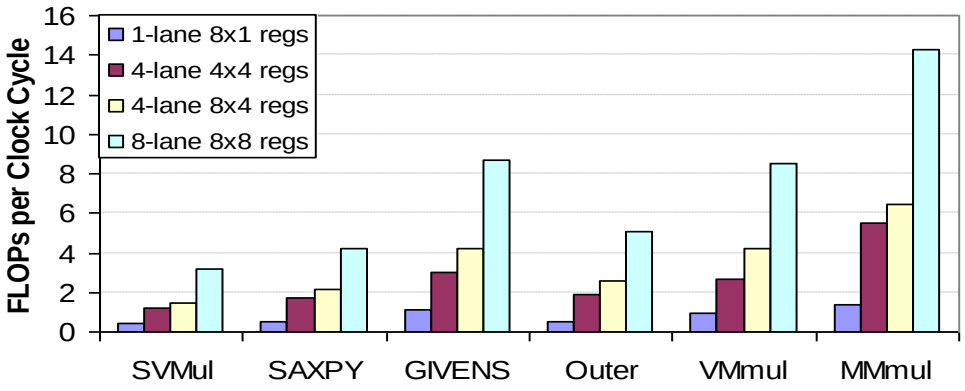


Figure 12: Scalability of Mat-Core architecture on linear algebra kernels.

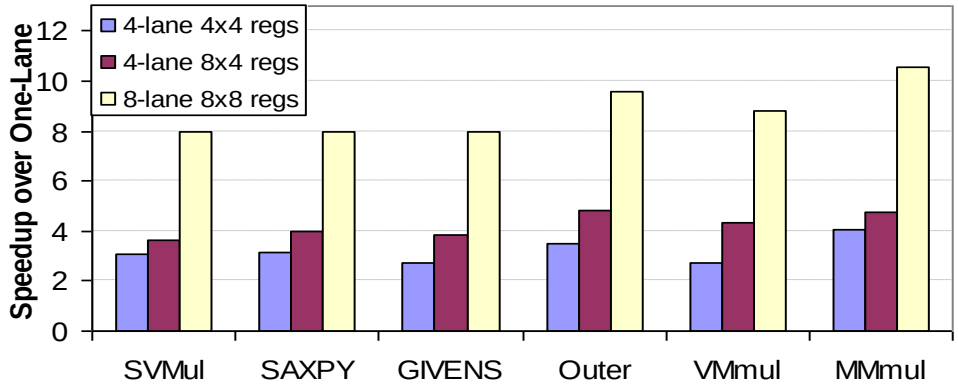


Figure 13: Speedup Multiple-lane Mat-Core over one-lane on linear algebra kernels.

5.2 Performance of DCT/IDCT on Mat-Core Processor

The performances in FLOPs per clock cycle of both DCT and IDCT on Mat-Core processor are similar. This is due to the similarity in the computation of the two algorithms of DCT and IDCT. In addition, processing the main operations $A \times M$, $A \times M^T$, and $M^T \times A$ on Mat-Core takes the same number of clock cycles. The number of FLOPs can be easily calculated from the algorithm as $4*8^3*(n/8)^2$ or $32n^2$ on $n \times n$ matrix and 8×8 DCT block. Moreover, the number of clock cycles on Mat-Core is calculated by our cycle accurate model, which is constructed using SystemC [3].

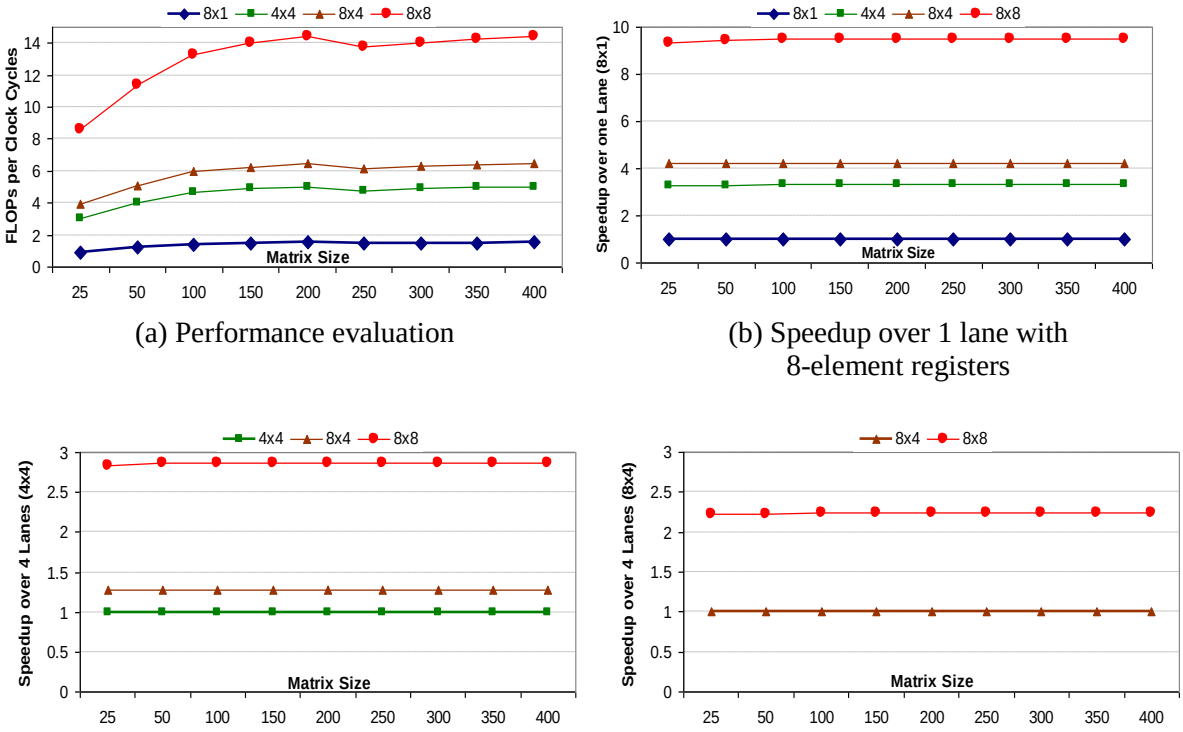
The ideal performances for DCT/IDCT are 2, 8, 8 and 16 FLOPs per clock cycle on Mat-Core-8×1, Mat-Core-4×4, Mat-Core-8×4, and Mat-Core-8×8, respectively. This is because each lane has one *MAC* pipeline, which represents two FLOPs (multiply and add). Figure 14-a shows the performance evaluation of the Mat-Core (four versions discussed in Section 2) on images with sizes 25×25, 50×50, 100×100, …, and 400×400. The maximum performances achieved are 1.5, 5, 6.4 and 14.4 FLOPs/cycle for Mat-Core-8×1, Mat-Core-4×4, Mat-Core-8×4, and Mat-Core-8×8, which represents 75%, 62.5%, 80%, and 90% of the ideal values, respectively. The saw tooth behavior in the

performance (see Figure 14-a) results from padding the image sizes to be multiple of eight. The effect of padding is large on small image sizes (25×25). As the image size increases, the effect of padding decreases until decaying.

Figure 14-b shows the speedup of Mat-Core-4×4, Mat-Core-8×4, and Mat-Core-8×8 over Mat-Core-8×1. Increasing the number of parallel lanes from one to four and then to eight results in speeding up the execution of DCT and IDCT by factors of 4.2, and 9.5, respectively. This indicates the scalability of Mat-Core architecture. Moreover, the enhancement in performance of DCT and IDCT on Mat-Core-8×4 and Mat-Core-8×8 over Mat-Core-4×4 (see Figure 14-c) is because the larger matrix blocks (8×4 or 8×8) amortize the pipelines latencies better than small blocks (4×4). In addition, increasing the number of parallel lanes from four to eight (see Figure 14-d) speeds up the execution of DCT and IDCT more than twice. Increasing the number of parallel lanes and enlarging the matrix registers results in reducing the dependency and gives more chance to reuse the data, which leads to improved performance.

5.3 Performance of Image Registration Kernels on Mat-Core Processor

The number of FLOPs in affine transformation can be easily calculated from Equation 4 as 2*(4*4*n) or 32n, where n is the number of vertices. The ideal performances for affine transformation are 2, 8, 8 and 16 FLOPs per clock cycle on Mat-



(a) Performance evaluation

(b) Speedup over 1 lane with 8-element registers

(c) Speedup over 4 lanes with 4×4 matrix registers

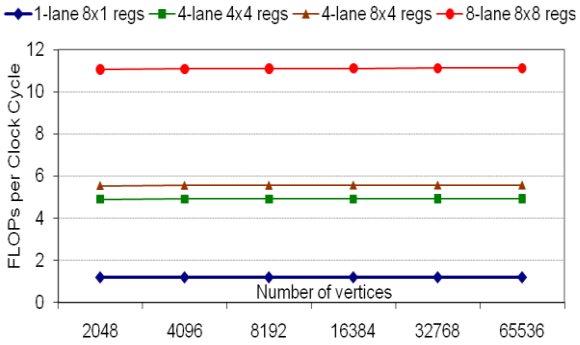
(d) Speedup over 4 lanes with 8×4 matrix registers

Figure 14: Performance evaluation of DCT and IDCT on scalable Mat-Core processor.

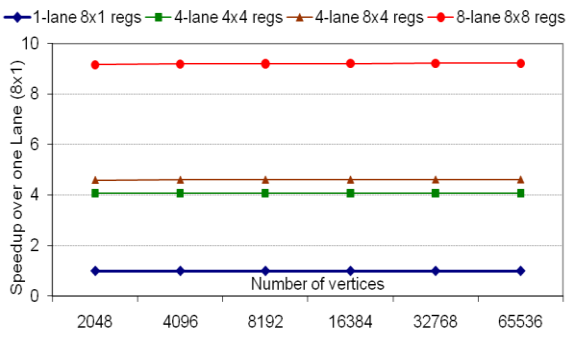
Core-8×1, Mat-Core-4×4, Mat-Core-8×4, and Mat-Core-8×8, respectively, because each lane has a *MAC* pipeline, which represents two FLOPs (multiply and add).

Figure 15-a shows the performance scalability of affine transformation on the Mat-Core versions on volume vertices 2k, 4k, 8k, . . . , and 64k, where k = 1024, versus FLOPs per clock cycle. The maximum performances achieved are 1.2, 4.9, 5.6 and 11.2 FLOPs/cycle for Mat-Core-8×1, Mat-Core-4×4, Mat-Core-8×4, and Mat-Core-8×8, respectively. The percentage of the maximum performance in the version with 8 lanes and matrix register size 8×8 (Mat-Core8×8) is 70% of the ideal value. This percentage is smaller than what achieved in DCT (90%) because affine transformation is less computationally intensive ($32n$ FLOPs) than DCT ($32n^3$ FLOPs). Mat-Core performance is higher in computationally intensive kernels rather than memory intensive kernels.

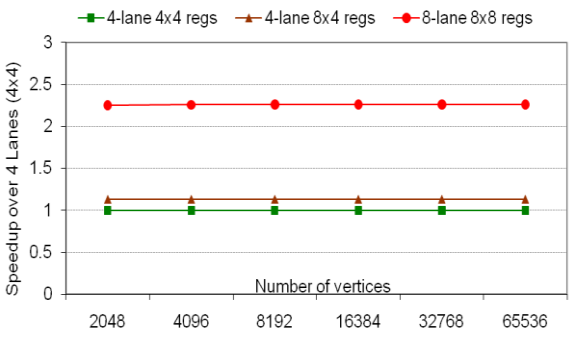
Figure 15-b shows the speedup of Mat-Core-4×4, Mat-Core-8×4, and Mat-Core-8×8 over Mat-Core-8×1. Increasing the number of parallel lanes from one to four and then to eight results in speeding up the execution of affine transformation by factors of 4.6, and 9.2, respectively. This indicates the scalability of Mat-Core architecture. Moreover, the enhancement in performance of affine transformation on Mat-Core-8×4 and Mat-Core-8×8 over Mat-Core-4×4 (see Figure 15-c) is because the larger matrix blocks (8×4 or 8×8) amortize the pipelines latencies better than small blocks (4×4). In addition, increasing the number of parallel lanes from four to eight (see Figure 15-d) speeds up the execution of affine transformation exactly twice.



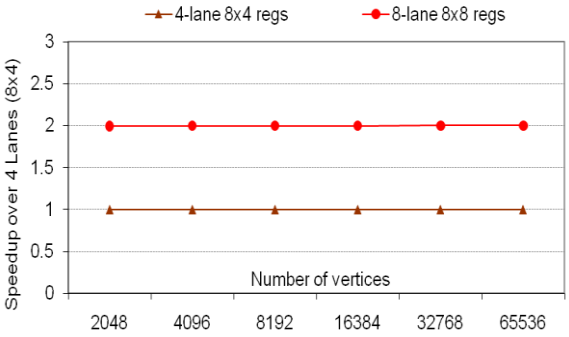
(a) Performance evaluation



(b) Speedup over 1 lane with 8-element registers



(c) Speedup over 4 lanes with 4×4 matrix registers



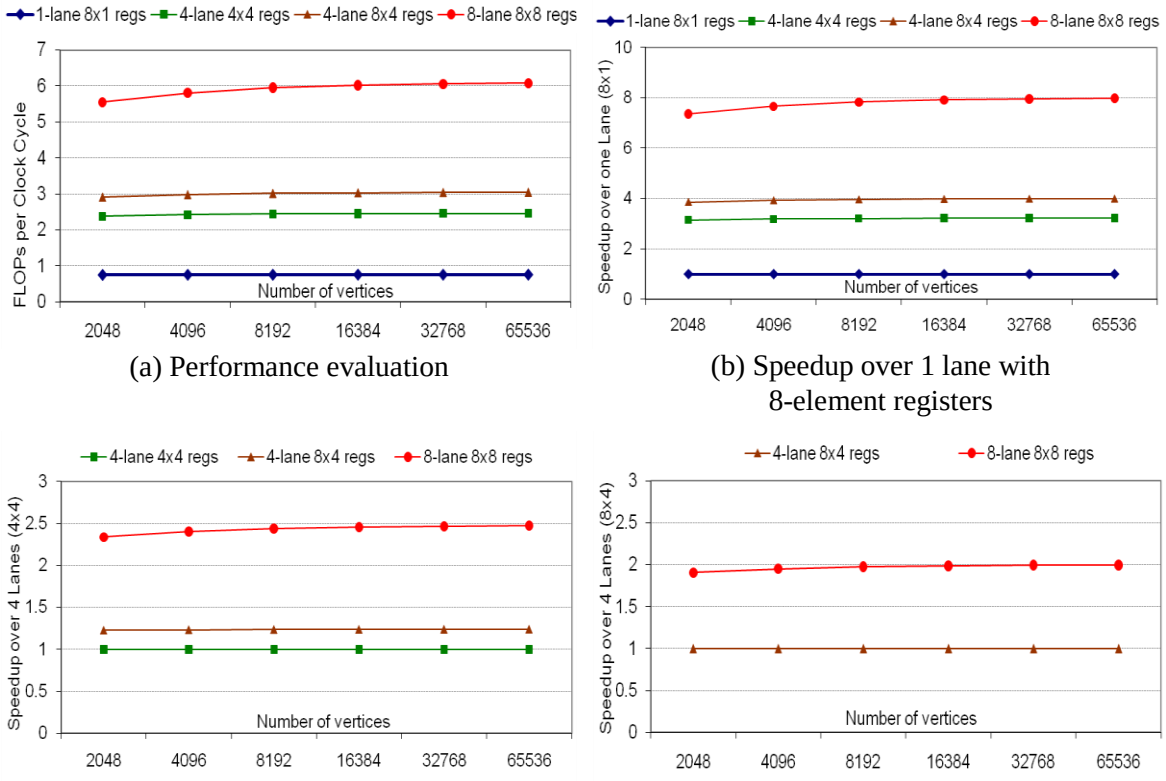
(d) Speedup over 4 lanes with 8×4 matrix registers

Figure 15. Performance evaluation of 3D affine transformation on scalable Mat-Core processor

The number of FLOPs in SAD can be easily calculated from Equation 5 as 2**n*-1, where n is the number of vertices. The ideal performances for SAD are 1, 4, 4 and 8 FLOPs per clock cycle on Mat-Core-8×1, Mat-Core-4×4, Mat-Core-8×4, and Mat-Core-8×8, respectively. This is because each lane has one *adder* pipeline, which is used by SAD kernel. The *MAC* pipeline is used only one time at the end of calculation. Since the number of vertices *n* is large, the *MAC* pipeline is ignored in calculation.

Figure 16-a shows the performance scalability of the SAD on the Mat-Core versions on volume vertices 2k, 4k, 8k, . . ., and 64k, where k = 1024, in FLOPs per clock cycle. The maximum performances achieved are 0.76, 2.46, 3 and 6.1 FLOPs/cycle for Mat-Core-8×1, Mat-Core-4×4, Mat-Core-8×4, and Mat-Core-8×8, which represents 76%, 61. 5%, 75%, and 76.25% of the ideal values, respectively.

Figure 16-b shows the speedup of Mat-Core-4×4, Mat-Core-8×4, and Mat-Core-8×8 over Mat-Core-8×1. Increasing the number of parallel lanes from one to four and then to eight results in speeding up the execution of SAD by factors of 4 and 8, respectively. This indicates the scalability of Mat-Core architecture. Moreover, the enhancement in performance of SAD on Mat-Core-8×4 and Mat-Core-8×8 over Mat-Core-4×4 (see Figure 16-c) is because the larger matrix blocks (8×4 or 8×8) amortize the pipelines latencies better than small blocks (4×4). In addition, increasing the number of parallel lanes from four to eight (see Figure 16-d) speeds up the execution of SAD twice on large number of vertices.



(a) Performance evaluation

(b) Speedup over 1 lane with 8-element registers

(c) Speedup over 4 lanes with 4×4 matrix registers

(d) Speedup over 4 lanes with 8×4 matrix registers

Figure 16. Performance evaluation of SAD on scalable Mat-Core processor

6. CONCLUSION

This paper has described the implementation of DCT/IDCT and image registration on Mat-Core processor. Moreover, it investigated the scalability of Mat-Core architecture with different number of parallel lanes (one, four, and eight) and variable matrix register size (4×4 and 8×4). Thus, four versions of Mat-Core processor (Mat-Core-8×1, Mat-Core-4×4, Mat-Core-8×4, and Mat-Core-8×8) have been implemented using SystemC as a cycle accurate models.

Our results show that scaling the Mat-Core processor from one lane with 8×1 vector registers to four lanes with 4×4 matrix registers improves the performance by factors of 3.1, 3.2, 2.75, 3.5, 2.75, 4.1, 3.3, 4.1, and 3.2 on scalar-vector multiplication, SAXPY, Givens, rank-1 update, vector-matrix multiplication, matrix-matrix multiplication, DCT, 3D affine transformation, and SAD, respectively. Moreover, increasing the number of lanes from one to eight with 8×8 matrix registers on the same kernels improves the performance by factors of 7.94, 7.95, 7.95, 9.6, 8.82, 10.6, 9.5, 9.2, and 8, respectively. In addition, scaling matrix registers from 4×4 to 8×4 improves the performance on the same kernels by factors of 1.2, 1.3, 1.4, 1.4, 1.6, 1.2, 1.3, 1.1, and 1.25, respectively. That is because the larger matrix blocks (8×4) amortize the pipelines latencies better than small blocks (4×4). The percentage of the maximum performance in the version with 8 lanes and matrix register size 8×8 (Mat-Core-8×8) is 90% of the ideal value for DCT and matrix-matrix multiplication. The speedups of the execution of the kernels from different application fields on Mat-Core-8×4, and Mat-Core-8×8 over Mat-Core-8×1 are 3.6x-4.8x and 7.94x-10.6x, respectively.

REFERENCES

[1] 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.

[2] M. Soliman, "Mat-Core: A Decoupled Matrix Core Extension for General-Purpose Processors," *Neural, Parallel and Scientific Computations*, Dynamic Publishers, Atlanta, USA, ISSN 1061-5369, Vol. 19, No. 2, 2011, pp. 91-110.

[3] M. Soliman and A. Al-Junaid "SystemC Implementation and Performance Evaluation of a Decoupled General-Purpose Matrix Processor," *Parallel Processing Letter (PPL),* World Scientific Publishing Company, June 2010, pp. 103-121.

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

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

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

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

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

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

[10] G. Moore, Cramming more Components onto Integrated Circuits, Electronics, Vol. 38, No. 8, 1965.

[11] M. Soliman and A. Al-Junaid, "Codevelopment of Multi-level ISA and hardware for an efficient matrix processor," *Proc. IEEE International Conference on Computer Engineering & Systems (ICCES'09*), Cairo, Egypt, December 2009, pp. 211-217.

[12] 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.

[13] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, 5th Edition, 2011.

[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] N. Ahmed, T. Natarajan, and K.R. Rao, "Discrete Cosine Transform," *IEEE Transaction on Computers*, vol. C-23, pp. 90-93, Jan 1974.

[17] K. R. Rao and P.C. Yip, *The Transform and Data Compression Handbook*, CRC Press, 2001.

[18] K. R. Rao and J. J. Hwang, *Techniques and Standards for Image, Video and, Audio Coding*, Upper Saddle River, NJ: Prentice Hall, 1996.

[19] B. Zitova and J. Flusser, "Image Registration Methods: a Survey," *Image and Vision Computing*, Vol. 21, No. 11, 2003, pp. 977-1000.

[20] V. Hajnal, D. Hill and D. Hawkes, *Medical Image Registration*, CRC Press, New York, 2001.

[21] A. Maintz and M. Viergever, A Survey of Medical Image Registration, *Medical Image Analysis*, Vol. 2, No. 1, 1998, pp. 1-36.

[22] T. Yoo, *Insight into Images: Principles and Practice for Segmentation, Registration, and Image Analysis*, A K Peters, Ltd, Massachusetts, 2004.

[23] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd Edition, Cambridge University Press, 1992.

[24] S. Buss, *3-D Computer Graphics: A Mathematical Introduction with OpenGL*, Cambridge University Press, New York, USA, 2003.

[25] M. Agoston, *Computer Graphics and Geometric Modeling: Implementation and Algorithms*, Springer-Verlag, London Limited, 2005.

[26] F. Bensaali, A. Amira, and A. Bouridane, "Accelerating Matrix Product on Reconfigurable Hardware for Image Processing Applications," *IEE Proceedings on Circuits Devices Systems*, Vol. 152, No. 3, June 2005, pp. 236-246.

[27] A. Watt, *3-D computer graphics*, Addison–Wesley, 2000.

[28] J. Fitzpatrick, J. West, and C. Maurer, "Predicting Error in Rigid-Body Point-Based Registration", *IEEE Transactions on Medical Imaging*, Vol. 17, No. 5, 1998, pp. 694-702.

[29] S. Umeyama, "Least-squares Estimation of Transformation Parameters between Two Point Patterns", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 13, No. 4, 1991, pp. 376-380.

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