University of Maryland Inistitute for Advanced Computer Studies Technical Report
UMIACS-TR-2006-42

Parallelization of the SSCA#3 Benchmark on the RAW Processor

Meng-Ju Wu and Donald Yeung
Department of Electrical and Computer Engineering
University of Maryland at College Park, Maryland 20742
mjwu@umd.edu, yeung@eng.umd.edu

ABSTRACT

The MIT Raw machine provides a point-to-point interconnection network for transferring register values between tiles. The programmer schedules the network communication for each tile by himself/herself and guarantees the correctness. It is not easy to parallelize benchmarks by hand for all possible tile configurations on the Raw processor. To overcome this problem, we develop a communication library and a switch code generator to create the switch code for each tile automatically. We implement our techniques for the SSCA#3 (SAR Sensor Processing, Knowledge Formation) benchmark, and evaluate the parallelism on a physical Raw processor. The experimental results show the SSCA#3 benchmark has dense matrix operations with abundant parallelism. Using 16 tiles, the 'SAR image formation' procedure achieves a speedup of 13.86, and the speedup of the 'object detection' procedure is 9.98.

1 Introduction

The MIT Raw microprocessor[1] fully exposes the low-level details of the underlying hardware architecture to the compiler and programmer. The RAW processor divides the silicon chip into 16 identical and programmable tiles. Each tile contains static and dynamic routers[2]. Communication signals travel across each tile in one cycle, and the low-level inter-tile communication is schedulable. Program complexity increases while mapping large applications on the Raw machine. For example, to avoid deadlock, to make sure all tiles obtain the correct data, and to operate the network efficiently require significant effort from the programmer.

The Raw compiler[3] tries to simultaneously hide the complexity from the programmer and preserve the memory system correctness and efficiency. This advantage is also a disadvantage. For the compiler, it is difficult to consider the whole application and analyze the communication cost across large regions of computation. Correctness and performance debugging is not easy when employing multiple tiles if the programmer only relies on the compiler. On the other hand, while the programmer and algorithm designer have full understanding of the application, it is tedious and error-prone for them to perform the compiler's tasks by hand.

In this report, we provide a programmer-assisted method to reduce the complexity of parallelizing the application on the Raw architecture. The programmer decides how to parallelize the application and route the communication. To assist the programmer, we built a communication library and a switch code generator to create the switch code for each router automatically. The HPCS SSCA#3 (SAR Sensor Processing, Knowledge Formation) benchmark[4] is our objective.

The rest of this report is organized as follows. Section 2 describes background on the SSCA#3 benchmark. Section 3 discusses the computation kernels and how to parallelize them. Section 4 explains the method used to create the tile code and switch code. Section 5 describes our experimental results. Section 6 summarizes and concludes our work.

2 SSCA#3 Benchmark Background

The SSCA#3 (SAR Sensor Processing, Knowledge Formation) benchmark defines a generalized sensor processing procedure consisting of a front-end sensor processing stage and a back-end knowledge formation stage. Figure 1 illustrates the basic block diagram of SSCA#3. The front-end sensor processing stage consists of Synthetic Aperture Radar (SAR) image formation and streaming image storage stage. The back-end knowledge formation stage consists of an image set retrieval and pixelated-letter recognition. This report focuses on the challenge of parallelizing these computation kernels on the Raw machine. We skip the file I/O part.

There are three computational kernels:

1. Scalable data and template generator (SDG), used to create the echo radar data that is necessary to produce the needed SAR image.
2. SAR image formation and letter template insertion, used to rebuild the SAR image and populate the image with letter templates for stage 2 back-end knowledge formation.
3. Pixelated-letter detection, used to find the region of interest (ROI) and recognize the letter.

The mathematical fundamentals associated with the SSCA#3 benchmark is described in the book 'Soumekh, Mehrdad, Synthetic Aperture Radar Signal Processing with Matlab Algorithms, Wiley, 1999'. The following sections will not give the mathematical description of this benchmark. Instead, we discuss the high-level flow of the benchmark.

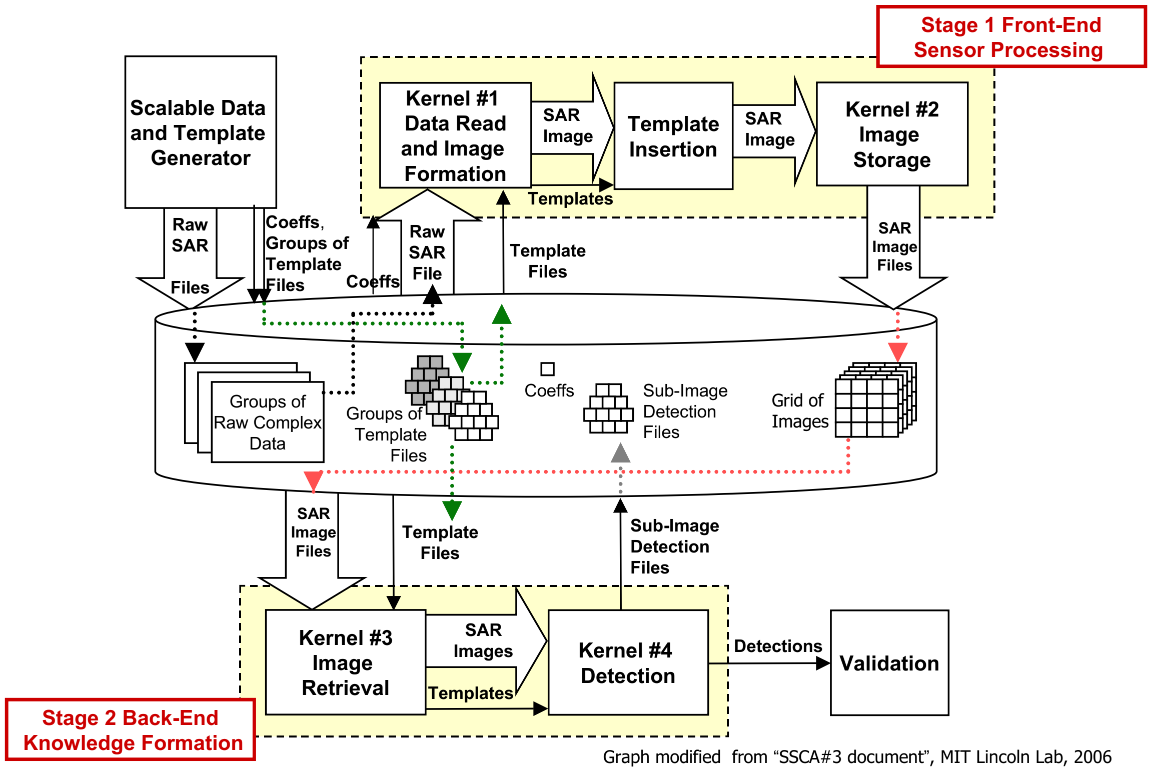

Stage 1 Front-End Sensor Processing
Scalable Data and Template Generator
Kernel #1 Data Read and Image Formation
SAR Image
Template Insertion
SAR Image
Kernel #2 Image Storage
Templates
Raw SAR Files
Coeffs, Groups of Template Files
Coeffs
Raw SAR File
Template Files
SAR Image Files
Groups of Raw Complex Data
Groups of Template Files
Coeffs
Sub-Image Detection Files
Grid of Images
SAR Image Files
Template Files
Sub-Image Detection Files
Kernel #3 Image Retrieval
SAR Images
Templates
Kernel #4 Detection
Detections
Validation
Stage 2 Back-End Knowledge Formation
Graph modified from "SSCA#3 document", MIT Lincoln Lab, 2006

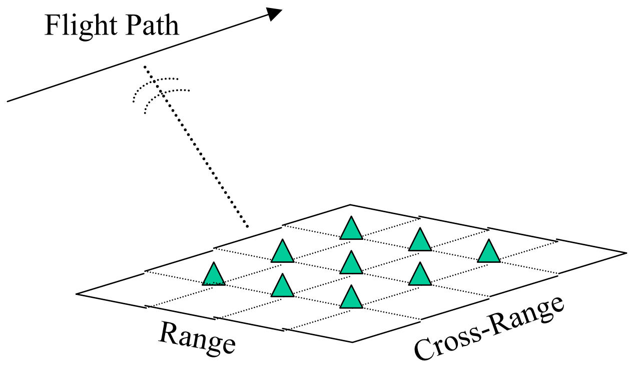
Figure 1: SSCA#3 Benchmark Architecture.

2.1 Scalable Data and Template Generator (SDG)

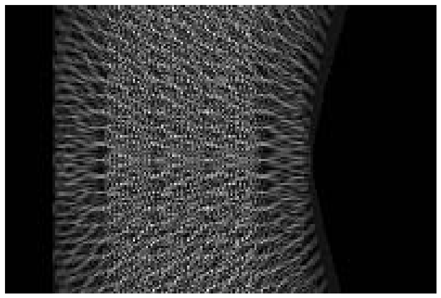
The Synthetic Data Generator creates scalable raw SAR data. This kernel simulates an airborne phased array radar moving over some terrain. The radar captures the echo from the swath along the flight path and produces the synthetic data.

The terrain is simulated by placing the elements regularly on the ground. Each sensor element and each echo pulse creates a synthetic sample. Figure 2 presents this simplified scenario.

The raw SAR data and sensor elements are scalable and controlled by input parameters. There are three raw SAR data sizes: 160x438, 320x754, and 480x1072. The number of sensor elements depends on how dense the terrain is. In this report, the raw SAR data size is 160x438, and the number of sensor elements is 24, arranged in a 6x4 grid. The size of the reconstructed SAR image is 382x266.

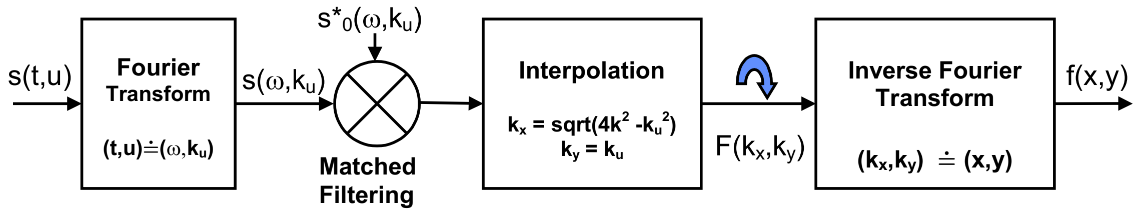


(a) Sensor elements



(b) Raw SAR data

Figure 2: Radar captures echoes and generates the synthetic data.



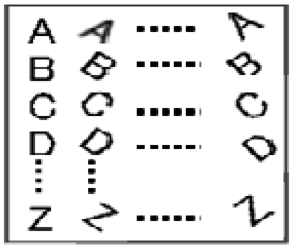
$s(t,u)$	Echoed SAR signal from target area
$s(\omega,k_u)$	Two-dimensional Fourier transform of SAR signal with respect to fast-time and slow-time
$s^*_0(\omega,k_u)$	Complex conjugate of two-dimensional Fourier transform of reference SAR signal with respect to fast-time and slow-time
$F(k_x,k_y)$	Two-dimensional Fourier transform of target function
$f(x,y)$	Resulting target function in time domain

Figure 3: Block diagram of a spotlight SAR digital reconstruction algorithm via spatial frequency domain interpolation[5][6].

2.2 SAR Image Formation and Letter Template Insertion

2D Fourier Matched Filtering and Interpolation[5] is used as the wavefront spotlight SAR image reconstruction method. The central idea is to filter the transmitted SAR signal against the returned signal in the frequency domain. Then, convert the coordinate representation from polar coordinates to rectangular coordinates by resampling the output of the match filter. The final step is to use the inverse Fourier transform to transfer the result from frequency-domain to human readable spatial-domain. Figure 3 shows the block diagram of the spotlight SAR digital reconstruction algorithm[5][6].

A set of capital letters at clock-wise rotations are added onto the rebuilt SAR image (Figure 4a). The group of letters is made in advance and does not occupy any computation time.


(a) Capital letters set

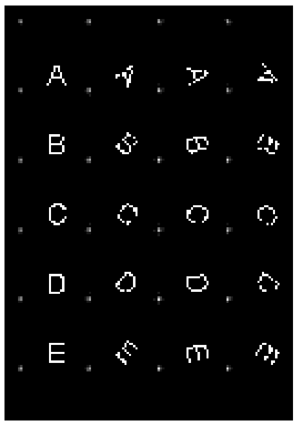

(b) SAR image

Figure 4: Insert the capital letters into the SAR image to simulate the terrain and targets.

These letter templates simulate the targets on the ground, and the recognition procedure in stage2 will detect these letters. The letters are placed at preselected locations to avoid overlapping with the "peak" in the SAR image. In other words, we place the letters in the valleys (Figure 4b). The pixel-value of the letter is half the maximal power level of the SAR image.

In the original benchmark, each letter is inserted into the SAR image with a 50% probability. This means the created SAR image will be different every time. Because we focus on parallelizing the computational kernels, we create only one SAR image with all possible letter positions populated. This circumstance does not affect the parallelizaton method of previous computation because letter insertion is the last operation in stage 1. This operation occupies a very small portion of overall execution time, so how the letter is populated makes insignificant inference for stage 1 computation.

2.3 Pixelated-Letter Detection

This kernel simulates a simplified automatic target recognition scheme. The algorithm scans pixel by pixel to mark the region of interest (ROI). Once the algorithm finds one ROI, a subimage will be extracted and correlated against all letter templates. A maximal likelihood decision chooses the letter that can give the maximal correlation value.

The original recognition kernel in SSCA#3 benchmark uses the difference of successive image pairs in a sequence to recognize the newly appeared target. Because we only create one image, we subtract the letter-populated SAR image from an SAR image that has no letters in it. The result image has full-populated letters without background. Our parallelization algorithm distributes this image to tiles evenly, and each tile uses its detection kernel to find

Table 1: SSCA#3 Stage1 Computation Kernels.

Computation Kernel	*Numbers*
1-D vector generation(1xN)	5
1-D vector generation(Nx1)	5
Matrix element-by-element operation	9
Discrete Fourier transform (vector)	1
Discrete Fourier transform (matrix)	3
Inverse discrete Fourier transform (matrix)	3
Matrix interpolation	1
Find maximum/minimum values in matrix	2

Table 2: SSCA#3 Stage2 Computation Kernels.

Computation Kernel	*Numbers*
Find maximum/minimum values in matrix	1
Matrix element-by-element operation	3
Two-dimensional correlation	1

the ROIs and to detect the letters in its partition of the image.

The computation time is related with the number of letters, so this case generates the maximum computation for the detection kernel. If the number of letters is changed, each tile may have different number of letters for detection. The overall computation time depends on the tile that has the most letters. Section 5 gives the detailed discussion.

3 Computation Kernels

The computational part of SSCA#3 is made up of nine different types of computation kernels. We list these kernels in Table 1 and Table 2. These kernels are all array and matrix operations. The dense matrix operations yield significant parallelism. This section gives an overview of each kernel and the parallelization method.

3.1 Parallelization Method

The main idea of our parallelization method is spreading the computation across the tiles and using the static network to share data among tiles. Data locality is an important consideration, because achieving high performance requires good load balance and efficient communication. Exploiting data locality can also help the programmer to debug and tune the program easily. Our parallelization method considers four key points:

1. The data partitioning should maximize physical locality.
2. Data dependencies must be observed.
3. Share the data in an efficient way.
4. The computation should be load balanced across tiles.

3.2 Computation Kernels

The following sections describe each computation kernel in array and matrix form, and illustrate how to achieve the parallelism on the Raw architecture. For simplicity, we use the 4x4 tile shapes to illustrate the computation kernels.

$$\begin{bmatrix} [tile0] & [tile1] & [tile2] & [tile3] \\ [tile4] & [tile5] & [tile6] & [tile7] \\ [tile8] & [tile9] & [tile10] & [tile11] \\ [tile12] & [tile13] & [tile14] & [tile15] \end{bmatrix}$$

3.2.1 1-D vector generation(1xN)

Given a 1xN array A, the goal is to distribute the elements and computations evenly among tiles. The 1xN array is distributed over columns. The first step is to divide array A into four subarrays. The first column has the first N/4 elements, the second column has the next N/4 elements, and so on. Then each column divides its subarray into four smaller subarrays, and each tile will have N/16 elements.

$$\text{A} = \left[\begin{array}{cccc} A_0 & A_1 & ... & A_{N-1} \end{array}\right]_{1\times N} \qquad , N = 400$$

$$= \left[\left[\begin{array}{ccc} A_0 & ... & A_{99} \end{array}\right] \left[\begin{array}{ccc} A_{100} & ... & A_{199} \end{array}\right] \left[\begin{array}{ccc} A_{200} & ... & A_{299} \end{array}\right] \left[\begin{array}{ccc} A_{300} & ... & A_{399} \end{array}\right] \right]$$

$$=> \left[\begin{array}{cccc} \left[\begin{array}{ccc} A_0 & ... & A_{24} \end{array}\right] & \left[\begin{array}{ccc} A_{100} & ... & A_{124} \end{array}\right] & \left[\begin{array}{ccc} A_{200} & ... & A_{224} \end{array}\right] & \left[\begin{array}{ccc} A_{300} & ... & A_{324} \end{array}\right] \\ \left[\begin{array}{ccc} A_{25} & ... & A_{49} \end{array}\right] & \left[\begin{array}{ccc} A_{125} & ... & A_{149} \end{array}\right] & \left[\begin{array}{ccc} A_{225} & ... & A_{249} \end{array}\right] & \left[\begin{array}{ccc} A_{325} & ... & A_{349} \end{array}\right] \\ \left[\begin{array}{ccc} A_{50} & ... & A_{74} \end{array}\right] & \left[\begin{array}{ccc} A_{150} & ... & A_{174} \end{array}\right] & \left[\begin{array}{ccc} A_{250} & ... & A_{274} \end{array}\right] & \left[\begin{array}{ccc} A_{350} & ... & A_{374} \end{array}\right] \\ \left[\begin{array}{ccc} A_{75} & ... & A_{99} \end{array}\right] & \left[\begin{array}{ccc} A_{175} & ... & A_{199} \end{array}\right] & \left[\begin{array}{ccc} A_{275} & ... & A_{299} \end{array}\right] & \left[\begin{array}{ccc} A_{375} & ... & A_{399} \end{array}\right] \end{array}\right]$$

3.2.2 1-D vector generation(Nx1)

The Nx1 array is spread over rows. First, we divide the array B into four subarrays. The first row receives the first N/4 elements, the second row receives the next N/4 elements, and so on. Then each row divides its subarray into four smaller subarrays, and each tile will have N/16 elements.

$$\text{B} = \left[\begin{array}{cccc} B_0 & B_1 & ... & B_{N-1} \end{array}\right]^T_{1\times N} \qquad , N = 400$$

$$= \left[\left[\begin{array}{ccc} B_0 & ... & B_{99} \end{array}\right] \left[\begin{array}{ccc} B_{100} & ... & B_{199} \end{array}\right] \left[\begin{array}{ccc} B_{200} & ... & B_{299} \end{array}\right] \left[\begin{array}{ccc} B_{300} & ... & B_{399} \end{array}\right] \right]^T$$

$$=> \left[\begin{array}{cccc} \left[\begin{array}{ccc} B_0 & ... & B_{24} \end{array}\right] & \left[\begin{array}{ccc} B_{25} & ... & B_{49} \end{array}\right] & \left[\begin{array}{ccc} B_{50} & ... & B_{74} \end{array}\right] & \left[\begin{array}{ccc} B_{75} & ... & B_{99} \end{array}\right] \\ \left[\begin{array}{ccc} B_{100} & ... & B_{124} \end{array}\right] & \left[\begin{array}{ccc} B_{125} & ... & B_{149} \end{array}\right] & \left[\begin{array}{ccc} B_{150} & ... & B_{174} \end{array}\right] & \left[\begin{array}{ccc} B_{175} & ... & B_{199} \end{array}\right] \\ \left[\begin{array}{ccc} B_{200} & ... & B_{224} \end{array}\right] & \left[\begin{array}{ccc} B_{225} & ... & B_{249} \end{array}\right] & \left[\begin{array}{ccc} B_{250} & ... & B_{274} \end{array}\right] & \left[\begin{array}{ccc} B_{275} & ... & B_{299} \end{array}\right] \\ \left[\begin{array}{ccc} B_{300} & ... & B_{324} \end{array}\right] & \left[\begin{array}{ccc} B_{325} & ... & B_{349} \end{array}\right] & \left[\begin{array}{ccc} B_{350} & ... & B_{374} \end{array}\right] & \left[\begin{array}{ccc} B_{375} & ... & B_{399} \end{array}\right] \end{array}\right]$$

3.2.3 Matrix element-by-element operation

This operation is not a cross product, dot product, nor matrix multiplication. Instead, this operation is performed element-by-element. Given a 1xN array A, an Mx1 array B, and an MxN matrix S, the operation uses the same A element that is in the same column and uses the same B element that is in the same row.

$$A = \left[\begin{array}{ccc} A_0, & ... & , A_{N-1} \end{array} \right]_{1\times N}$$

$$B = \left[\begin{array}{ccc} B_0, & ... & , B_{M-1} \end{array} \right]^T_{1\times M} = \left[\begin{array}{c} B_0 \\ B_1 \\ : \\ B_{M-1} \end{array} \right]_{M\times 1} \qquad S = \left[\begin{array}{ccc} S_{0,0} & ... & S_{0,N-1} \\ : & & : \\ S_{M-1,0} & .. & S_{M-1,N-1} \end{array} \right]_{M\times N}$$

$$f(S,A,B) = \left[\begin{array}{cccc} f(S_{0,0},A_0,B_0) & f(S_{0,1},A_1,B_0) & ... & f(S_{0,N-1},A_{N-1},B_0) \\ : & & & : \\ f(S_{M-1,0},A_0,B_{M-1}) & f(S_{M-1,1},A_1,B_{M-1}) & .. & f(S_{M-1,N-1},A_{N-1},B_{M-1}) \end{array} \right]_{M\times N}$$

Our parallelization tries to give each tile independent computations. The MxN matrix S is distributed to tiles evenly. If we use the approach described in the previous two sections to calculate A and B, then we need to use the static network to coordinate the data. The goal is to let tiles in the same column to have the same A elements and tiles in the same row to have the same B elements. Section 4.1 gives an example.

3.2.4 Discrete Fourier Transform (vector)

The discrete Fourier transform in one dimension can be expressed using the following equation:

$$X(n) = \sum_{k=0}^{N-1} x(k) e^{-jk2\pi n/N} \quad , \; n = 0..N-1$$

For this computation, each tile needs all array elements to complete the summation. As mentioned in section 3.2.3, the array can be divided for calculation and merged for element-by-element operation. After each tile gets all its elements, it can perform the DFT procedure. Each tile computes a portion of the N elements. For example, tile0 performs n=[0,(N/16)-1], and tile15 performs n=[15*N/16,N-1].

3.2.5 Discrete Fourier Transform and Inverse Discrete Fourier Transform (matrix)

The matrix discrete Fourier transform here is not a 2D DFT computation. Instead, this operation is computing the 1D DFT on one column (or one row) and repeating until all columns (or all rows) are finished.

$$DFT \quad X(n,l) = \left[\sum_{k=0}^{N-1} x(k,l)e^{-jk2\pi n/N}, n = 0..N-1\right] \quad, l = 0..M-1$$

$$IDFT \quad x(n,l) = \left[\frac{1}{N}\sum_{k=0}^{N-1} X(k,l)e^{jk2\pi n/N}, n = 0..N-1\right] \quad, l = 0..M-1$$

3.2.6 Matrix Interpolation

Interpolation is a method of constructing new data points from known data points. For example, the old matrix S is interpolated with matrix A to form the new matrix S_1. The equation can be expressed as follows:

$$S_1(k,l) = f(S,A), \quad k = 0..N-1, \quad l = 0..M-1$$

In this case, the data will be shared across boundaries. The operation acts like the matrix element-by-element operation described in Section 3.2.3. The data can be distributed for calculation and merged for the final result by using the static network.

3.2.7 Find Maximum/Minimum values in Matrix

This operation is very simple. Suppose the partitioning of matrix S follows Section 3.2.3, and we want to find the maximum value in matrix S. Then there are three steps:

1. Each tile finds the maximum value from its portion of matrix S.
2. Each tile sends its maximum value to the other 15 tiles.
3. Each tile finds the maximum value amongst these 16 values.

After completing these three steps, every tile will know the maximum value. The same approach is used to find the minimum value in a matrix.

3.2.8 Two-dimensional Correlation

2D correlation is equivalent to two-dimensional convolution with the filter matrix rotated 180 degrees. Assuming S, P,and Q are matrices, the matrix form of convolution can be expressed as follows:

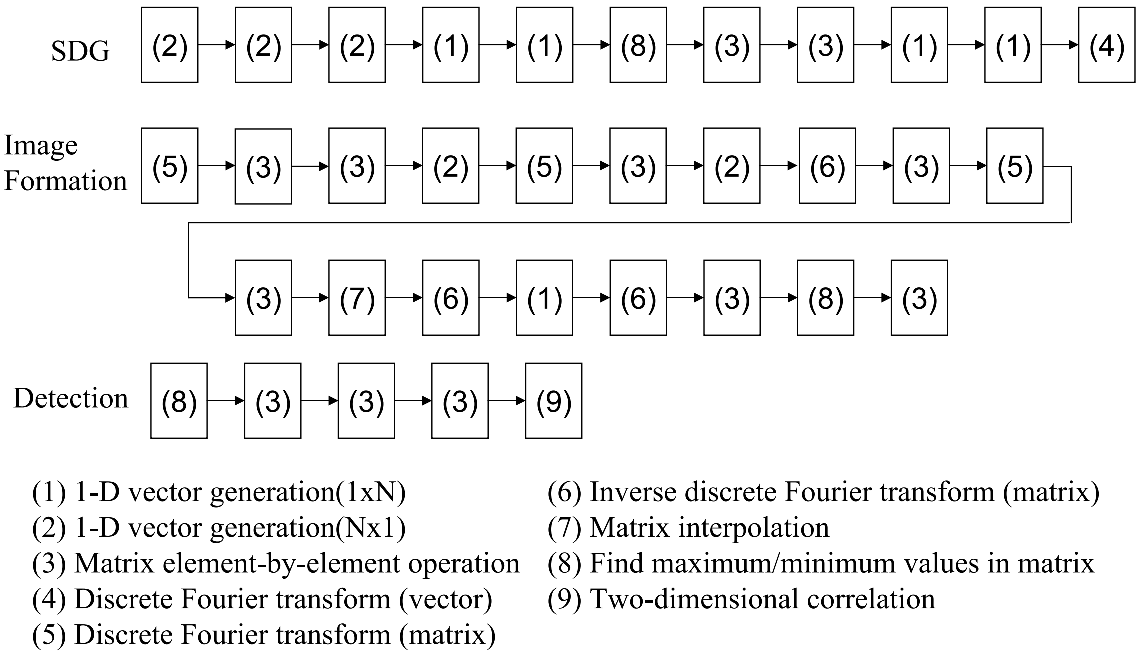

SDG
(2) (2) (2) (1) (1) (8) (3) (3) (1) (1) (4)
Image Formation
(5) (3) (3) (2) (5) (3) (2) (6) (3) (5)
(3) (7) (6) (1) (6) (3) (8) (3)
Detection
(8) (3) (3) (3) (9)
(1) 1-D vector generation(1xN)
(2) 1-D vector generation(Nx1)
(3) Matrix element-by-element operation
(4) Discrete Fourier transform (vector)
(5) Discrete Fourier transform (matrix)
(6) Inverse discrete Fourier transform (matrix)
(7) Matrix interpolation
(8) Find maximum/minimum values in matrix
(9) Two-dimensional correlation

Figure 5: Computation part of SSCA#3 represented by computation kernels.

$$S(i,j) = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} P(m,n) * Q(i-m, j-m)$$

This operation will cross partitioning boundaries, so each tile must share boundary data with their neighbors.

3.3 Combine Computation Kernels

Figure 5 illustrates the sequential computation kernels of the SSCA#3 computation. These kernels are parallelized by the methods described in Section 3.2. Data dependencies between kernels are considered to reduce communication.

4 Switch Code Generator

The physical RAW processor used in our experiments contains 16 tiles arranged in a 4x4 grid. Since valid RAW machine configurations must be rectangular, this yields 16 possible configurations (1x1, 1x2, 1x3, 1x4, 2x1, 2x2, 2x3, 2x4, 3x1, 3x2, 3x3, 3x4, 4x1, 4x2, 4x3, and 4x4). It is unreasonable to expect the programmer to write the tile and switch codes for every possible configuration. Hence, providing tools for automatically generating code for different configurations is critical.

Because the communication pattern is regular, it will enable us to write a route-plan file to describe the communication occurring in the static network and create the switch code for each tile from this route plan. In this section, we will discuss the static network communication, how to write the route-plan file, and how to let the route-plan file and tile code cooperate.

4.1 Communication in Static Network

In Section 3.2.3, we describe the array as being distributed on tiles for computation then synchronized to let the tiles in the same column or in the same row have the same data for the matrix element-by-element operation. Without loss of generality, we use a vector F and a 3x4 tile shape as an example to explain the exact operation.

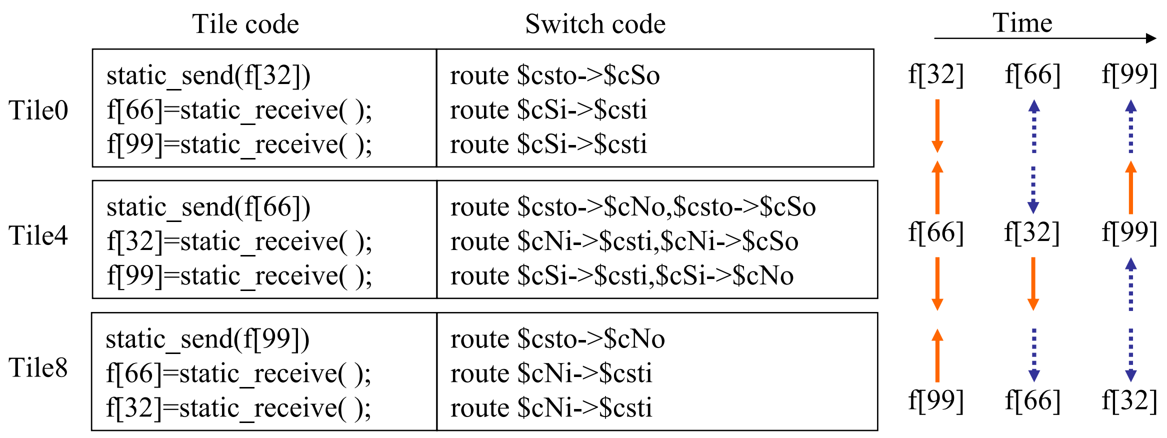
The vector F has 1x400 elements, so it is divided into 4 subarrays first (Section 3.2.1). Each column of Raw machine will have 100 elements (step1). Then each subarray distributes its 100 elements across 3 rows. Because 3 cannot evenly divide 100, the tile in the first row will take one more element than the other tiles. Tile0 has 34 elements, tile4 has 33 elements, and tile8 has 33 elements (step2). If there are 101 elements in the first column, then tile0 will have 34 elements, tile4 will have 34 elements, and tile8 will have 33 elements.

$$
\begin{aligned}
&\mathrm{F} = \begin{bmatrix} f_0 & f_1 & \cdots & f_{399} \end{bmatrix}_{1\times 400} \\
&(step1) => \begin{bmatrix} \begin{bmatrix} f_0 & \cdots & f_{99} \end{bmatrix} & \begin{bmatrix} f_{100} & \cdots & f_{199} \end{bmatrix} & \begin{bmatrix} f_{200} & \cdots & f_{299} \end{bmatrix} & \begin{bmatrix} f_{300} & \cdots & f_{399} \end{bmatrix} \end{bmatrix}
\end{aligned}
$$

$$
(step2) => \begin{bmatrix} \begin{bmatrix} f_0 & \cdots & f_{33} \end{bmatrix} & \begin{bmatrix} f_{100} & \cdots & f_{133} \end{bmatrix} & \begin{bmatrix} f_{200} & \cdots & f_{233} \end{bmatrix} & \begin{bmatrix} f_{300} & \cdots & f_{333} \end{bmatrix} \\ \begin{bmatrix} f_{34} & \cdots & f_{66} \end{bmatrix} & \begin{bmatrix} f_{134} & \cdots & f_{166} \end{bmatrix} & \begin{bmatrix} f_{234} & \cdots & f_{266} \end{bmatrix} & \begin{bmatrix} f_{334} & \cdots & f_{366} \end{bmatrix} \\ \begin{bmatrix} f_{67} & \cdots & f_{99} \end{bmatrix} & \begin{bmatrix} f_{167} & \cdots & f_{199} \end{bmatrix} & \begin{bmatrix} f_{267} & \cdots & f_{299} \end{bmatrix} & \begin{bmatrix} f_{367} & \cdots & f_{399} \end{bmatrix} \end{bmatrix}
$$

$$
\begin{aligned}
&\text{synchronize} \quad \text{elements} \quad \text{across} \quad \text{rows} \\
&(step3) => \begin{bmatrix} \begin{bmatrix} f_0 & \cdots & f_{99} \end{bmatrix} & \begin{bmatrix} f_{100} & \cdots & f_{199} \end{bmatrix} & \begin{bmatrix} f_{200} & \cdots & f_{299} \end{bmatrix} & \begin{bmatrix} f_{300} & \cdots & f_{399} \end{bmatrix} \\ \begin{bmatrix} f_0 & \cdots & f_{99} \end{bmatrix} & \begin{bmatrix} f_{100} & \cdots & f_{199} \end{bmatrix} & \begin{bmatrix} f_{200} & \cdots & f_{299} \end{bmatrix} & \begin{bmatrix} f_{300} & \cdots & f_{399} \end{bmatrix} \\ \begin{bmatrix} f_0 & \cdots & f_{99} \end{bmatrix} & \begin{bmatrix} f_{100} & \cdots & f_{199} \end{bmatrix} & \begin{bmatrix} f_{200} & \cdots & f_{299} \end{bmatrix} & \begin{bmatrix} f_{300} & \cdots & f_{399} \end{bmatrix} \end{bmatrix}
\end{aligned}
$$

After each tile finishes its portion of array F, the tiles in the same column will share their data, so the tiles in the same column will have the same data. In this example, tile0, tile4, and tile8 all have the same F elements, $[f_0 ... f_{99}]$(step3). Figure 6 shows how the data are



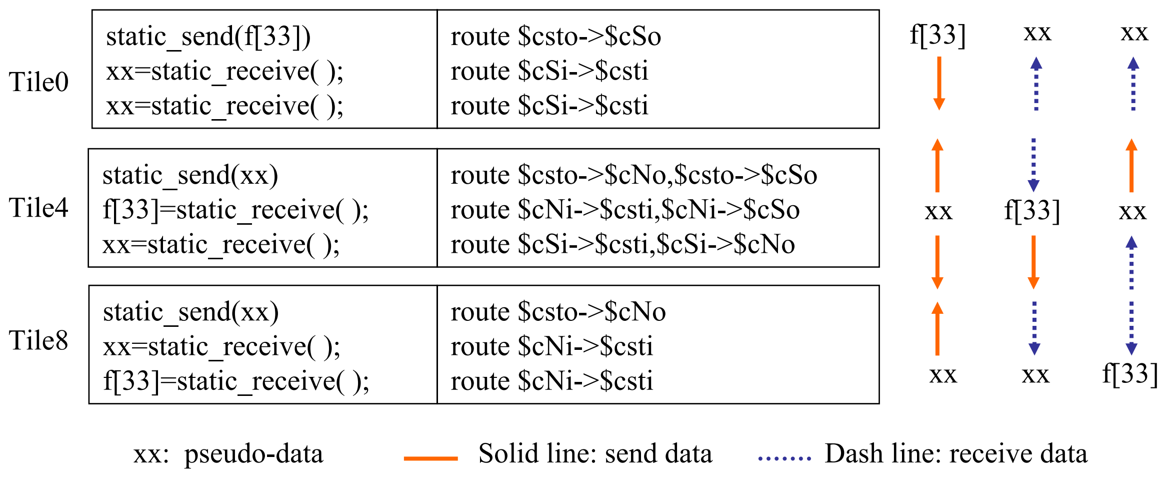
	Tile code	Switch code

Tile0	static_send(f[32])
	f[66]=static_receive( );
	f[99]=static_receive( );
		route $csto->$cSo
		route $cSi->$csti
		route $cSi->$csti

Tile4	static_send(f[66])
	f[32]=static_receive( );
	f[99]=static_receive( );
		route $csto->$cNo,$csto->$cSo
		route $cNi->$csti,$cNi->$cSo
		route $cSi->$csti,$cSi->$cNo

Tile8	static_send(f[99])
	f[66]=static_receive( );
	f[32]=static_receive( );
		route $csto->$cNo
		route $cNi->$csti
		route $cNi->$csti

(a) The 33rd iteration.



	Tile code	Switch code

Tile0	static_send(f[33])
	xx=static_receive( );
	xx=static_receive( );
		route $csto->$cSo
		route $cSi->$csti
		route $cSi->$csti

Tile4	static_send(xx)
	f[33]=static_receive( );
	xx=static_receive( );
		route $csto->$cNo,$csto->$cSo
		route $cNi->$csti,$cNi->$cSo
		route $cSi->$csti,$cSi->$cNo

Tile8	static_send(xx)
	xx=static_receive( );
	f[33]=static_receive( );
		route $csto->$cNo
		route $cNi->$csti
		route $cNi->$csti

(b) The 34th iteration.

Figure 6: Example of data synchronization.(a)The first 33 iterations are the same except for the index of f. (b) Introduce pseudo-data to simplify the communication pattern.

coordinated. The first 33 iterations are the same as Figure 6a except for the index of f. To simplify the communication pattern, we introduce pseudo-data. Because tile0 has one more element than tile4 and tile8, tile4 and tile8 will pretend that they have one extra data and send it (Figure 6b). The for-loop has a total of 34 iterations, and the tiles which receive the pseudo-data drop it.

The pseudo-data are not any special value. We provide a library that can calculate the exact index range of array on every tile. For example, tile0 knows the index range of array F on tile8 is from 67 to 99. At the 34th iteration, tile8 pretends to own f_{100} and sends any value to tile0. When tile0 receives this data, it knows the data is out of index range and drops it.

Table 3: Four syntaxes used in the route-plan file.

Syntax	*Comment*	*Example*
Tiles #NCol #NRow	Declare that there are #NRow × #NCol tiles.	Tiles 3 2
Loop 1 h	Communication among tiles in the same column	Section 4.1 Section 4.3
Loop 1 v	Communication among tiles in the same row	Section 4.3
2DLoop-begin 2DLoop-end	Nested loop: begin Nested loop: end	2DLoop-begin Loop 1 h 2DLoop-end

4.2 Route-Plan File

The regular communication patterns enable us to develop the 'switch code generator' and communication library to help parallelize the SSCA#3 benchmark on different tile-shapes. The switch code generator reads the route-plan file and creates the switch code of each tile automatically.

The 'route-plan' file describes the communication occurring in the static network. Table 3 shows four syntaxes used in the route-plan file. (1)'Tiles #NCol #NRow' declares the tiles shape, i.e. there are #NRow rows by #NCol columns tiles. (2)'Loop 1 h' means the data is a 1xN vector type, and the communication occurs among tiles that are in the same column. Section 4.1 gives an example about this communication type. (3)'Loop 1 v' means the data is an Nx1 vector type, and the communication occurs among tiles that are in the same row. (4)'2DLoop-begin' and '2DLoop-end' are for nested loop. The matrix can be viewed as the combination of several vectors. When synchronizing the elements of matrix across tiles, we need a nested for-loop statement.

4.3 Tile Code and Route-Plan File Cooperation

Figure 7a shows how the tile C code and the route plan file work. To keep the parallelization simple, there is only one tile code. Each tile uses *pid* to distinguish the array and matrix index ranges that belong to it and does the calculation in these index ranges. The 'initial_col()' and 'initial_row()' provided by our library calculate these index ranges automatically.

In Figure 7a, the four variables, x_begin, x_end, y_begin, and y_end, represent the index ranges. The programmer can change these four values freely to give partial calculation and communication - for example, to share the boundary data. We provide two communication functions: 'communication_dist_row_col_tile()' and 'communication_dist_col_row_tile()'. The first function means the data is distributed in row major and needs communication across tiles in the same row. The second function means that the data is distributed in column major and needs communication across tiles in the same column.

The route-plan file shows how to manage the communication between tiles. Vector A needs communication across columns, so it uses 'communication_dist_row_col_tile()' in tile C code and 'Loop 1 v' in the route-plan file. Vector B needs communication across rows, so it uses 'communication_dist_col_row_tile()' in tile C code and 'Loop 1 h' in the route-plan file. Matrix C needs communication across rows, and it uses '2D-Loop-begin' and '2D-Loop-end' syntax in the route-plan file. The created switch code for tile0 is also displayed in Figure 7a. The tile code shows that while using '2D-Loop-begin' and '2D-Loop-end' syntax, it will create nested iterations. So the switch code will be compacted.

Figure 7b gives a numerical example to show how the data localization is changed. One 200x1 vector A, one 1x300 vector B and one 200x300 matrix C are used in the 2x3 tile shape. There are three stages to initialize, compute, and distribute the elements of vector and matrix. Using vector A as an example, each tile calculates the index range of array A in the initialization stage. Then in the computation stage, tile0 calculates $[A_0..A_{33}]$, tile1 calculates $[A_{34}..A_{66}]$, tile2 calculates $[A_{67}..A_{99}]$, tile4 calculates $[A_{100}..A_{133}]$, tile5 calculates $[A_{134}..A_{166}]$, and tile6 calculates $[A_{167}..A_{199}]$. In the communication stage, tile0, tile1, and tile2 exchange data. At the same time, tile4, tile5, and tile6 exchange data, too. Finally, tile0, tile1, and tile2 will have the same data $[A_0..A_{99}]$, and tile4, tile5, and tile6 will have the same data $[A_{100}..A_{199}]$. Figure 7b shows the operations of B and C.

5 Results

This section presents the evaluation of our library and switch code generator. Evaluation is performed on a real Raw microprocessor that has 4x4 tiles. There are 16 rectangular tile shapes, and the speedup is derived from comparison with the performance of the original C code on one RAW tile. The original C code does not have any redundant code needed by multiple tiles. We use the Raw GNU C compiler to compile the tile and switch codes.

Figure 8a breaks down the SSCA#3 stage1 computation parts into computation kernels. It shows for each tile shape the percentage of computation kernels whose execution time is related to the whole program. The four computation kernels, discrete Fourier transform, inverse discrete Fourier transform, matrix element-by-element operation, and interpolation, account for more than 90% of the execution time. Figure 8b measures the speedup of each computation kernel. The four computation kernels that occupy the most execution time have nearly linear speedup, because their communication is not significant. The other computation kernels do not give such high speedup ratio because their operations involve communication among tiles to provide good data locality needed by the DFT, matrix operation,

Tile code	Route-plan file	Tile0 switch code

initial_col(N);   initial_row(M);
loop_length=y_end-y_begin-1; j=0;
static_send(loop_length);
for(i=y_begin;i<y_end;i++){
        ……
    communication_dist_row_col_tile(A,A[i],j); j=j+1;
}
	Tiles 3 2
	
	Loop 1 v
		sw_start:
		
		move $1,$csto
		sw_1:
		    nop route $csto->$cEo
		    nop route $cEi->$csti
		    bnezd- $1,$1,sw_1 route $cEi->$csti

loop_length=x_end-x_begin-1; j=0;
static_send(loop_length);
for(i=x_begin;i<x_end;i++){
        ……
    communication_dist_col_row_tile(B,B[i],j);j=j+1;
}
	Loop 1 h	move $1,$csto
		sw_2:
		    nop route $csto->$cSo
		    bnezd- $1,$1,sw_2 route $cSi->$csti

loop_length=x_end-x_begin-1;
static_send(loop_length);
for(i=x_begin;i<x_end;i++){
    loop_length=y_end-y_begin-1; k=0;
    static_send(loop_length);
    for(j=y_begin;j<y_end;j++){
        ……
        tmp[j]=C[j][i];
        communication_dist_col_row_tile(tmp,tmp[j],k);
        k=k+1;
    }
    //copy tmp[j] back to C[j][i]
}
	2DLoop-begin
	Loop 1 h
	2DLoop_end
		move $2,$csto
		sw_3:
		    move $1,$csto
		sw_4:
		    nop route $csto->$cSo
		    bnezd- $1,$1,sw_4 route $cSi->$csti
		    bnezd- $2,$2,sw_3

(a) Tile C code and route-plan file.

Initialization	Computation	Communication
$A = \begin{bmatrix} A_0 & A_1 & \cdots & A_{199} \end{bmatrix}^T_{1\times 200}$	$\Rightarrow \begin{bmatrix} [A_0 \cdots A_{33}] & [A_{34} \cdots A_{66}] & [A_{67} \cdots A_{99}] \\ [A_{100} \cdots A_{133}] & [A_{134} \cdots A_{166}] & [A_{167} \cdots A_{199}] \end{bmatrix}$	$\Rightarrow \begin{bmatrix} [A_0 \cdots A_{99}] & [A_0 \cdots A_{99}] & [A_0 \cdots A_{99}] \\ [A_{100} \cdots A_{199}] & [A_{100} \cdots A_{199}] & [A_{100} \cdots A_{199}] \end{bmatrix}$
$B = \begin{bmatrix} B_0 & B_1 & \cdots & B_{299} \end{bmatrix}_{1x300}$	$\Rightarrow \begin{bmatrix} [B_0 \cdots B_{49}] & [B_{100} \cdots B_{149}] & [B_{200} \cdots B_{249}] \\ [B_{50} \cdots B_{99}] & [B_{150} \cdots B_{199}] & [B_{250} \cdots B_{299}] \end{bmatrix}$	$\Rightarrow \begin{bmatrix} [B_0 \cdots B_{99}] & [B_{100} \cdots B_{199}] & [B_{200} \cdots B_{299}] \\ [B_0 \cdots B_{99}] & [B_{100} \cdots B_{199}] & [B_{200} \cdots B_{299}] \end{bmatrix}$
$C = \begin{bmatrix} C_{0,0} & \cdot\cdot & C_{0,299} \\ \vdots & & \vdots \\ C_{199,0} & \cdot\cdot & C_{199,299} \end{bmatrix}_{200\times 300}$	$\Rightarrow \begin{bmatrix} \begin{bmatrix} C_{0,0} & \\ & C_{99,99} \end{bmatrix} & \begin{bmatrix} C_{0,100} & \\ & C_{99,199} \end{bmatrix} & \begin{bmatrix} C_{0,200} & \\ & C_{99,299} \end{bmatrix} \\ \begin{bmatrix} C_{100,0} & \\ & C_{199,99} \end{bmatrix} & \begin{bmatrix} C_{100,100} & \\ & C_{199,199} \end{bmatrix} & \begin{bmatrix} C_{100,200} & \\ & C_{199,299} \end{bmatrix} \end{bmatrix}$	$\Rightarrow \begin{bmatrix} \begin{bmatrix} C_{0,0} & \\ & C_{199,99} \end{bmatrix} & \begin{bmatrix} C_{0,100} & \\ & C_{199,199} \end{bmatrix} & \begin{bmatrix} C_{0,200} & \\ & C_{199,299} \end{bmatrix} \\ \begin{bmatrix} C_{0,0} & \\ & C_{199,99} \end{bmatrix} & \begin{bmatrix} C_{0,100} & \\ & C_{199,199} \end{bmatrix} & \begin{bmatrix} C_{0,200} & \\ & C_{199,299} \end{bmatrix} \end{bmatrix}$

(b) A simple numerical example to show the data localization.

Figure 7: A simple example to show how tile C code and route-plan file work together.

and interpolation kernels.

Figure 8c,d show the execution clock cycles and overall speedups of stage1 computation part for a varying number of tiles. The dense matrix operations have significant parallelism and dominate the execution time. The speedup is nearly linear.

In the second stage, detection, correlation operation dominates nearly 100% of the execution time (Figure 9a). The speedup ratio in stage2 is not as good as stage1. The reason is that each tile does not have an equal number of letters to be detected. Figure 10 shows the partitions of the SAR image on the 4x4 tiles. Tile0 has only one letter (A), but tile4 has two letters (B and C). Tile8 has one letter (D), because it shares the boundary data with tile4. Tile12 has one letter (E). By comparing the total letters (4x5=20) and the letters of tile4 (2), the ratio is 20/2=10. ”10” is the best speedup we can expect.

Another way to improve the performance is by sharing the whole image on every tile. Then, each tile does some work on the image. For example, if we have 16 letters and each letter has 10 rotations, then each tile can detect letters on the whole image by using one letter template. Afterward, we can combine the results from the 16 tiles and choose the correct answer. This report does not implement this method.

Figure 9c,d show the execution clock cycles and overall speedup of stage2 computation part. The speedup is limited by the correlation kernel.

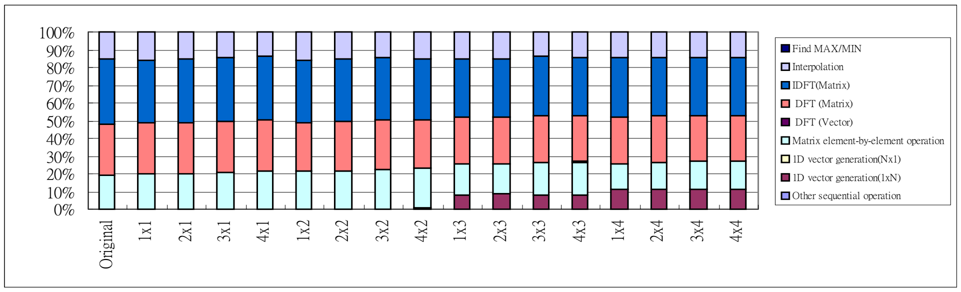
Because the benchmark runs on multiple tiles, the floating-point numbers operations cause round off errors. For example, the operation (10.01 + 10.02)/3.0 in the sequential program will give 6.67666674. If we parallelize this on two tiles, then the calculation will become (10.01/3.0) + (10.02/3.0) = 6.67666721. Although this error is not significant, it could be a problem in certain cases. For example, the difference in the value may prevent the parallel version from converging even though the sequential version converges.

6 Conclusion

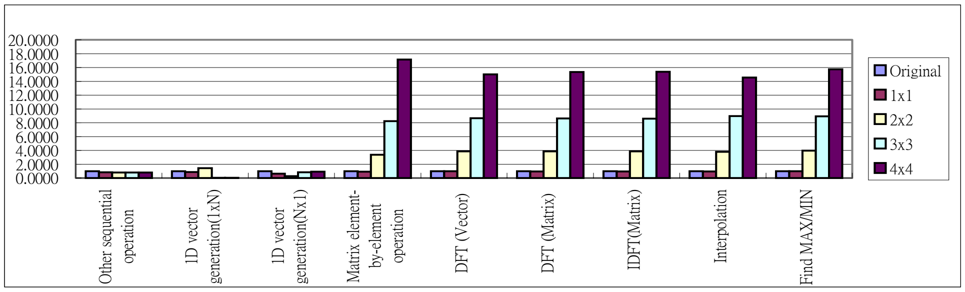
The Raw microprocessor distributes memory and resources over on-chip tiles coupled with point-to-point interconnection. The low-level details of hardware architecture are exposed to compiler and programmer, so it can provide the freedom to parallelize the application.

This report addresses the challenge of orchestrating distributed memory and communication resource to parallelize the SSCA#3 benchmark. We present a programmer-assisted method for writing the program.

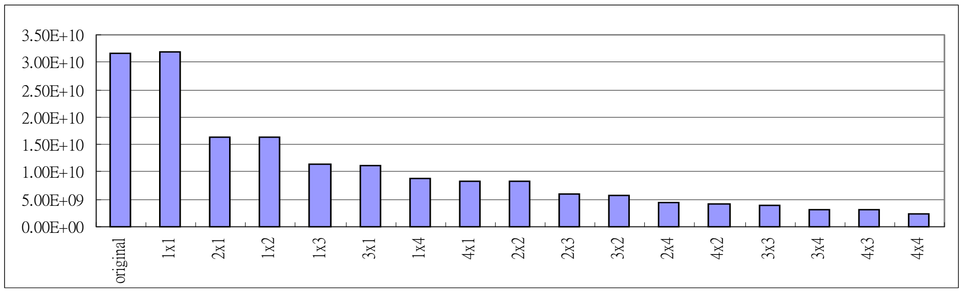
We demonstrate that our method gets a speedup of about 13.86 on 16 tiles in the stage1 computation part. There are two reasons for this result: first, the SSCA#3 has dense matrix



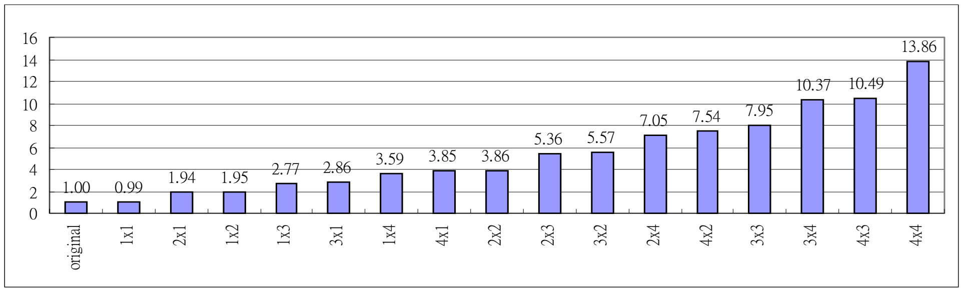
(a) Execution time percentage



(b) Speedup of each computation kernel

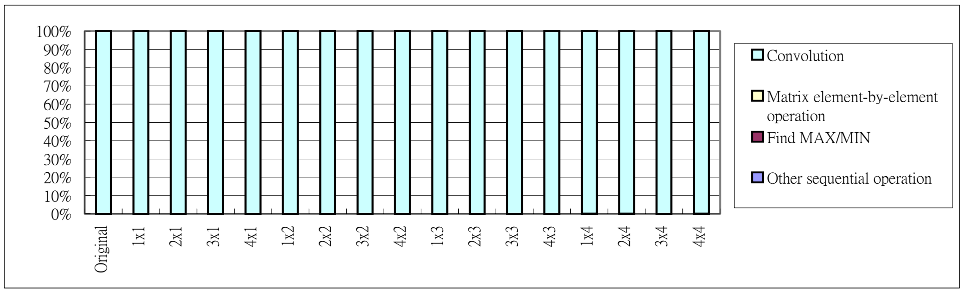


(c) Clock cycles

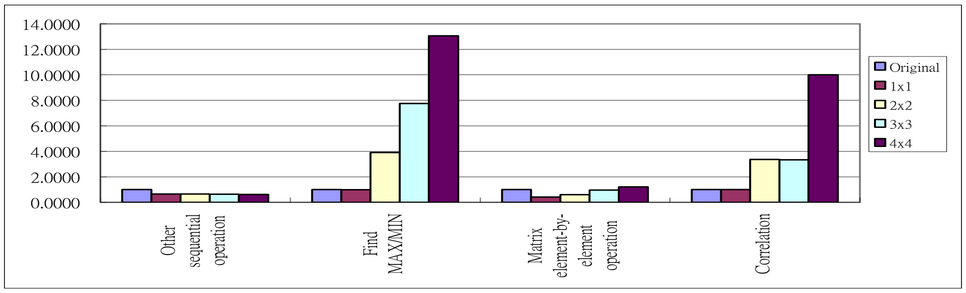


(d) Overall speedup

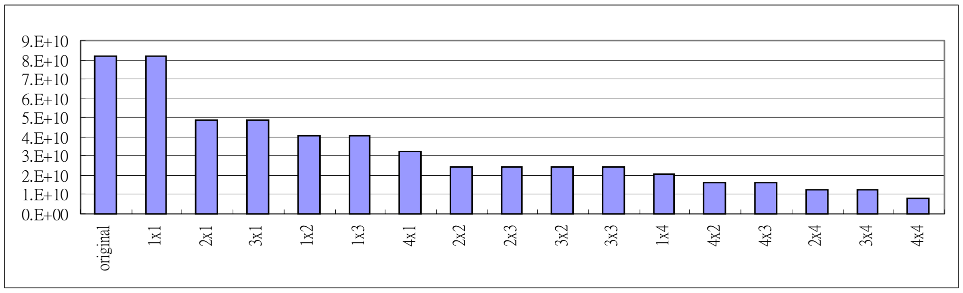
Figure 8: The distribution and speedup of stage1 computation kernels.



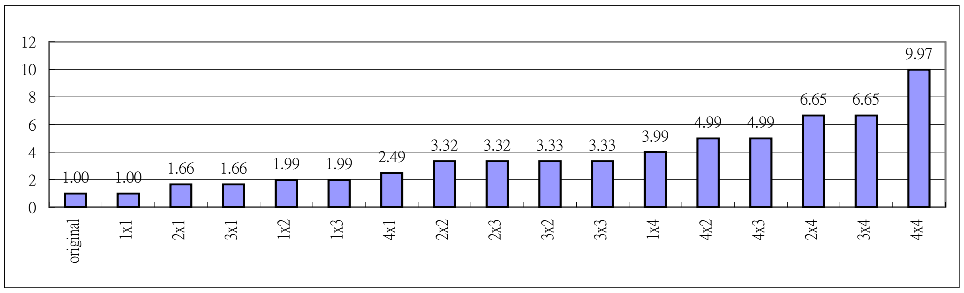
(a) Execution time percentage



(b) Speedup of each computation kernel

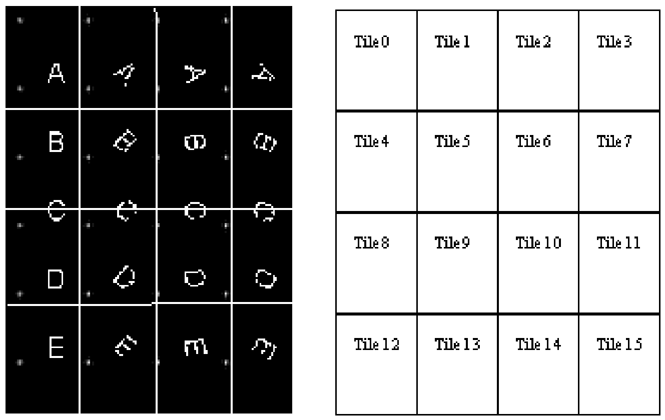


(c) Clock cycles



(d) Overall speedup

Figure 9: The distribution and speedup of stage2 computation kernels.



(a) (b)

Figure 10: The image partitioned into 4x4 tiles. Tile4 to tile7 need to detect two letters(B and C), and the others only have to detect one letter.

operations to help parallelism. Second, the communication patterns are regular and easy to manage. These two reasons also enable the static network communication library and switch code generator to lessen the complexity of programming on varying number of tiles. Further, the programmer needs only to write one tile C code and one route-plan file. Then, the switch code generator creates switch code for each tile automatically.

Our future work will apply this method to enable the run-time memory orchestration to be reconfigurable. When the Raw processor changes tile-shape, the data can move to the proper tiles by using the method described in Section 4.3. Then, each tile calculates the new index ranges and continues to execute. Our communication library will be extended with new functions to support the reconfigurable feature.

7 Acknowledgement

We thank Steve Crago, Janice Onanian McMahon, and Jinwoo Suh for offering great suggestions about implementing the parallel SSCA#3 benchmark on the Raw microprocessor board.

References

[1] RAW Architecture Workstation. *www.cag.csail.mit.edu/raw.*

[2] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, Mar/April 2002.

[3] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A compiler-managed memory system for raw machines. *Proceedings of the Twenty-Sixth International Symposium on Computer Architecture (ISCA-26), Atlanta, GA*, June 1999.

[4] High Productivity Computing Systems. *http://www.highproductivity.org/SSCABmks.htm.*

[5] Mehrdad Soumekh. *Synthetic aperture radar signal processing with MATLAB algorithms.* Wiley, 1999.

[6] Theresa Meuse. *HPCS Scalable Synthetic Compact Applications #3 Sensor Processing, Knowledge Formation and Data I/O (Intelligence).* MIT Lincoln Laboratory, January 2006.