ESC-TR-2006-063

Project Report
PCA-Kernel-3

Polymorphous Computing Architecture (PCA) Kernel Benchmark Measurements on the MIT Raw Microprocessor

R.J. Haney
J.M. Lebak
M.A. Alexander
H. Chan
P.A. Jackson
E.L. Wong

14 June 2006

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Lexington, Massachusetts

Prepared for the Defense Advanced Research Projects Agency
under Air Force Contract FA8721-05-C-0002.

Approved for public release; distribution is unlimited.

ADA 453293

This report is based on studies performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology. This work was sponsored by the Defense Advanced Research Projects Agency/IPTO under Air Force Contract FA8721-05-C-0002.

This report may be reproduced to satisfy needs of U.S. Government agencies.

The ESC Public Affairs Office has reviewed this report, and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER

Gary Tutungian
Administrative Contracting Officer
Plans and Programs Directorate
Contracted Support Management

Non-Lincoln Recipients

PLEASE DO NOT RETURN

Permission has been given to destroy this document when it is no longer needed.

Massachusetts Institute of Technology
Lincoln Laboratory

Polymorphous Computing Architecture (PCA) Kernel Benchmark Measurements on the MIT Raw Microprocessor

R.J. Haney
J.M. Lebak
M.A. Alexander
H. Chan
P.A. Jackson
E.L. Wong
Group 102

Project Report PCA-Kernel-3

14 June 2006

Approved for public release; distribution is unlimited.

Lexington Massachusetts

Acknowledgments

This work was sponsored by the Defense Advanced Research Projects Agency's Polymorphous Computing Architecture Program, directed by Mr. Robert Graybill.

A Raw board for the use of this project was provided by the Computer Architecture Group at the Massachusetts Institute of Technology and by the Information Sciences Institute East of the University of Southern California. We are grateful for the help of Anant Agarwal, Paul Johnson, Jason Miller, Michael Taylor, and David Wentzlaff at MIT, and Chen Chen and Steve Crago at USC/ISI East, in obtaining and setting up the board.

Hank Hoffmann of MIT developed the methodology of programming Raw as a systolic streaming machine in his Master's thesis at MIT. His methodology heavily influenced this work.

Jinwoo Suh of USC/ISI East contributed results for an optimized FIR filter implementation for Raw.

The authors acknowledge contributions from the MIT Lincoln Laboratory PCA team, including Janice McMahon, Robert Bond, Glenn Schrader, Bill Coate, Jim Daly, and others.

TABLE OF CONTENTS

1. Introduction 1

2. Methodology 3
   2.1 Kernels and Data Sets 3
   2.2 Metrics 3
   2.3 Measurement Platforms 3

3. Signal Processing Benchmarks 7
   3.1 Finite Impulse Response Filter Bank 7
   3.2 QR Decomposition 16
   3.3 Singular Value Decomposition 29
   3.4 Constant False Alarm Rate Detection 40

4. Communication Benchmark: Corner Turn 45
   4.1 Algorithm Description 45
   4.2 Implementation on Raw 45
   4.3 Benchmark Results 46
   4.4 Raw Static Network Implementation 48

5. Information and Knowledge Processing Benchmarks 49
   5.1 Pattern Matching 49
   5.2 Database Operations 52
   5.3 Graph Optimization via Genetic Algorithm 59

6. Raw Kernel Benchmark Observations 63
   6.1 Board-Simulator Comparison 63
   6.2 Development Effort 65
   6.3 Baseline Results and Platform Comparison 66

7. Conclusions 71

APPENDIX A – Testbed hardware design 73
   A.1 Introduction 73
   A.2 Hardware Setup 73
   A.3 Software Design 76
   A.4 Firmware Design 76
   A.5 Summary 78

APPENDIX B – Testbed software design 79
   B.1 Introduction 79
   B.2 Tutorial 79
   B.3 Overview 81
   B.4 Tasks 83

B.5 Testbed 87
B.6 Map Grammar 93

REFERENCES 97

LIST OF ILLUSTRATIONS

Figure No.		Page
1	Identification of tiles and ports on the Raw chip.	4
2	Serial time-domain FIR filter operations.	8
3	Usage of a 4×4 Raw chip for performing a time-domain FIR filter.	8
4	Parallel time-domain FIR filter operations.	9
5	Usage of a 4×4 Raw chip for performing the FFT.	10
6	FFT Butterfly operations.	12
7	FFT Base-4 reversal.	13
8	Time-domain FIR filter results.	15
9	Frequency-Domain FIR Filter results.	15
10	Matlab code for computing Fast Givens QR.	18
11	Usage of a 4×4 Raw chip for performing complex QR Decomposition.	19
12	Usage of an arbitrary sized $R \times R$ Raw chip for performing complex QR Decomposition.	19
13	High-level pseudocode of Fast Givens QR implementation for Raw.	21
14	Storage of R after QR computation.	22
15	Storage of Q after QR computation.	22
16	Data flow and computation that occurs while computing Fast Givens rotations.	23
17	Data flow and computation that occurs while applying Fast Givens rotations.	24
18	Data flow and computation for computing the final $m - n$ columns of Q.	25
19	QR results using a 4×4 Raw.	26
20	QR results using a 8×8 Raw.	27
21	QR scalability results: 4×4 vs. 8×8 Raw.	29
22	Usage of a 4×4 Raw chip for performing the complex Stream Hestenes SVD.	31
23	High-level pseudocode of the Stream SVD implementation for Raw.	32
24	SVD data flow for computing Jacobi rotations for a boundary case.	34
25	SVD data flow for computing Jacobi rotations for a non-boundary case.	34
26	SVD data flow for applying Jacobi rotations for a boundary case.	36
27	SVD data flow for applying Jacobi rotations for a non-boundary case.	36
28	Stream Hestenes SVD data flow for a complete block sweep.	37
29	SVD results using a 4×4 Raw.	39
30	Sliding window in CFAR detection.	41
31	CFAR tile allocation on MIT Raw.	41
32	Performance of CFAR on the MIT Raw.	43
33	Corner turn results for square matrices on a single tile of Raw.	47
34	Corner turn results for square matrices on 16 tiles of Raw.	47

35 Corner turn throughput for the G4, Xeon, and Raw board. 48
36 The pattern match kernel mapping. 50
37 Pattern match throughput on Raw with 512 library patterns. 51
38 Pattern match efficiency on Raw with 512 library patterns. 51
39 Usage of a 4×4 Raw chip for performing Database Operations. 53
40 Data flow for performing the search Database Operation. 54
41 Data flow for performing the insert Database Operation. 55
42 Data flow for performing the delete Database Operation. 56
43 Database throughput results. 57
44 Database latency results. 57
45 Database latency results when performing search operations only. 58
46 Throughput of the genetic algorithm benchmark. 61
47 QR Decomposition results using the Raw Board and the Raw cycle-accurate simulator. 64
48 QR Decomposition results using the Raw Board, cycle-accurate simulator, and the cycle-accurate simulator with modified DRAM read and write penalties. 64
49 Achieved throughput of kernels on defined data sets on the Raw board. 67
50 Average throughput for each kernel on the Raw board and on the Raw board scaled to 425 MHz, compared to the G4 and Xeon. 68
51 Average throughput per unit power for each kernel on the Raw board compared to the G4 and Xeon. 68
52 Achieved data set stability for kernels on the Raw board compared with the G4 and Xeon. 69
53 PCA testbed system components. 74
54 WildStar II Processing Element (PE) 0. 77
55 WildStar Data Port Daughter Card I/O Processing Element (PE). 77
56 Raw Handheld Left Expansion FPGA. 78
57 The client side of the testbed software shown running the CFAR kernel. 82
58 The host platform's side of the testbed software. 82
59 A sample task map with two tasks: CFAR and Genetic Algorithm. 84
60 A sample platform map for a testbed that supports four different platforms. 88
61 Methods used to send data to the various Raw instantiations. 91

LIST OF TABLES

Table No.		Page
1	Comparison of SVD Workloads.	39
2	Lines of code for four FIR filter implementations.	65
3	Processor Parameters	66
4	Kernel stability numbers for the Raw, G4, and Xeon.	70
5	Average Performance and Performance Per Watt for the Raw, G4, and Xeon	70

1. Introduction

The DARPA Polymorphous Computing Architecture (PCA) program is a research initiative aimed at developing new computer architectures with a high degree of flexibility. Unlike current computer architectures that are rigid in nature, PCAs will have the capability to adapt ("morph") to match the problem being solved. This flexibility will allow higher overall system performance in a broad range of applications.

MIT Lincoln Laboratory has defined a set of kernel benchmarks for the PCA program [11]. The kernel-level benchmarks have been chosen to stress both computation and communication aspects of the architecture. The particular benchmarks chosen are based on the frequency of their use in current and future applications. They are drawn from the areas of signal and image processing, communication, and information and knowledge processing. Each of these areas imposes different processing requirements on the architecture in terms of operations performed and memory bandwidth required.

This document describes a set of measurements of the PCA kernel benchmarks on a prototype PCA chip, the Raw processor developed by MIT. Chapter 2 describes the measurement platform, the metrics, and the test methodology in more detail. Chapters 3, 4, and 5 describe results for, respectively, signal processing, communication, and information and knowledge processing benchmarks. Finally, Chapter 6 gives a summary of the kernel benchmark measurements. Where appropriate, we compare these measurements to earlier measurements on the PowerPC G4 [10] as well as to measurements on an Intel Xeon server.

2. Methodology

This report provides measurements for the kernel benchmarks defined by MIT/LL [11] on the MIT Raw processor. We will be comparing these results with those previously taken on the PowerPC G4 embedded processor [10], and with results obtained on an Intel Xeon server. In this section, we describe the kernels, data sets, metrics, and platforms in more detail.

2.1 Kernels and Data Sets

The kernel benchmarks were distilled from a survey of Department of Defense applications. They fall into three broad categories: signal and image processing (SIP), communication, and information and knowledge processing (IKP). In general, one could characterize the operation performed by the SIP kernels as regular and predictable, and the operations performed by the IKP kernels as data-dependent.

There are four SIP kernel benchmarks: finite impulse response (FIR) filter, QR factorization, singular value decomposition (SVD), and constant false-alarm rate (CFAR) detection. The communication benchmark is the corner turn. The IKP kernel benchmarks are pattern match, genetic algorithm, and database operations.

For each benchmark, a set of problem sizes are defined in the original kernel benchmark report [11]. We report performance on these sizes in Section 6.3. In each kernel's section, we vary parameters at a finer level of granularity, in order to understand in more detail the effect of different data set sizes on performance. Throughout this section, we refer to the kernel by the index k, and refer to particular data sets for a given kernel as d_i, where $i = 1, 2, \ldots, N_k$, and N_k varies from kernel to kernel.

2.2 Metrics

The major metric of interest for each problem size is the total time or latency, $L_1(k, d_i)$, to perform kernel k for a data set size d_i. In most cases, this time includes the time to send data into the chip from outside.

Metrics that we calculate from latency include *throughput*, *efficiency*, and *stability* as defined in the original benchmark report [11]. Workload values for each kernel benchmark are specified in that report; these values are used in the computation of throughput. We compute performance per unit power from the throughput values.

2.3 Measurement Platforms

2.3.1 Raw board

The Raw board used for this project contains a single Raw chip consisting of 16 Raw tiles. Each tile is identified either with an (x, y) address pair or with a single identifying number. The chip has sixteen ports numbered zero though fifteen through which it communicates with the outside world. The numbering of the tiles and ports is shown in Figure 1.

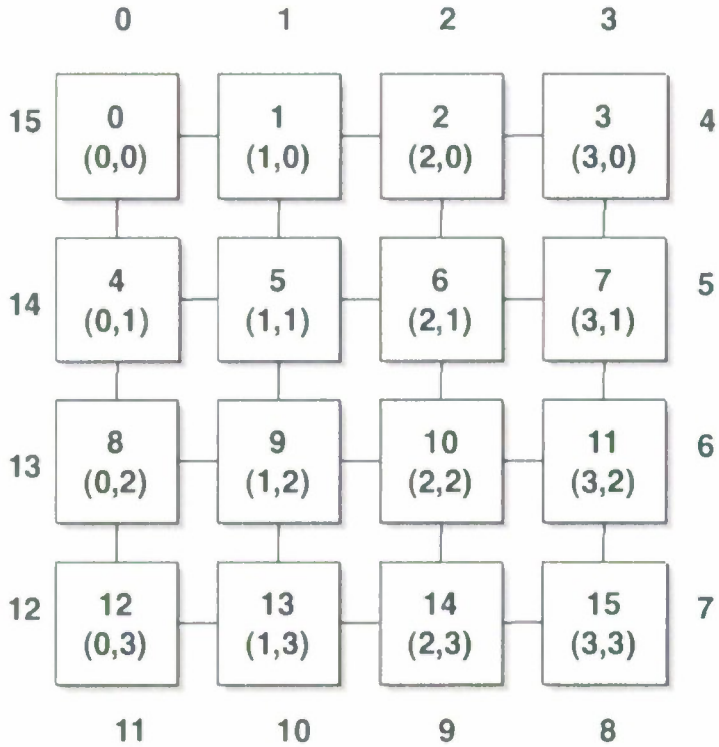

Figure 1. Identification of tiles and ports on the Raw chip.

Each tile is capable of performing one operation per clock cycle and includes a 32-kbyte level-1 cache. The default clock rate for the chip is 100 MHz. This reflects the constraints of the current board firmware. The board components are designed to run at up to 300 MHz with appropriate firmware. With suitable external hardware, MIT estimates that the current prototype Raw chip could be clocked at rates up to 425 MHz [13]. In the tests described in this document, the chip runs at 100 MHz and we calculate its peak performance as 1.6 Gflop/s.

The tiles on Raw are connected by two *static networks* and two *dynamic networks*. One of the dynamic networks is used by the chip for memory accesses: the other is available to the programmer. The term "static" applied to the networks implies that the programmer must set up the communication pattern used by the switch in advance of issuing the commands for communication. Once programmed, the static networks are generally faster than the dynamic networks. For more information see Taylor *et al.* [20].

The Raw board puts the interface to memory only on the east side of the chip (ports 4–7). It also includes an *expansion connector* that gives access to the north and south sides of the chip. In order to test Raw in a streaming situation, MIT/LL built external hardware to move data into the chip from the expansion connector. An Annapolis Microsystems WildStar-2 board serves as a buffer and a controller for the interface. The interface streams data into ports 0, 3, 8, and 11 of the Raw chip, using streaming data port cards connected to the Annapolis board, and a firmware design referred to as the *speed gasket* on the Raw board's Virtex-II 3000 FPGAs. More details of the testbed hardware can be found in Appendix A. We refer to this interface as the "high-speed input/output" or HSIO.

The Raw board also has a USB interface that can be used for I/O for data sizes that are too large for the buffers on the HSIO board. We found this interface to have a high latency and thus tried to avoid using it. The only situation where its use could not be avoided was in the case of the CFAR kernel, which defined data sets that were too large for HSIO. For more details, see Section 3.4.

The Raw chip was measured to consume 2.5 A when operating at 100 MHz and a core voltage of 1.8 V while running the "vpenta" benchmark from the SPEC '92 benchmark suite [9]. We round upward and use a figure of 5 W for the typical operating power of the Raw chip at 100 MHz.

2.3.2 Raw Simulator

MIT provides a cycle-accurate simulator for Raw that was used in the early stages of this project. A major benefit of the simulator is that it allows us to scale the number of tiles in the Raw chip and to experiment with other board designs, including ones where all sides of the chip have an interface to memory. Use of the simulator also eases development, since there is only one Raw board available at MIT/LL.

While the simulator is presented by MIT as being an accurate model of the Raw chip, we have found that it does not accurately model the board. Our comparison of the board and the simulator can be found in Section 6.1. Despite these disagreements between the board and simulator, the simulator still has a great deal of value for showing the potential of the Raw chip. We describe experiments on the Raw simulator scaled to an array of 64 tiles for the QR factorization kernel in Section 3.2 and for the pattern match kernel in Section 5.1. These experiments show the potential of Raw's architecture as feature sizes continue to shrink.

2.3.3 G4 Platform

As previously described, the target platform for the G4 measurements is a single node of a multi-node Mercury computing system [10]. The system occupies a single VME chassis and has sixteen compute nodes. Each node is a 500 MHz Motorola PowerPC G4 processor, model 7410, with a 32 kbyte level-1 cache on-chip and a 2 Mbyte L2 cache connected through a 250 MHz bus [12]. Each node has 256 Mbyte of "local" DRAM, connected through a 125 MHz bus. The nodes are connected through a RACE++ crossbar network, though this is not used in any of the benchmarks. The daughter-cards used in these systems were first announced by Mercury in March of 2002. At the time of this report, the platform and the processors are approximately four years old.

The PowerPC G4 includes a vector processing unit referred to as the AltiVec unit [14]. This unit operates in parallel on data in 128-bit registers as if they were multiple smaller data registers. Operations are performed in a SIMD (single instruction stream, multiple data stream) fashion. For our purposes, this means that each register is treated as four single-precision (32-bit) floating-point numbers. Since the AltiVec floating-point units can retire four multiply-add instructions in a single cycle, we calculate its peak performance as 8 flop per cycle times the clock speed of the processor, or 4 Gflop/s.

The hardware specification for the Motorola 7410 processor gives a typical power dissipation of 5.3 W when running "typical benchmarks" and a maximum power dissipation of 11.3 W when running a set of instructions contrived to keep the processor "maximally busy" [15]. When computing achieved performance per unit power, we use the *typical* number rather than the *maximal* number. This decision follows from the efficiency numbers that our benchmarks achieve, which are far from 100% utilization of the processor.

2.3.4 Xeon Platform

The target platform for the Xeon measurements is a single node of a Dell PowerEdge 2650 Server. This system contains two Intel Xeon processors operating at 2.8 GHz in a 2U rack. The 2.8 GHz Xeon processor was introduced by Intel in November of 2002, making it slightly newer than the G4 processor previously described. It is a newer process technology generation (0.13 micron) than the G4 (0.18 micron). It is also targeted at the server market, rather than the embedded market.

Each processor has an 8 kbyte level-1 cache and a 512-kbyte level-2 cache, both located on the chip. The Xeon includes vector instructions referred to as *streaming SIMD extensions* or SSE instructions. These have a similar function to the AltiVec instructions on the G4, and in many cases there are roughly equivalent instructions on the two processors. However, unlike AltiVec, SSE does not include a multiply-add instruction. Thus we calculate the peak performance of the Xeon as 4 flop per cycle times the clock rate or 11.2 Gflop/s. We use a power consumption figure of 74 W for the Xeon, reflecting Intel's cooling guidance for the chip [8].

3. Signal Processing Benchmarks

The four signal processing kernels defined for PCAs are the FIR filter, the QR factorization, the SVD, and CFAR detection. Each presents a different set of characteristics in terms of operation counts and memory references. The results for these kernels are respectively discussed in Sections 3.1, 3.2, 3.3, and 3.4.

3.1 Finite Impulse Response Filter Bank

3.1.1 Algorithm Description

The FIR filter benchmark measures the performance of a bank of FIR filters. Each FIR filter $m, m \in \{0, 1, ..., M-1\}$, has a set of impulse response coefficients $w_m[k], k \in \{0, 1, ..., K-1\}$. If the length of the input vector is N, the output of filter m, y_m, is the convolution of w_m with the input x_m:

$$y_m[i] = \sum_{k=0}^{K-1} x_m[i-k]w_m[k] \qquad \text{for } i = \{0, 1, ..., N-1\}. \tag{1}$$

Direct implementation of equation 1 is referred to as a *time-domain* implementation of the FIR. Another common implementation uses fast convolution with the fast Fourier transform (FFT): this is referred to as a *frequency-domain* implementation. The most efficient implementation depends on various factors including the size of the filter response vector. For the Raw Processor, we have chosen to base the frequency-domain implementation on a radix-4 FFT. For a description and example of a radix-4 FFT, see [22].

3.1.2 Implementation Features

Time-Domain FIR Filter

Serial Implementation. To understand the parallel implementation of the time-domain FIR filter kernel, it is helpful to first be familiar with a serial implementation. The time-domain FIR filter kernel consists of vector-scalar multiplies of the input sample length N vector by each of the m filter elements. This results in m vectors $(M_i, 0 \leq i < m)$ of length N. These vectors are then shifted right i elements. Once shifted, the vectors are accumulated to calculate the final result. Figure 2 shows the flow of serial time-domain FIR filter operations.

Parallel Raw Implementation. The symmetric stream mapping shown in Figure 3 provides the ability to perform two parallel complex time-domain FIR filters. This mapping requires the four corner tiles (hereafter referred to as memory tiles) to supply streaming data and manage memory for the Raw processor. The remaining 12 tiles (hereafter referred to as the computation tiles) will perform the complex time-domain FIR computations.

The parallel Raw time-domain FIR filter kernel is similar to the serial kernel implementation described above. Using the mapping in Figure 3 however, it is possible to do six of the vector-scalar multiplies in parallel. After the initial six vector-scalar multiplies have been performed, the results are accumulated as they stream down towards the lower memory tiles. Once arriving at

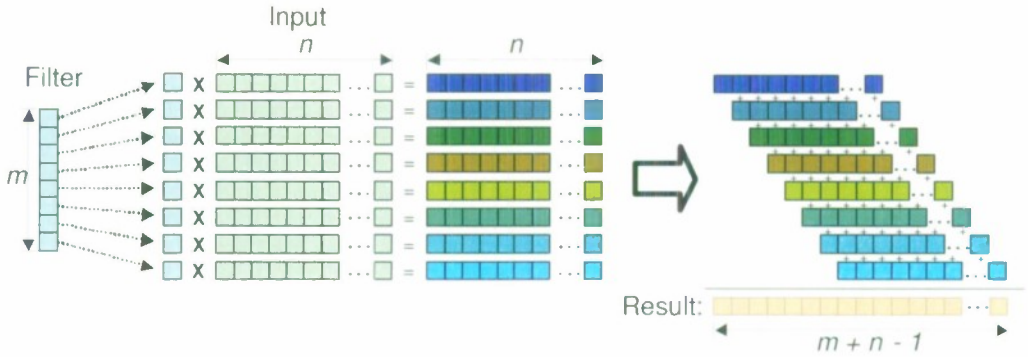

Figure 2. Serial time-domain FIR filter operations.

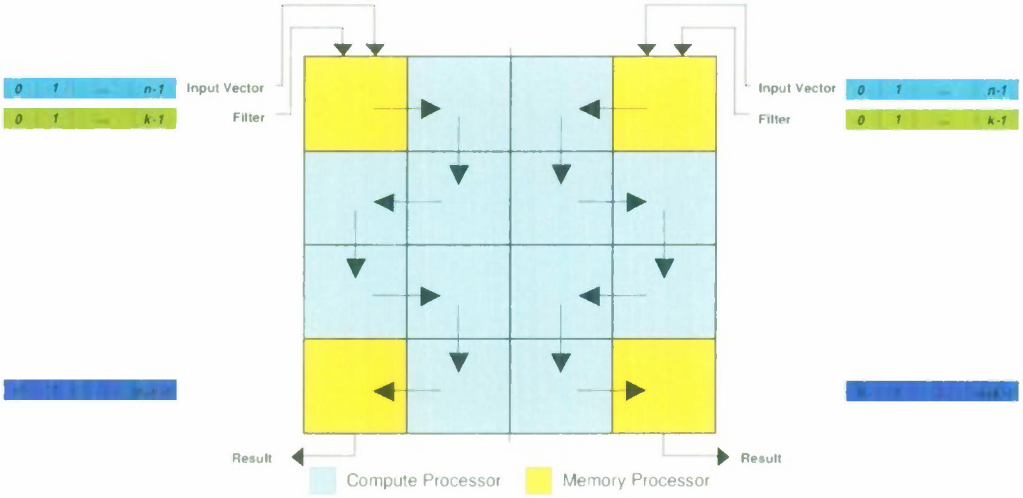

Figure 3. Usage of a 4 × 4 Raw chip for performing a time-domain FIR filter. The four corner (in yellow: tiles 0, 3, 12, and 15) tiles are used for memory and I/O. The remaining tiles are used for computation.

the memory tiles, the sum is stored in an intermediate vector in memory. The next six parallel results are computed, and again accumulated as they stream down towards the lower memory tiles. This time, the intermediate result index is shifted right six elements (to compensate for the previous intermediate result) before being accumulated with the previous intermediate vector. In general, an index for an intermediate vector arriving at the lower memory tiles is shifted six elements for each previous intermediate vector. It is then accumulated with the summation thus far. This procedure is conducted until each of the filter elements have been accounted for. Figure 4 shows the flow of parallel time-domain FIR filter operations.

Frequency-Domain FIR Filter

The following section will discuss the three major components of the frequency-domain FIR filter:

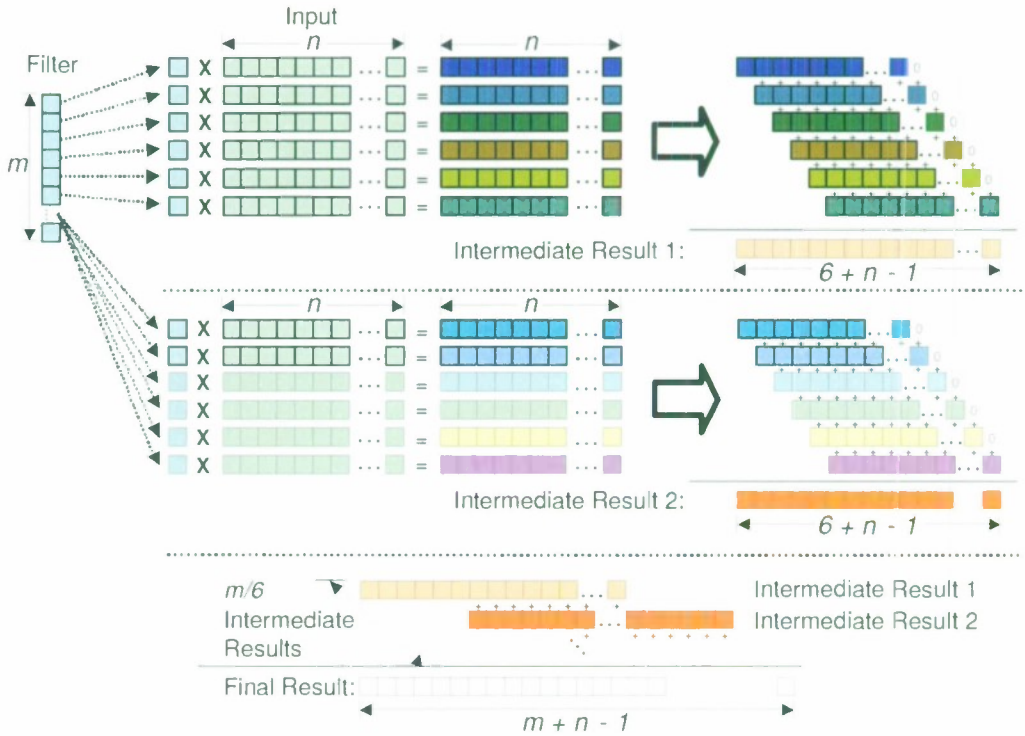

Figure 4. Parallel time-domain FIR filter operations.

1. an FFT of the input,
2. an element-wise multiply of the FFT result and the filter, and
3. an IFFT of the element-wise multiply result.

We will also discuss optimizations made to the FIR filter including loop unrolling and internal twiddle factor consistencies.

Similar to the time-domain algorithm, the symmetric mapping displayed below in Figure 5 provides the ability to perform two parallel frequency-domain FIR filters simultaneously. This mapping also requires the four corner memory tiles to supply streaming data and manage memory for the Raw processor. The 8 tiles in the middle columns (hereafter referred to as the computation tiles) will perform the FIR computations. The two center tiles on the left and right sides will be used to buffer and store intermediate butterfly results.

Fast Fourier Transform. The outline of a FFT of length N can be viewed as:

```
for (log N phases)
    for (N / 4 butterflies)
        Butterfly()
    end
end
```

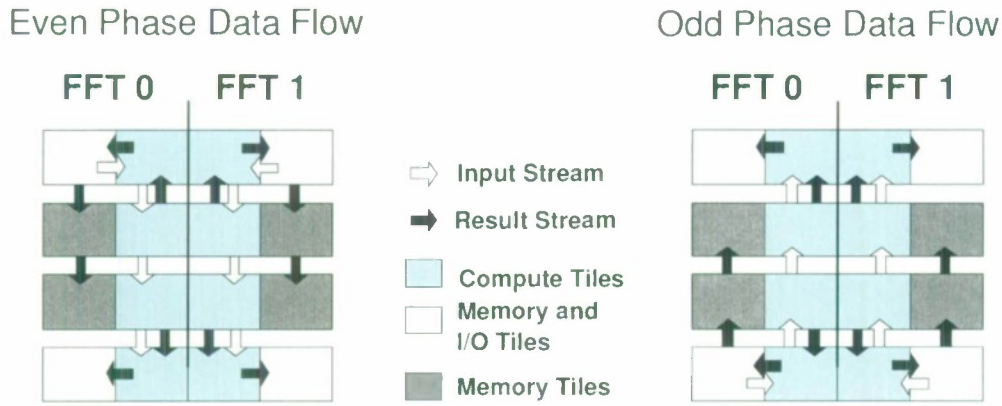

Figure 5. Usage of a 4 × 4 Raw chip for performing FFT. The four corner tiles (in white: tiles 0, 3, 12, and 15) are used for memory and I/O. The two middle tiles on both the east and west sides of the chip (in grey: tiles 4, 7, 8, and 11) are used for memory. The remaining 8 center tiles (in blue: tiles 1, 2, 5, 6, 9, 10, 13, and 14) are used for computation.

The main function in an FFT is the butterfly operation. The butterfly operation consists of the input samples being multiplied by twiddle factors. A twiddle factor has the general form:

$$w_k = e^{j\frac{2\pi}{N}}. \tag{2}$$

The length-N FFT consists of $\log_4 N$ phases of $N/4$ butterflies per phase. Each butterfly takes four inputs and computes four outputs.

In the initial even phase (see Figure 5), input samples are streamed into the computation tiles in the order they are required to perform butterfly operations (to be described in the following section). As they are streamed into the four computation tiles, each tile computes one of the four intermediate results. The results are sent to the bottom memory tiles (tiles 8, 11, 12, and 15) as depicted by the result stream shown in the even phase data flow of Figure 5.

Once each set of butterflies has been computed in the initial phase, data strides are recomputed on the bottom four memory tiles (tiles 8, 11, 12, and 15). The results from the initial phase then become inputs to the subsequent phase, and are streamed up into the compute tiles in the order they are required to perform butterfly operations. Again, as they are streamed into the four computation tiles, each tile computes one of the four intermediate results. The same process described above is performed, only this time the final destination for the results will be the top four memory tiles (tiles 0, 3, 4, and 7). This data flow is the odd phase depicted in Figure 5. This process is conducted for each phase. When all the phases are completed, the result is a vector whose elements are stored in base-4 reversed order. The FFT result will always reside on the top two memory tiles (tiles 0 and 3).

Butterfly Operation. In a radix-4 FFT implementation, a butterfly operation requires 4 input elements and a series of twiddle factors. For example, in phase zero of a length-16 FFT, butterfly zero performs the following computations using elements x_0, x_4, x_8 and x_{12} :

$$x_0^1 = (x_0^0 w^{(4*0*0)} + x_4^0 w^{(4*0*1)} + x_8^0 w^{(4*0*2)} + x_{12}^0 w^{(4*0*3)}) * w^0 \tag{3}$$

$$x_4^1 = (x_0^0 w^{(4*1*0)} + x_4^0 w^{(4*1*1)} + x_8^0 w^{(4*1*2)} + x_{12}^0 w^{(4*1*3)}) * w^0 \tag{4}$$

$$x_8^1 = (x_0^0 w^{(4*2*0)} + x_4^0 w^{(4*2*1)} + x_8^0 w^{(4*2*2)} + x_{12}^0 w^{(4*2*3)}) * w^0 \tag{5}$$

$$x_{12}^1 = (x_0^0 w^{(4*3*0)} + x_4^0 w^{(4*3*1)} + x_8^0 w^{(4*3*2)} + x_{12}^0 w^{(4*3*3)}) * w^0 \tag{6}$$

where w are twiddle factors and x are input and output elements whose superscript represents the number of phases completed. Each butterfly operation calculates four butterfly elements that involve the sum of four product terms.

The exponent of the twiddle factors within the parenthesis is calculated as:

$$(\frac{N}{\text{Radix}}) * (\text{Butterfly Element}) * (\text{Product Term Index}). \tag{7}$$

To calculate the power of the twiddle factor outside the parenthesis, we use:

$$floor(\frac{\text{Butterfly}}{4^{phase}}). \tag{8}$$

The twiddle factors for each of the elements in the final phase should all be one. If we refer to the four inputs to a butterfly as x_a, x_b, x_c, and x_d and the four results as $\{x_k, 0 \leq k < 3\}$, then a general formula for radix-4 butterfly computation b in phase p is:

$$x_k^{p+1} = (x_a^p w^{(\frac{N}{4}*k*0)} + x_b^p w^{(\frac{N}{4}*k*1)} + x_c^p w^{(\frac{N}{4}*k*2)} + x_d^p w^{(\frac{N}{4}*k*3)}) * w^{\frac{b}{4^p}}. \tag{9}$$

Figure 6 depicts the initial butterfly operation in both phase zero and phase one for a 16 sample input. The stride between input elements in phase zero is $\frac{N}{4}$. As we move from one phase to the next, the stride between input elements is always $\frac{1}{4}$ that of the stride from the previous phase. The stride in the final phase is always one. The selection of sets of butterfly inputs varies from phase to phase. For a more in-depth description of butterfly operations, see [22].

Element-Wise Multiply. As noted above in the Butterfly section, the external twiddle factors for each of the elements in the final phase are all one. This means there are N complex operations where intermediate values are being multiplied by one. We can use this to our advantage by embedding the element-wise multiply of the FIR filter into the final phase of the FFT. During the final phase of the FFT, rather than calculating the external twiddle factor (always equal to one), an index is calculated to fetch the appropriate filter value from memory. This value is then used in place of the twiddle factor to complete the butterfly operation.

The portion of the filter required by a given tile is stored in memory in a base-4 reversed order during initialization. This corresponds to the output of the FFT, which is also stored in base-4 reversed order.

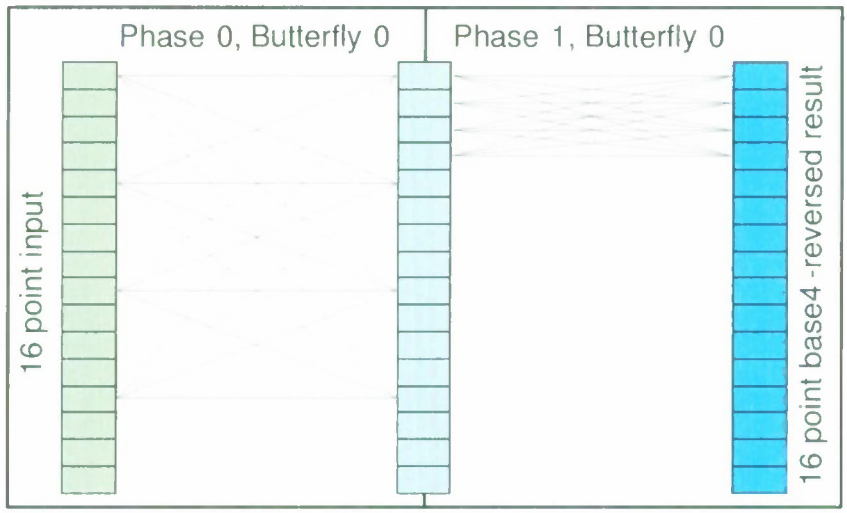

Figure 6. Initial butterfly operation in phase zero and phase one for a 16 sample input. The stride between elements in phase zero is 4, calculated from $\frac{N}{4}$. In phase one, the stride is reduced to $\frac{1}{4}$ that of the previous phase's stride.

Inverse Fast Fourier Transform The IFFT implementation is very similar to the FFT described above. We note three major differences. One major difference is that the input indices to the IFFT are base-4 reversed. To compensate for this, during the initial phase of the IFFT, the normal indices are not used to fetch samples from memory for the butterfly operations. Instead, we use the indices' pre-computed base-4 reversed values to index the appropriate samples. These samples are then streamed into the computation tiles. Figure 7 shows an example for a 16 sample input. Note that because of the base-4 reversed input to the IFFT, the result will be base-4 reversed.

Another difference between the FFT and the IFFT implementations is the calculation of twiddle factors. The values used in the IFFT are the conjugates of the values used in the FFT. Based on performance measurements, reusing the FFT twiddle factors and multiplying the imaginary components by negative one is more efficient than fetching new conjugated values from memory.

The final difference between the FFT and IFFT is that the IFFT requires each element to be multiplied by $\frac{1}{N}$ after the last phase of butterfly computations. To optimize this, the multiply operation has been embedded into the butterfly weight multiplication in the final phase as was done with the element-wise multiply in the FFT routine.

FIR Optimizations

It is necessary to make three significant optimizations to the FIR filter in order to improve performance. The first is motivated by the consistency in all of the internal twiddle factors. The formula for the exponent of the internal twiddle factors is depicted in Equation (7). Because of the $\frac{N}{Radix}$ component in Equation (7), the internal twiddle factors are the same no matter the current phase, butterfly, or butterfly product. It is also true that each of the twiddle factors falls on an axis of the unit circle. The first twiddle factor falls on $(1, 0)$, the second on $(0, 1)$, the third on $(-1, 0)$, and finally the fourth on $(0, -1)$. Therefore the multiplication of the internal twiddle factors can be

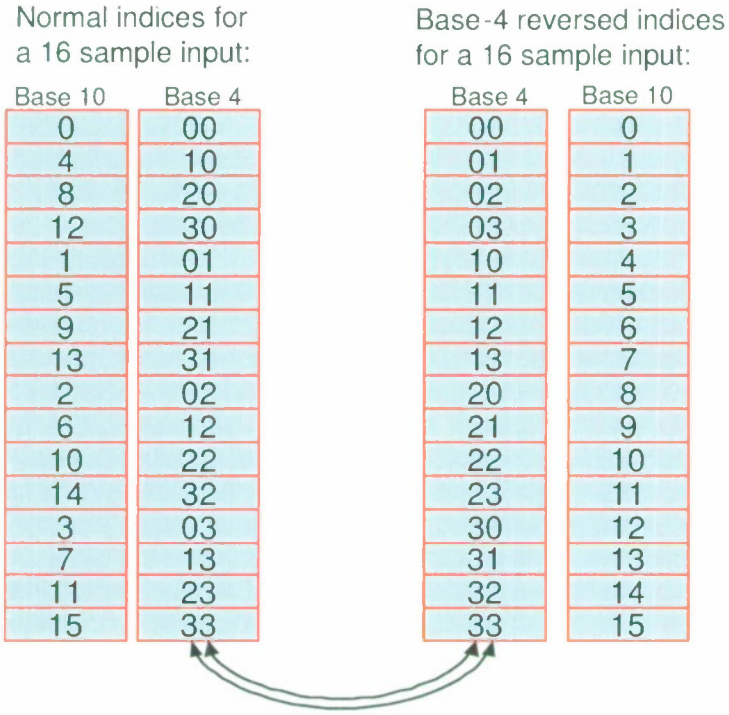

Figure 7. Normal indices for a 16 sample input and their base-4 reversed equivalents.

replaced with a sequence of equivalent additions and subtractions. Equation (4) can be expressed as the following:

$$x_4^1.r = (x_0.r + x_4.i - x_8.r - x_{12}.i) * w^0, \tag{10}$$

$$x_4^1.i = (x_0.i - x_4.r - x_8.i + x_{12}.r) * w^0, \tag{11}$$

where 'r' and 'i' respectively represent the real and imaginary components of the inputs and result. This eliminates the need to fetch the four twiddle factors from memory to be multiplied by the inputs. The IFFT is very similar; however, due to the conjugated values, the second and fourth internal twiddle factors have switched signs. For the IFFT, the Equations (10) and (11) are replaced by:

$$x_4^1.r = (x_0.r - x_4.i - x_8.r + x_{12}.i) * w^0, \tag{12}$$

$$x_4^1.i = (x_0.i + x_4.r - x_8.i - x_{12}.r) * w^0. \tag{13}$$

Another optimization performed was loop unrolling. If one butterfly operation is performed at a time, delays arise from subsequent steps within the butterfly requiring results from previous steps. For example, in Equation (12), $x_0.r - x_4.i$ is performed on cycle n; however, the result is not available until cycle $n + 4$. Therefore, if $x_8.r$ is subtracted on cycle $n + 1$, the processor will be required to stall until the result from $x_0.r - x_4.i$ has become available. An initial attempt to solve this issue was to perform four parallel butterfly operations simultaneously, referred to as a $4\times$ unroll. This means that $x_8.r$ will be subtracted on cycle $n + 4$ rather than on cycle $n + 1$, and other parallel butterfly operations will be issued during the intervening cycles. This technique, $4\times$

unrolling, eliminates any data dependencies found in the complex multiply operation. However, performing four simultaneous butterfly operations requires 16 complex inputs to the compute tiles and produces 16 complex outputs. Although the compute tiles perform optimally, this comes at the cost of memory storage delays. We investigated a 2× unroll operation where two parallel butterflies are performed simultaneously. Although a few dependency delays arose, the 2× unroll version that was finally implemented did not experience the memory issues that the 4× unroll did.

The last significant optimization was made to improve the time required to store butterfly results. Initially, results were only stored by the corner memory tiles. The side tiles were simply used to buffer results until the corner tiles were ready to store. This resulted in a backup of butterfly results, extending all the way through the sending memory tiles, causing delays in supplying inputs for the subsequent set of butterflies. To alleviate this backup, both side tiles were made to share the task of storing butterfly results. Because the backup of results does not extend through the supplying memory tiles, inputs for the next set of butterflies can be supplied earlier than if only one storing memory tile is used on each side of the chip.

3.1.3 Results

Time-Domain FIR Filter Results

Figure 8 shows the throughput in Mflop/s for the time-domain FIR kernel for both a single filter, and also dual filters across the entire Raw processor. As the time-domain FIR filter kernel is performed on a filter length $K = 256$, the results show that the throughput of the kernel is dependent on the input length. Plots for other filter lengths would show similar results. This is because each distribution of six filter elements requires loading N input vector elements from memory. As we approach an input length of 4096 elements, the throughput reaches 475 Mflop/s (950 Mflop/s across the entire chip).

The throughput in Figure 8 has one drop-off at input length 4096. Beyond this point, the input no longer fits into the 32 kBytes of cache memory, as

$$8\text{Bytes/element} * 4096\text{elements} = 32\text{kBytes}.$$

Frequency-Domain FIR Filter Results

Figure 9 shows the throughput in Mflop/s of the frequency-domain FIR kernel. As seen in the time-domain results, results for both a single filter and also dual filters across the entire Raw processor are given. The frequency-domain FIR kernel analysis differs from the time-domain FIR filter kernel in that the results below are dependent upon the input length. If the filter length were to increase or decrease, the performance of the kernel would not be effected. Based on these observations, the results between the two FIR Filter kernels are similar. For short filter sizes (64 to 1024 elements), the throughput increases as the vector size increases. As we approach a filter length of 1024 elements, the throughput reaches 102 Mflop/s (204 Mflop/s across the entire chip).

The low resolution of the results shown in Figure 9 is due to the fact that the FFT and IFFT were implemented in radix-4, which can only compute input lengths that are powers of four. For example, an input vector of length 1025 will be padded to the next greatest power of four; in turn taking just as long as an input vector of length 4096. With this said, the drop-off seen between

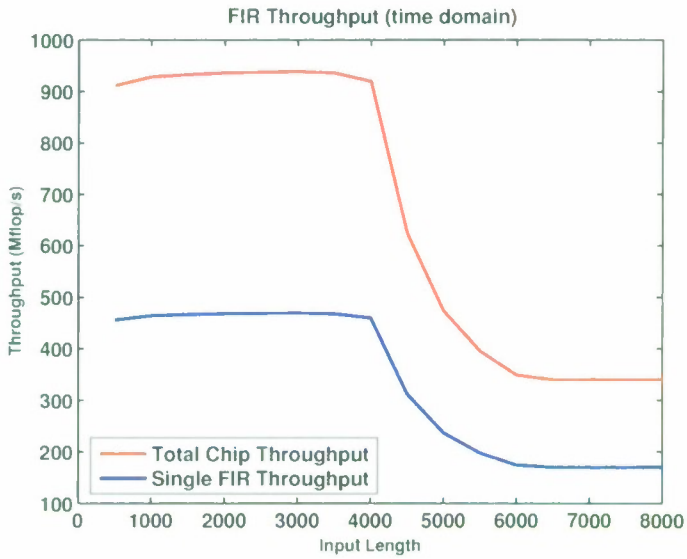

Figure 8. Time-domain FIR filter results for a filter length of 256 using the 4 × 4 Raw Handheld Board. The total chip throughput is based on parallel FIR filters (each side of the chip performs a FIR filter). The single FIR throughput is based on one FIR filter being performed on one half of the chip.

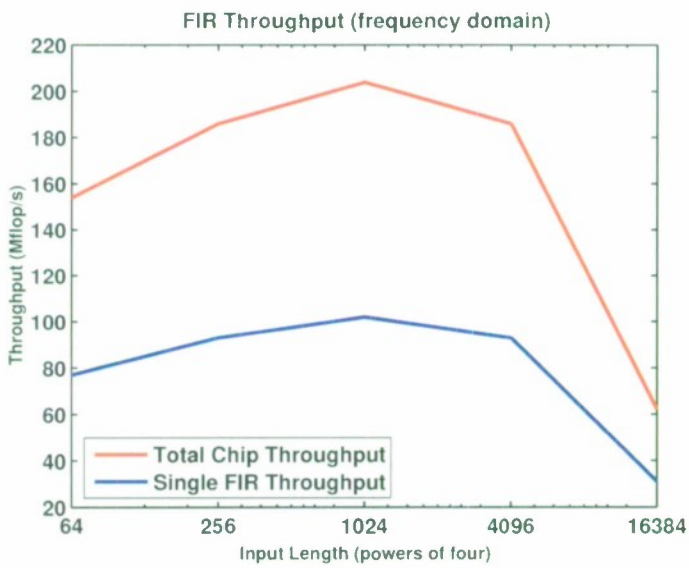

Figure 9. Frequency-Domain FIR Filter results using a 4 × 4 Raw Handheld Board. The total chip throughput is based on parallel FIR filters (each side of the chip performs a FIR filter). The single FIR throughput is based on one FIR filter being performed on one half of the chip

1024 and 4096 is probably due to the lack of data points leading up to 4096. At 4096 elements, the effect of cache misses is seen as the input vector no longer fits into cache, as

$$8\text{Bytes/element} * 4096\text{elements} = 32\text{kBytes}.$$

Frequency-Domain FIR Filter Workload Equation Due to the distributed nature of this kernel, an accurate performance analysis can only be provided with the use of a modified FFT/IFFT workload equation. The normal workload equation assumes that computations performed in a butterfly routine can be reused. This is not the case in this distributed kernel.

The conventional radix-4 FFT workload is $8.5N \log_4 N$, where $\log_4 N$ is the total number of phases, and $8.5N$ is the number of flops per phase. The number of phases in the distributed kernel does not change; however, the number of flops per phase does.

A phase can be viewed as:

```
for (N / 4 butterflies)
    Butterfly()
end.
```

The quarter of a butterfly routine on any given process performs 6 adds/subtracts for the inner component of Equation (10), and also 6 adds/subtracts for the outer complex multiplication. The 12 operations are performed on each of the 4 tiles for a total of 48 flops, or $48 * \frac{N}{4} \log_4 N$ total floating point operations; therefore, the total FFT workload is

$$12N \log_4 N. \tag{14}$$

The IFFT formula will be the same as the FFT equation with the addition of the N divides, performed as multiplies of $\frac{1}{N}$, at the end of the routine. With each complex divide being 2 flops, this results in an IFFT workload formula of

$$12N \log_4 N + 2N. \tag{15}$$

Finally, the element-wise multiply between the FFT and IFFT requires $6N$ operations. Summing the three portions of the frequency-domain FIR Filter, the total workload is

$$24N \log_4 N + 8N. \tag{16}$$

3.2 QR Decomposition

The QR decomposition is an important factorization used for least squares solutions of overdetermined systems of equations [5]. The Raw QR implementation is based on an algorithm mapping and real data implementation designed and written by Hank Hoffmann [7]. The implementation described in this document processes complex data.

3.2.1 Algorithm Description

The QR computation produces the decomposition of an $m \times n$ matrix A into the product $A = QR$, where the $m \times m$ matrix Q is orthogonal and the $m \times n$ matrix R is upper triangular [5]. The particular algorithm used for this implementation is Fast Givens. The Fast Givens algorithm loops over columns of the input matrix A, zeroing the lower triangular elements by computing and applying Fast Givens transformations over the rows. These transformations are also applied to a matrix M, that is initialized to the identity matrix, to compute the matrix Q. The Fast Givens transformations consist of the values α and β, computed to zero out the element $A(i, j)$ as,

$$\alpha = \frac{-A(i-1, j)}{A(i, j)}, \tag{17}$$

and

$$\beta = \frac{-conj(alpha)\ d(i)}{d(i-1)}, \tag{18}$$

where $d(1:n)$ are the diagonal elements of a diagonal scaling matrix, D, initialized to the identity matrix. D values are updated in every Fast Givens transformation calculation as,

$$\gamma = -\alpha\beta, \tag{19}$$

$$\tau = d(i-1), \tag{20}$$

$$d(i-1) = (1+\gamma)d(i), \tag{21}$$

$$d(i) = (1+\gamma)\tau. \tag{22}$$

The transformations are then applied to the rows of A and columns of M by,

$$A(i-1:i, j:n) = \begin{bmatrix} \beta & 1 \\ 1 & \alpha \end{bmatrix} A(i-1:i, j:n), \tag{23}$$

and

$$M(:, i-1:i) = M(:, i-1:i) \begin{bmatrix} \beta & 1 \\ 1 & \alpha \end{bmatrix}', \tag{24}$$

when zeroing element $A(i, j)$. After zeroing all lower triangular elements of the input matrix A, the scaling matrix, D, is applied to A and M, to compute:

$$Q = MD^{-1/2}, \tag{25}$$

and

$$R = D^{-1/2}A. \tag{26}$$

A Matlab program for computing the Raw Fast Givens implementation is shown in Figure 10. The algorithm described above, as well as the Matlab code shown in Figure 10 use only one type of transformation. This is done to improve the efficiency of the algorithm, but may result in a loss of numerical stability [7].

```
[m, n] = size(A);   % Compute the dimensions of the input matrix
d(1:m) = 1;         % Initialize a vector for the diagonal elements of D
M = eye(m);         % Initialize the matrix M to the identity matrix

for(j=1:n)
   for(i=m:-1:j+1)
      % Compute the Fast Givens transformation to zero A(i,j)
      [alpha, beta, d(i-1), d(i)] = fastGivens(A(i-1:i, j), d(i-1:i));

      % Apply the Fast Givens transformation to A and M
      A(i-1:i,j:n) = [beta 1; 1 alpha] * A(i-1:i, j:n);
      M(:, i-1:i) = M(:, i-1:i) * [beta 1; 1 alpha]';
   end
end

% Create a diagonal matrix D from diagonal elements in vector d
d = d.^(-1/2);
D = diag(d);

% Apply scaling matrix, D, to M and A, to compute Q and R
Q = M*D;
R = D*A;
```

Figure 10. Matlab code for computing the Fast Givens QR. The values computed within the "fastGivens" call (α, β, and updated D values) are computed as shown in Equations 17, 18, 21, and 22.

3.2.2 Mapping to Raw

The algorithm mapping specified in [7] requires a storage device accessible to each outer Computation tile. The original real QR Decomposition program simulated a 4×4 Raw chip, surrounded by simulated streaming DRAM devices. The implementation described in this document was designed to run on the Raw Handheld board, or a similar configuration, which does not have a streaming DRAM interface. To resolve this issue the outer tiles of Raw are used for I/O and data storage (to replace the streaming DRAM devices), and the inner block of tiles are used for computation. Figure 11 illustrates this for a 4×4 Raw chip.

Because the Raw chip design is meant to be scalable, the QR code was also designed with scalability in mind to allow the QR to be simulated on different Raw chip sizes. Testing on different chip sizes allows us to estimate how the performance of the QR design will scale. Figure 12 shows how an arbitrary sized Raw chip is used for the QR. There is one important difference from Figure 11 that should be discussed in detail; the configuration in Figure 12 does not assume the same I/O interface as a 4×4 Raw. The 4×4 Raw inputs and outputs to FPGAs on the Raw Handheld board via the 4 corner tiles. It is assumed that a board design for a larger Raw chip would allow more of the top and bottom tiles to perform I/O. The current implementation could easily be adapted to the 4×4 I/O interface, but the first iteration of computation would take a

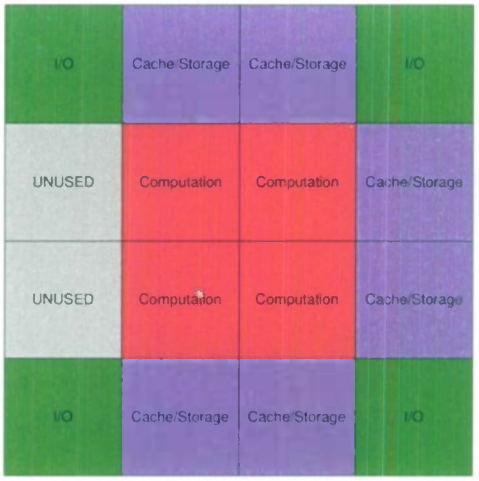

Figure 11. Usage of a 4×4 *Raw chip for performing complex QR Decomposition. The inner* 2×2 *block of tiles (in red: tiles 5, 6, 9, and 10) are used for computation. The corner tiles (in green: tiles 0, 3, 12, and 15) are used for I/O. Tiles (in blue) 1, 2, 7, 11, 13, and 14, are used as cache tiles that stream data to and from their on-tile memory and the Computation tiles. All other tiles (grey) are unused.*

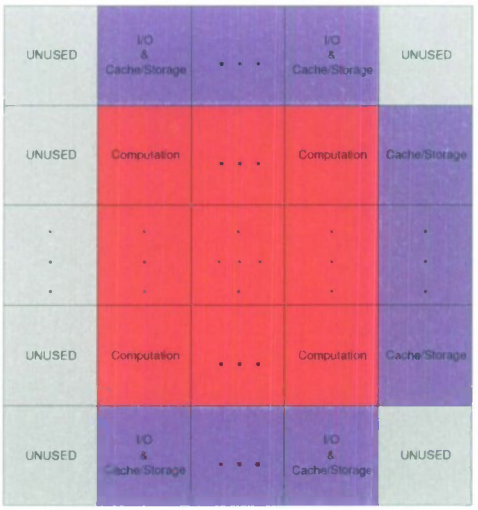

Figure 12. Usage of an arbitrary sized $R \times R$ *Raw chip for performing complex QR Decomposition. The inner* $R - 2 \times R - 2$ *tiles (red) are used for computation. Tiles 1 through* $R - 2$ *of the top and bottom rows (blue) are used for I/O (possibly; see the text) and data storage. Tiles 1 through* $R - 2$ *of the right-most column are used only for data storage. All other tiles (grey) are unused.*

performance hit because of the lower relative bandwidth to the Computation tiles. One way to get around making any board design assumptions is to time the QR in a different manner. For larger Raw chip sizes, the upper tiles receive the matrix A, store it completely to memory, then begin to stream the data into the Computation tiles. Using this method, the timing of the QR begins when the data is streamed into the computation tiles. The difference from the board results is not expected to be significant because the off-chip I/O is only required in the first iteration.

3.2.3 Implementation

Because the specific mapping required by the Raw Handheld board slightly changes the general mapping designed in [7], this section will give general implementation details and explain what happens at important stages of the algorithm for the specific 4×4 Raw chip. The resulting data storage issues will be discussed as well.

Initialization and Timing

The QR program receives inputs from the north and outputs results to the south. The organization of the inputs is handled by the PCA testbed, and passed into Raw via the High Speed I/O (HSIO) system (see Appendix A, B). The testbed appends the matrix dimensions, m and n, to the start of the data that is streamed into the I/O tiles of the chip. During the initialization phase of the algorithm, m and n are read in from the I/O tiles, and distributed to all the "working" tiles (all tiles in Figure 11 excluding those marked "UNUSED"). Once the working tiles receive m and n, the Cache tiles allocate appropriate memory. After allocating memory, initialization is complete, and each northern Cache tile stores the current Raw cycle count. Upon completion of the QR computation, the southern Cache tiles again store the current Raw cycle count. All start and finish cycle counts are sent from north and south Cache tiles to the south west Cache tile. This tile computes the total number of cycles, or time taken during the computation. The cycle count is appended to the start of the output data at the end of the program, and extracted by the PCA testbed.

QR Computation

The pseudocode shown in Figure 13 gives a high-level view of how the Matlab QR algorithm shown in Figure 10 is executed on Raw. The following sub-sections give details on how Fast Givens rotations are computed and applied on the Raw Handheld board.

As mentioned in the pseudocode comments in Figure 13, for each loop, the direction that data streams switches (i.e. north→south to south→north). During iteration i, if data is flowing from north to south, the updated values of A and M are stored in the southern Cache tiles. Therefore, in the next iteration the updated values are used, forcing data to stream from south to north. For each iteration, two rows of R, and two columns of Q are computed. Because of the change in data flow direction, every other two rows or columns of R and Q are stored in the north or south Cache tiles. The resulting storage (for an example 8×8 input) of R and Q is shown in Figures 14 and 15 respectively. Note that because of the manner in which the Fast Givens rotations are applied, Q is stored in row-major fashion and A (which is rotated to eventually contain R) is stored column-major. Upon completion of the QR, appropriate values of Q and R are streamed from the northern Cache tiles to the southern Cache tiles, and all correct rows and columns of Q and R are combined.

```
COMP_R = 2                      # Size of the computation block of tiles.
for i = 1:COMP_R:n              # Loop through columns (in sets of 2)
                                # of A. For each loop, the direction
                                # that data is streamed switches.

  fastGivens()                  # Compute Fast Givens rotations.
  for j = i:(2*COMP_R):n        # Loop on columns starting at column i.
    applyRotations(A)           # Apply rotations to A, to compute
  end                           # r and updated A.


  for j = 1:(2*COMP_R):m        # Loop through rows of M.
    applyRotations(M)           # Apply rotations to M, to compute q
  end                           # and updated M.
end

finish_Q(M)                     # If m != n, final m-n columns of Q
                                # need updating.
```

Figure 13. High-level pseudocode for Raw implementation of the Matlab algorithm shown in Figure 10.

The matrices, Q and R, are then output via the southern I/O tiles, taken by the HSIO system, and reorganized into Matlab matrices within the PCA testbed. Combining the data in this manner is an inefficient operation. The appropriate data could be streamed out of both north and south I/O tiles, and the combination of data could occur within the testbed. In fact, streaming R and Q values out via north or south I/O tiles as they are computed after each iteration would cut down on the amount of memory required by the algorithm. However, the QR was designed to be a possible sub-kernel of the Singular Value Decomposition, so R and Q are not streamed off the Raw chip. If the QR were used as a sub-kernel, the small overhead of combining the data on-chip is small relative to the overall computation. The time taken to combine the data on-chip is not included in the timing of the QR.

Fast Givens Rotation Computation. Figure 16 shows the data flow and computation that occurs during the calculation of the set of Fast Givens rotations for the first two columns of A. For subsequent iterations, data flows from the Cache tiles, and the data flow direction switches back and forth between north→south and south→north. All west↔east data flow directions remain unchanged. When the data flow direction changes, the mapping of computation changes as well. Switching the direction of data flow occurs by changing each tiles' conceptual notion of what row they are contained in, to maximize the reuse of code. When the data flow is south→north, each tile recomputes its row by computing $conceptualRow = (R - 1) - physicalRow$, where R is the number of rows or columns of tiles on the chip.

The Fast Givens rotations are computed by streaming columns of the input matrix, A, interleaved with diagonal elements of the scaling matrix, D, from the I/O or Cache/Storage tiles into

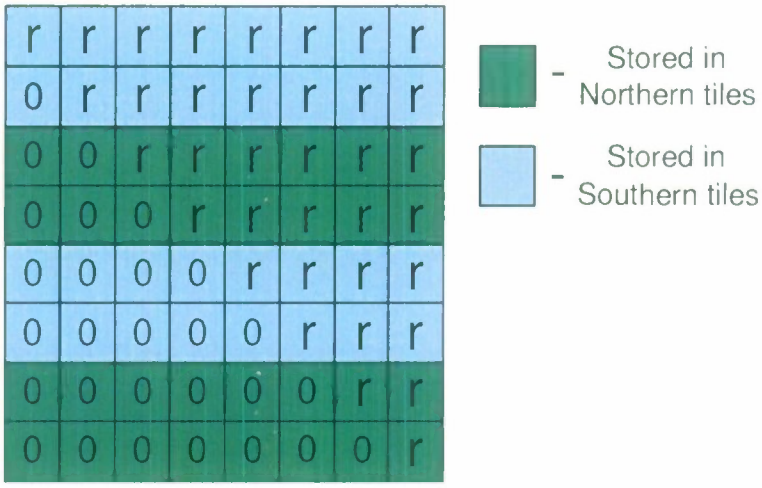

Figure 14. Storage of R after QR computation. Every other pair of rows of R is stored in either the north or south Cache tiles, because the data flow for the QR changes for each iteration of the algorithm. The zeros shown are not explicitly computed.

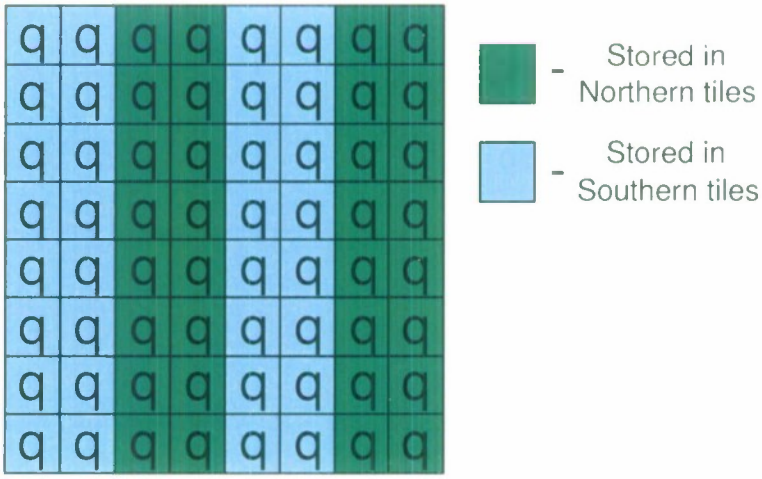

Figure 15. Storage of Q after QR computation. Every other pair of columns of Q is stored in either the north or south Cache tiles, because the data flow for the QR changes for each iteration of the algorithm.

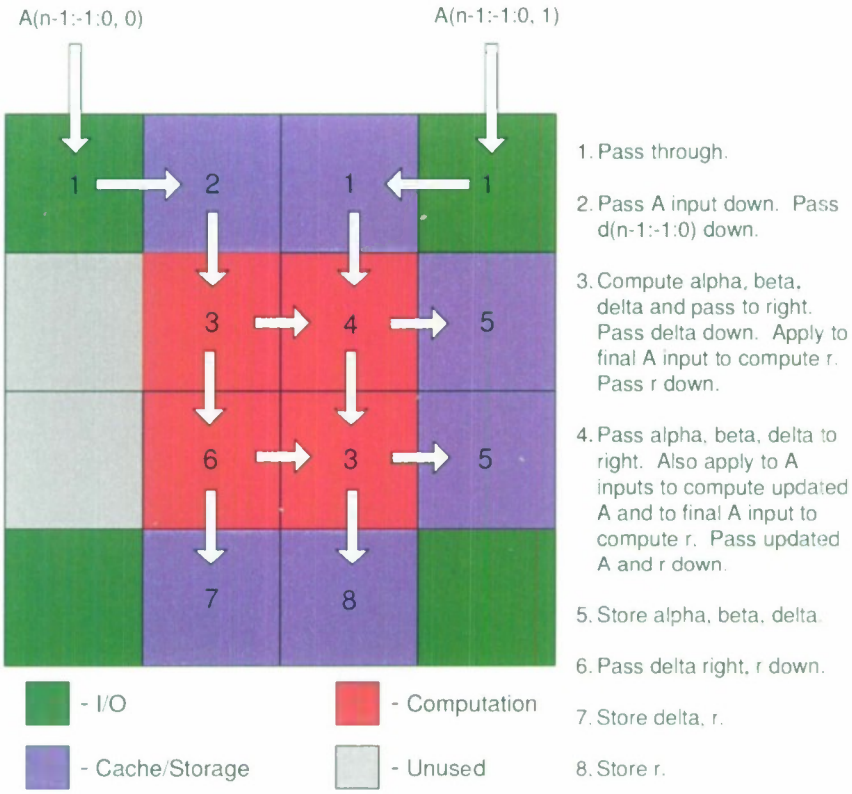

Figure 16. Data flow and computation that occurs while computing the Fast Givens Rotations. This example shows the first iteration (data flows from the I/O tiles). Fast Givens transforms are stored in eastern tiles. Updated A, R, and D values are stored in southern tiles. For subsequent iterations, data would flow from north or south Cache tiles.

the Computation tiles. During iteration i, computations are performed on columns $A(n-1:-1:2i, 2i)$ and $A(n-1:-1:2i, 2i+1)$, i.e. columns 0, 2,..., n are passed into column 1 of the chip, while columns 1, 3,..., n are passed into column 2 of the chip. As A and D values are fed into the Computation tiles, rotation values α and β are computed, as well as updated D values, and are passed to the Cache tiles on the eastern side of the chip. During the final iterations for columns $2i$ and $2i+1$, δ values are computed from the $2i$th and $2i+1$th rows of D. These values are applied to A to compute values $R(2i:2i+1, 2i:2i+1)$, then passed to the northern or southern Cache tiles.

Application of Fast Givens Rotations. Figure 17 shows the data flow and computation that occurs during the application of the Fast Givens rotations that were computed in Figure 16, to columns $i+2$ to $i+5$ of A. These same rotations are also applied in a similar fashion to rows of the matrix M. For subsequent iterations, data flow directions change as in the rotation computation phase.

The rotations are applied by streaming columns of the input matrix, A, from the I/O or Cache tiles into the Computation tiles. At the same time, rotation values α and β are streamed into the

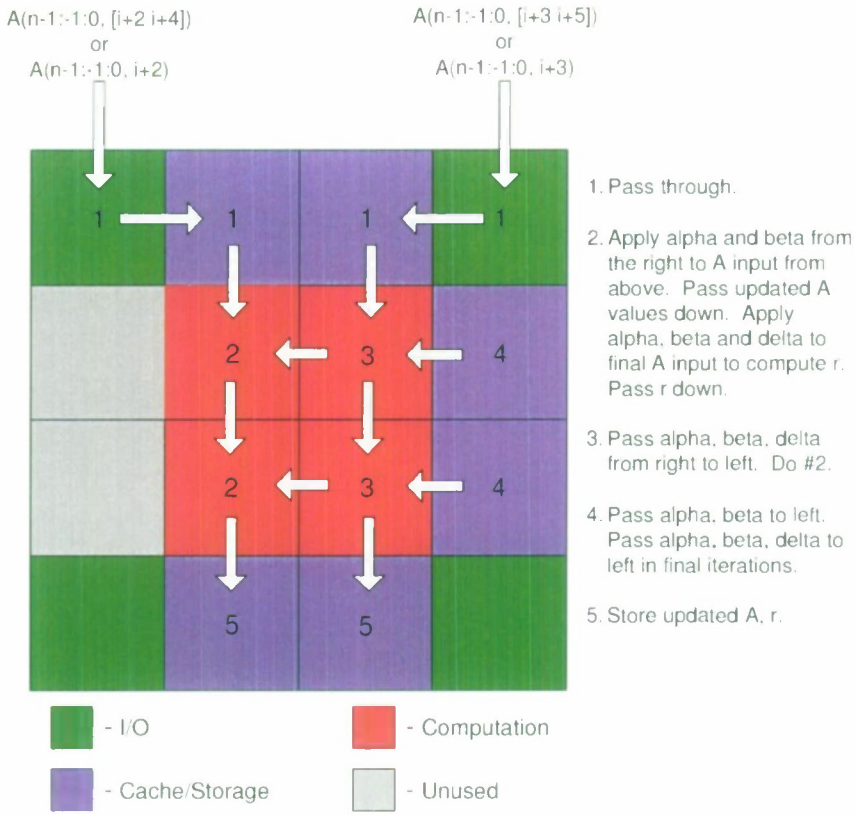

Figure 17. Data flow and computation that occurs while applying the Fast Givens Rotations to input A. This example shows the first algorithm iteration (data flows from the I/O tiles). α, β, and δ values flow from the east Cache tiles. Updated A or M values, and computed R or Q values are stored in south Cache tiles. In subsequent iterations, data (A or M) would flow from north or south Cache tiles.

Computation tiles from eastern Cache tiles. The rotations are applied to A, and the updated values are passed along for storage in the northern or southern Cache tiles. During iteration j of the ith overall update (refer to the pseudocode in Figure 13), the rotations are applied to $A(n-1:-1:2i, 4j:4j+3)$. The sub-matrix $A(n-1:-1:2i, 4j:4j+1)$ is streamed through column 1 of the chip, while $A(n-1:-1:2i, 4j+2:4j+3)$ is streamed through column 2. Each column of the chip streams its two columns in an interleaved fashion. Streaming two columns at once per column of tiles on the chip is a product of unrolling the second loop of the QR, eliminating processor data dependency stalls throughout computation for improved efficiency. In the case that the remaining number of columns is not divisible by 4, the columns are streamed through the Computation tiles one at a time. During the final iterations of the application during iteration j of overall iteration i, for columns $A(n-1:-1:2i, 4j:4j+3)$, δ values are applied to the final updated A values, computing $R(2i:2i+1, 2j:2j+1)$, which is then streamed to the northern or southern Cache tiles. Therefore, at the end of iteration i (after the QR has looped through $j = i:n$), the $2i$th and $2i+1$th rows of R have been computed.

Completing Q Computation for Tall-Thin Input Matrices. As explained in the previous paragraphs, as A and M are updated by applying Fast Givens rotations, values of Q and R are computed at the end of each iteration by computing and applying δ values (diagonal elements of the matrix $D^{-1/2}$) to the updated M and A values respectively. Recall that the resulting matrices are computed as $Q = MD^{-1/2}$, and $R = D^{-1/2}A$, where M and A have been updated via Fast Givens rotations [5]. However, for non-square input matrices, the outer loop shown in Figure 13 only loops to n, leaving $m-n$ values of δ uncomputed that still need to be applied to M to compute the final $m-n$ columns of Q. Figure 18 shows how this is accomplished.

During iteration i of the final updates to Q, the values $D(i,i)$ and $D(i+1,i+1)$ are passed from the northern or southern Cache tile in column 1 of the chip into the Computation tiles in chip column 1. The first Computation tile passes on $D(i,i)$, then reads in $D(i+1,i+1)$. From this the two Computation tiles in chip column 1 compute δ values and pass these on to column 2 of the chip. Chip column 1 then streams $M(0:2:m,[i\ i+1])$, while column 2 streams $M(1:2:m,[i\ i+1])$, both interleaving the streaming columns. M values are then passed to the appropriate Computation tiles, multiplied by δ, and the resulting Q values are output to the northern or southern Cache tiles.

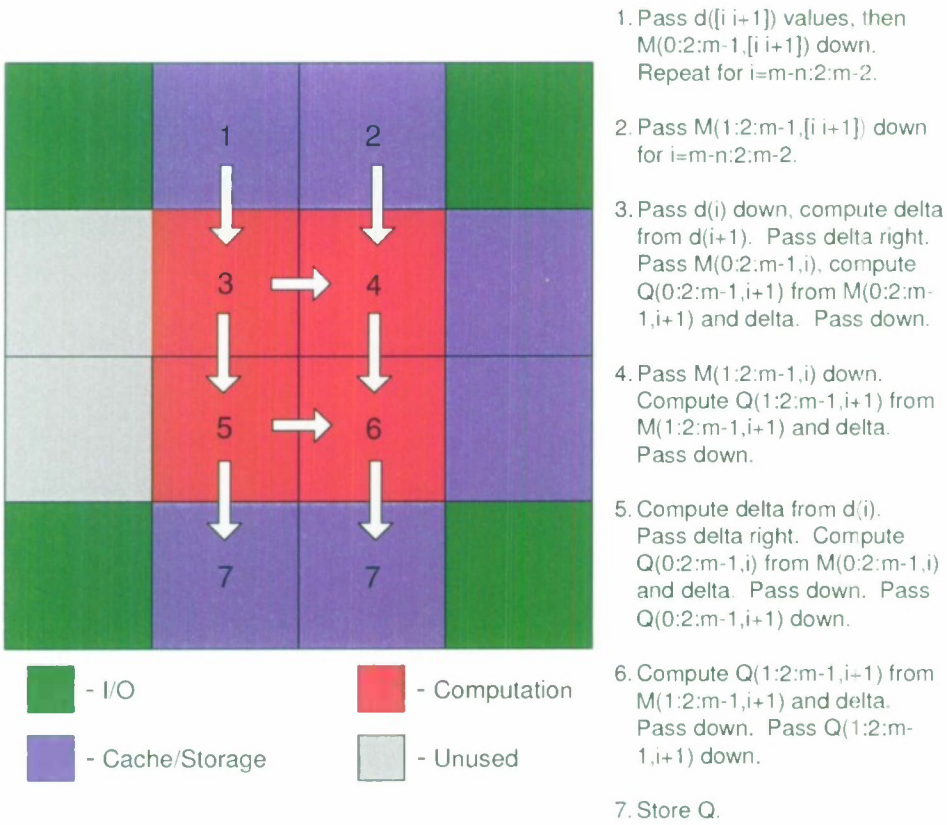

Figure 18. Data flow and computation for computing the final $m-n$ columns of Q. Data in the figure flows from north to south, but may flow from south to north, depending on the direction of data flow for the last iteration of the algorithm.

3.2.4 Benchmark Results

Performance results have been obtained for the QR benchmark running on the 4×4 Raw simulator, 4×4 Raw Handheld board, and finally on an 8×8 Raw simulator. Results will be shown for these platforms in the following sections.

4×4 Simulator and Handheld Board Results

Figure 19 shows results obtained from running the QR on square matrix inputs on the 4×4 Raw cycle-accurate simulator and Handheld board. A performance drop-off is seen when m (for an $m \times m$ input matrix, A), is equal to 64, similar to cache effects seen in results for the G4 [10]. At $m = 64$, we see that,

$$64 \text{ rows } * 64 \text{ columns } * 8 \text{ Bytes per complex element} = 32 \text{ kB (Size of matrix } Q\text{)},$$

and

$$64 \text{ rows } * 64 \text{ columns } * 8 \text{ Bytes per complex element} = 32 \text{ kB (Size of matrix } R\text{)}.$$

Because the data is divided between two storage tiles, when $m = 64$ each storage tile holds 32kB of data. A performance drop-off is seen at this point because, for larger values of m, Q and R no longer fit into the 32kB data cache contained in each of the Raw tiles.

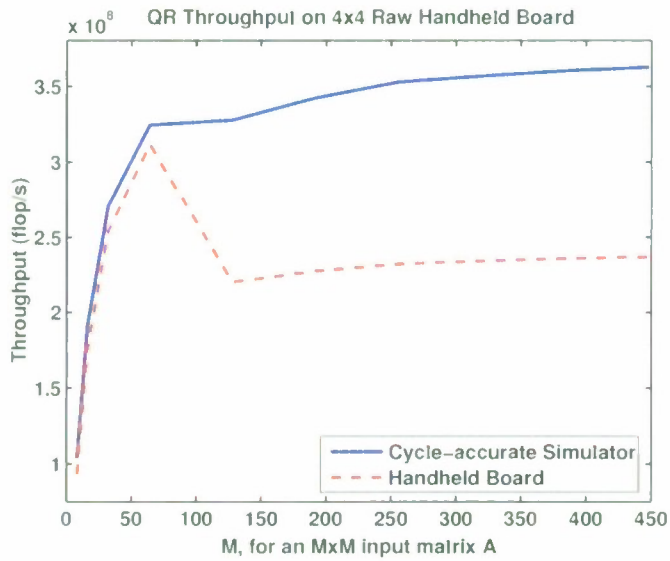

Figure 19. QR Decomposition results using the 4×4 Raw cycle-accurate simulator and 4×4 Handheld board on square matrices. Differences between the simulator and board results are discussed in detail in section 6.1.

Differences seen between the results for the simulator and Handheld board are due to DRAM access penalties not being accurately represented in the simulator. These differences are discussed in detail in section 6.1. The reason that the differences between the simulator and board are so

dramatic for the QR is due to the data access patterns of the implementation. When applying Fast Givens rotations to R throughout computation, non-contiguous data accesses are made at the beginning of each iteration due to the fact that as columns are streamed from the Cache tile, only the bottom $m - i$ values are used for iteration i. Also, switching between updates for Q and R each iteration can cause many cache conflicts for large input matrix sizes. The increased number of cache-misses, forcing reads and writes to the external DRAM, amplify the differences seen between the simulator and board results due to the inaccurate DRAM access penalties found in the simulator.

8×8 Simulator Results

QR results obtained from the cycle-accurate simulator for an 8×8 Raw are shown in Figure 20. A clock speed of 100 MHz was used to generate the throughput results to allow for easy comparison to the 4×4 Raw results. Three memory configurations were used to generate the graphs in the figure. The first configuration assumes a similar DRAM setup as for the 4×4 Raw; the DRAM is located on the eastern side of the chip only. The second configuration uses DRAM located on all four sides of the chip. The third configuration also has a DRAM on all four sides of the chip, but the simulator was modified to have DRAM read and write penalties similar to those observed on the 4×4 Handheld board. The method for how these DRAM access penalties were obtained and set is described in section 6.1.

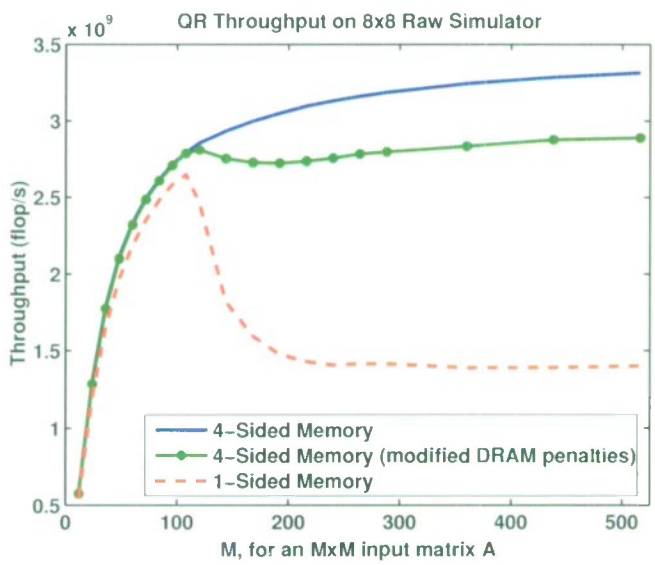

Figure 20. QR Decomposition results using the 8×8 Raw cycle-accurate simulator. Three configurations of the 8×8 simulator were used to generate the plots; A four-sided memory configuration, a one-sided memory configuration, and a four-sided memory configuration with "realistic" DRAM access penalties.

Each of the plots in Figure 20 see performance drops at similar points in the graphs; when

$m = 110$. At this point we see that,

$$110 \text{ rows } * 110 \text{ columns } * 8 \text{ Bytes per complex element} = 97 \text{ kB (Size of matrix } Q),$$

and

$$110 \text{ rows } * 110 \text{ columns } * 8 \text{ Bytes per complex element} = 97 \text{ kB (Size of matrix } R).$$

Because the two matrices are divided over 6 Cache tiles,

$$(97 \text{ kB} + 97 \text{ kB})/6 \text{ Cache tiles} = 32 \text{ kB per Cache tile.}$$

As the input data exceeds this size, Q and R begin to fall out of the data cache located on the Cache tiles, and a degradation or leveling off of performance is seen. The cache effects are more pronounced in the 1-sided memory configuration because of memory dynamic network conflicts when multiple Cache tiles make DRAM requests.

Scalability Analysis

Figure 21 demonstrates the possible performance advantages of scaling the size of the Raw chip. Throughput graphs are taken from 4×4 Handheld board results and compared with 8×8 simulator results for running the QR with square input matrices. The 8×8 simulator was run using four-sided memory with modified DRAM access penalties modeled after those found for the 4×4 board. On average, for points plotted in Figure 21, the 8×8 simulator outperforms the Handheld board by a factor of 11. This superlinear[1] factor in performance is seen due to the increased memory accessibility found while locating a DRAM on all four sides of the Raw chip.

The peak achievable throughput for the 4×4 Raw QR mapping, which uses 4 out of 16 tiles for computation, is

$$25\% * 1.6 \text{ Gflop/s (Peak for entire } 4 \times 4 \text{ chip)} = .4 \text{ Gflop/s.}$$

The peak achievable throughput using the 8×8 mapping, which uses 36 out of 64 tiles for computation, is

$$56\% * 6.4 \text{ Glop/s (Peak for entire } 8 \times 8 \text{ chip)} = 3.6 \text{ Gflop/s.}$$

The potential of using a higher relative number of Computation tiles for larger tiled-array chip sizes allows for more efficient use of the entire chip. Also, the fact that the efficiency of the algorithm on the Computation tiles remains consistent for larger chip sizes shows that linear performance improvements can be obtained by scaling the size of the Raw chip for streaming algorithms such as the QR decomposition.

3.2.5 Further Optimizations

Given the current board design, improvements could be made to the current implementation to alleviate cache effects seen for large input matrices. One possible solution is to separate the computations for Q and R. Updates could be applied to A until R is fully computed, then applied

[1] The 8×8 Raw QR mapping uses 9 times the number of Computation tiles as the 4×4 mapping.

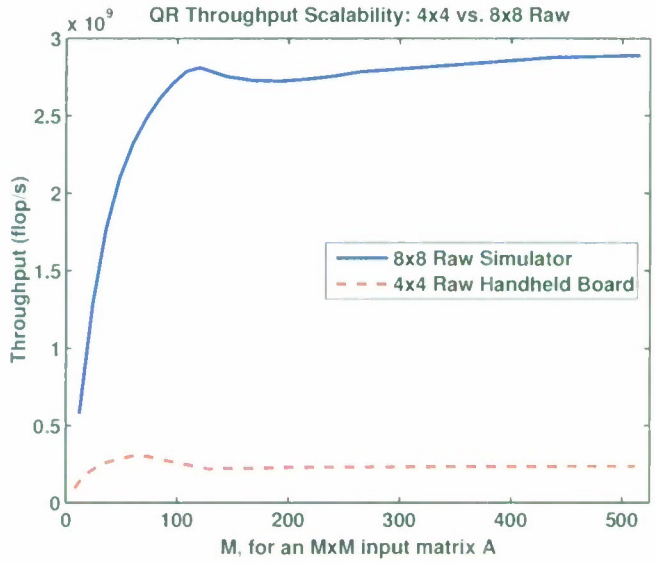

Figure 21. QR Decomposition results using the 8×8 Raw cycle-accurate simulator vs. the 4×4 Raw Handheld board results. The 8×8 simulator results used a four-sided memory configuration, with modified DRAM access penalties modeling those observed on the 4×4 Handheld board.

to M to compute Q. This would eliminate the cache conflicts due to switching back and forth between the matrices during each iteration of the algorithm, but would force the storage of all of the Fast Givens rotation matrices for each update. There is a possibility that storing the rotations could introduce delays.

More significant optimizations could be achieved if the Raw chip were used on a board designed for streaming applications. Surrounding the periphery of the chip with streaming memory devices would remove the necessity of the Cache tiles, allowing the entire chip to be used for computation. Even without streaming memory devices, cache effects could be reduced by placing DRAMs on more than one side of the chip. Evidence of this is seen in the 8×8 simulator results (Figure 20), where dramatic improvements are seen for a four-sided memory configuration over a one-sided configuration.

3.3 Singular Value Decomposition

The singular value decomposition (SVD) is of increasing importance in signal processing. It is an advanced linear algebra operation that produces a basis for the row and column space of the matrix and an indication of the rank of the matrix. In adaptive signal processing, the matrix rank and the basis are useful for reducing the effects of interference [11].

3.3.1 Algorithm Description

Given an $m \times n$ complex matrix A, the singular value decomposition of A is

$$A = U\Sigma V^H, \tag{27}$$

where U is a unitary matrix of size $m \times n$, Σ is an $n \times n$ diagonal matrix with all entries real and sorted in descending order, and V is an $n \times n$ unitary matrix.

The algorithm chosen to implement the SVD is the Stream Hestenes SVD algorithm (or simply Stream SVD) proposed by Strumpen, Hoffmann, and Agarwal [19]. The algorithm is based on the Hestenes-Jacobi method, applying Jacobi rotations to decompose the input matrix A [5]. What is unique about the Stream SVD method is that it computes and applies rotations in blocks. While this sacrifices speed of convergence, it allows for a parallel implementation that is highly suitable for an architecture such as Raw.

3.3.2 Mapping to Raw

The algorithm mapping specified in [19], similar to the QR mapping, requires a storage device accessible to each outer computation tile. The implementation described in this report was designed to run on the Raw Handheld board, which does not have such a storage device accessible at the periphery of the chip. To resolve this issue the outer tiles of Raw are used for I/O and data storage, and the inner block of tiles are used for computation. Figure 22 illustrates this for a 4×4 Raw chip.

The present Raw implementation of the SVD only computes Σ. A method for calculating U and V is described in the algorithm mapping document [19]. We do not expect the performance on the Raw processor to significantly change due to the calculations required for computing U and V. The application of Jacobi transformations in the computation of U can be performed in an efficient streaming manner similar to the computations performed to compute Σ. The divisions required in the computation of V, where

$$B = U^T A, \tag{28}$$

$$\sigma_i = ||B_i||, \tag{29}$$

and

$$v_i^T = \frac{B_i}{||B_i||}, \tag{30}$$

are insignificant compared to the overall work required to compute Σ and U. The only foreseeable negative affect due to the computation of U and V is the potential increase of cache misses encountered in the computations due to the extra memory usage required to store the matrix U. The number of cache misses can be minimized by storing the Jacobi rotations and applying them separately in the updates to A and U; however, an increase in latency is expected. The magnitude of the increased latency will be data set size dependent.

3.3.3 Implementation

This section describes the specific implementation of the SVD for the Raw Handheld board. Details of this implementation are slightly different than the general mapping described in [19].

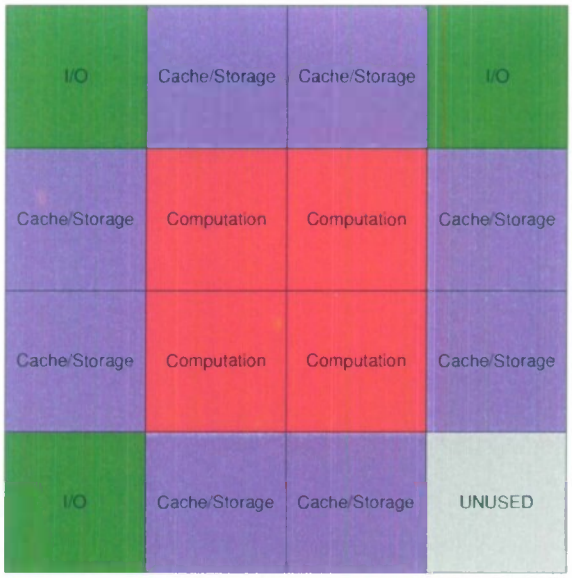

Figure 22. Usage of a 4 × 4 Raw chip for performing the complex Stream Hestenes SVD. The inner 2 × 2 block of tiles (in red: tiles 5, 6, 9, and 10) are used for computation. The corner tiles except for the southeastern corner (in green: tiles 0, 3, 12) are used for I/O. Tiles (in blue) 1, 2, 4, 7, 8, 11, 13, and 14, are used as memory tiles that will stream data to and from their memory and the computation tiles. The 15th, or southeastern-most tile (grey) is unused.

Initialization and Output

The SVD receives inputs from the north and outputs the resulting matrix and timing information out of the southwest-corner I/O tile. The organization of the inputs is handled by the PCA testbed and passed into Raw via the High Speed I/O system (see Appendix A, B). The testbed takes the input matrix, A, and divides the rows of the matrix between the two I/O tiles. These rows are streamed into the I/O tiles in a *snaked row distribution*[2]. Along with reorganizing the input data, the testbed will append the matrix dimensions, m and n, to the start of the data that is streamed into the I/O tiles of the chip. During the untimed initialization phase of the algorithm, m and n are read in from the I/O tiles, and distributed to all the "working" tiles (all tiles in Figure 22 excluding those marked "UNUSED") on the chip. The working tiles receive m and n and allocate any necessary memory. After allocating memory, initialization is complete, a synchronization or barrier function is called, the current Raw cycle count is stored, and the SVD function is called. After completing the SVD, another synchronization or barrier function is called, and the final Raw cycle count is stored. The beginning and ending cycle counts are then output to the testbed along with the resulting matrix, Σ.

[2]Tile 1 will receive rows 0, 3, 4, 7, 8, ..., m, and tile 2 will receive rows 1, 2, 5, 6, ..., m. The input row indices can be computed by interleaving values [0:4:m] and [3:4:m] for tile 1, and interleaving values [1:4:m] and [2:4:m] for tile 2.

SVD Computation

Figure 23 shows a pseudocode example for the Stream SVD computation on Raw. The calculation of row norms is performed on the I/O or Cache tiles, and the Jacobi calculation and applications are performed on the Computation tiles. Each of the tiles will perform computations on different row pairs in parallel due to the use of block transformations within the SVD algorithm [19]. The following sections discuss the details of the computations outlined in Figure 23.

```
COMP_R = 2                          # Size of the computation block of tiles.
do                                  # Loop until convergence criteria are met.
  for i=0:COMP_R:m                  # Loop over rows of A in blocks of 2.
    j=i+1
    calculateRowNorms(i,j)
    computeJacobi(i,j)              # Perform boundary operations for
    applyJacobi(i,j)                # rows i and j.

    for ib=i+COMP_R:COMP_R:m        # Loop over remaining rows after row j.
      jb=ib+1
      calculateRowNorms(i,j,ib,jb) # Perform non-boundary operations:
      computeJacobi(i,j,ib,jb)      # Compute and apply transformations in
      applyJacobi(i,j,ib,jb)        # parallel for rows i, j, ib, jb.
    end
  end
while(convergence criteria not met)

calculateSigma()                    # Calculate Sigma from row norms of A.
```

Figure 23. High-level pseudocode of the Stream SVD implementation for Raw. Calculation of the row norms is performed on the I/O or Cache tiles, while the Jacobi transformation calculation and application is performed on the Computation tiles.

Testing for Convergence. Within the SVD computation, a δ value is calculated and used in each iteration to determine whether convergence to an orthogonal matrix has been achieved. This value is calculated by summing the norms of each row in the input matrix, A, and multiplying by the floating point precision ϵ. Because the implementation is performed on Raw in a streaming manner, δ cannot not be calculated before the SVD without streaming in the entire matrix before the computation begins. Therefore, it is assumed that the input matrix is not orthogonal (i.e. it will not meet convergence in the first block sweep of the algorithm), and δ is computed throughout the first block sweep. As each row is first streamed into the chip the I/O tiles calculate the row norm as they stream the values into the Cache tiles. The row norms are summed, then passed on to each Computation tile at the end of the first block sweep, where δ is computed. In subsequent block sweeps, the calculated δ value will be used for checking the convergence.

Computing Jacobi Rotations. The first step of the SVD algorithm is to compute the row norms, $S(i)$ and $S(j)$, of rows i and j respectively, which will be used in the Jacobi rotation calculation. The row norm calculations are performed on the I/O tiles when rows i and j are first streamed onto the chip, and computed on the Cache tiles for subsequent iterations. As the data for rows i and j stream onto the chip, the value, g_{ij}, for rows i and j is computed as

$$g_{ij} = A(i,:) * A(j,:)'. \tag{31}$$

The value, g, is compared with δ where a value of $|g|$ greater than δ means that convergence has not been met for the current block sweep. The value, g_{ij}, is used with $S(i)$ and $S(j)$, the norm values for rows i and j, to calculate the Jacobi rotation values;

$$[c, s] = jacobi(S(i), S(j), g_{ij}). \tag{32}$$

The Jacobi rotation values are calculated for a complex input as follows:

$$w = \frac{S(j) - S(i)}{2 * g_{ij}}, \tag{33}$$

$$t = \frac{sign(w)}{|w| + \sqrt{1 + w^2}}, \tag{34}$$

$$c = \frac{1}{\sqrt{1 + t * conj(t)}}, \tag{35}$$

$$s = t * c. \tag{36}$$

Jacobi rotations are computed for only two rows of the input matrix when the data that must be rotated resides on a single side of the chip. This *boundary* case can be seen in Figure 24. This figure describes the data flow and computation that occurs upon the start of the SVD algorithm. At the start, no data has been streamed into the chip, so it conceptually resides on only the northern side of the chip. In this case, Jacobi rotations will be computed for rows $i = 0$ and $j = 1$. After the application of the rotations to rows 0 and 1, the updated rows will reside in the eastern Cache tiles (tiles 7 and 8 respectively). For all remaining updates using rows 0 and 1, Jacobi rotations will be computed for four row-pairs simultaneously because data may be streamed from north→south and east↔west at the same time. Figure 25 shows how this is performed for computing Jacobi rotations for the rows 0 and 1 with rows 2 and 3. During this step, rows 0 and 1 are streamed from the eastern Cache tiles to the west, while rows 3 and 2 (recall that data is input in a *snaked row distribution*), are streamed into the chip and flow north→south. As the values cross on the Computation tiles, g is computed for each of the row pairs, then used to calculate the Jacobi rotations.

Applying Jacobi Rotations. After the Jacobi rotations have been calculated for a given row pair, i and j, the rotations are applied to the rows. For a row index, k, in row i and j, the values are rotated as follows:

$$temp = A(i,k), \tag{37}$$

$$A(i,k) = c * A(i,k) - conj(s) * A(j,k), \tag{38}$$

$$A(j,k) = s * temp + c * A(j,k). \tag{39}$$

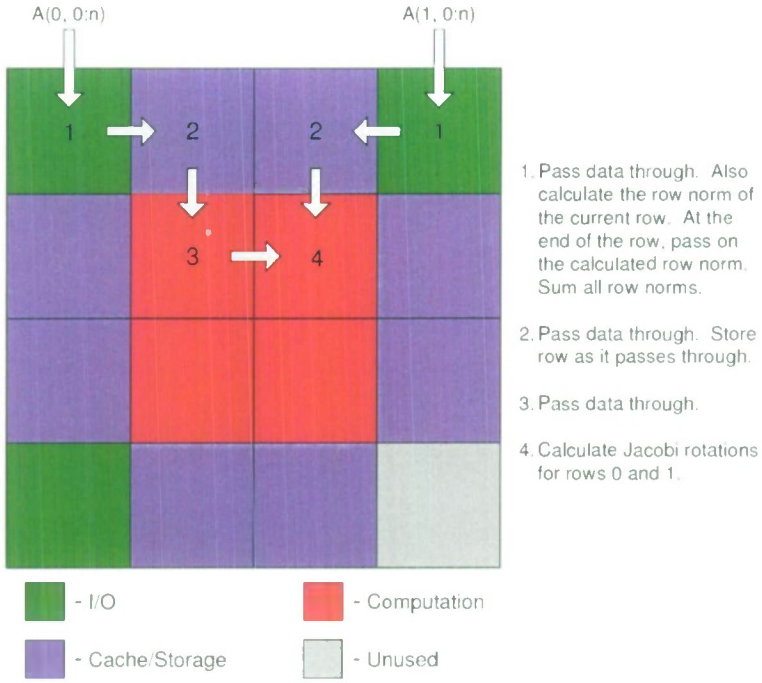

Figure 24. Data flow and computation that occurs while computing Jacobi rotations for a boundary case.

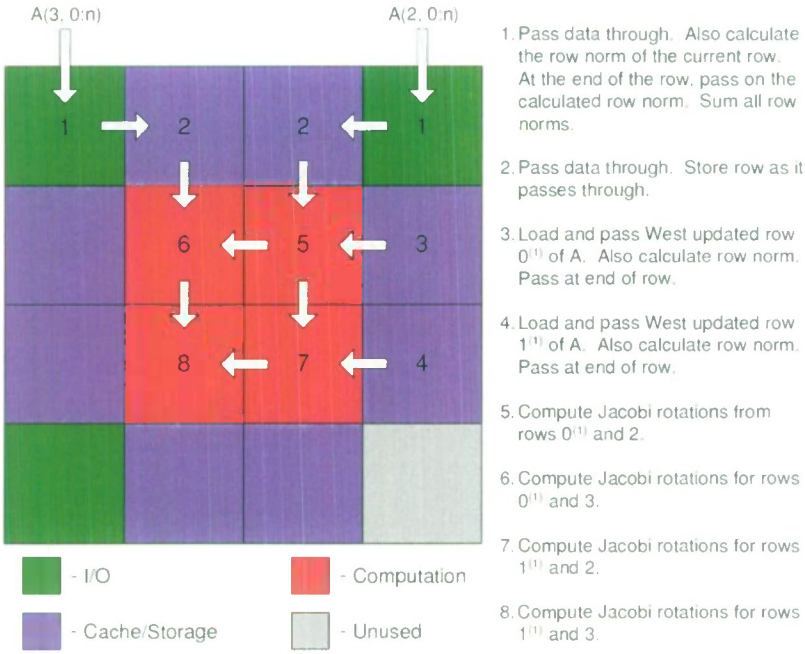

Figure 25. Data flow and computation that occurs while computing Jacobi rotations for a non-boundary case. Superscript values denote the number of times the row has been rotated.

As with the computation of the Jacobi transformations, rotations are applied to only two rows at once in the boundary situation where the data to rotate resides on one side of the chip. The data flow for the boundary case is seen in Figure 26. The computation shown in this figure directly follows the computation of the rotation values shown in Figure 24. In Figure 26, rows 0 and 1 are streamed into the Computation tiles, and the Jacobi rotations are applied on tile 6. The updated values of rows 0 and 1 are then stored in the eastern Cache tiles (tiles 7 and 11 respectively). For all remaining applications of Jacobi matrices using rows 0 and 1, Jacobi rotations will be applied for four row-pairs simultaneously because data may be streamed from north→south and east↔west at the same time. An example of this type of rotation application is shown in Figure 27. In this figure, the Jacobi rotations calculated in the step shown in Figure 25 are applied to updated rows $0^{(1)}$ and $1^{(1)}$, and rows 3 and 2, where superscript values denote the number of times the row has been rotated. Rows $0^{(1)}$ and $1^{(1)}$ are streamed west from tiles 7 and 11 respectively, while rows 3 and 2 are streamed south from tiles 2 and 1 respectively. Jacobi rotations are first applied to rows $0^{(1)}$ and 2 on tile 6. The output values of $0^{(2)}$ are streamed to the west, and $2^{(1)}$ are streamed south. To the west, on tile 5 row values $0^{(2)}$ are rotated with row 3, outputting $0^{(3)}$ to be stored in the western Cache tile 4, and row values $3^{(1)}$ streamed to the south. Concurrently, tile 10 processes row values $1^{(1)}$ with $2^{(1)}$, outputting $2^{(2)}$ to be stored in the southern Cache tile 14, and row values $1^{(2)}$ streamed to the west. Also, tile 9 rotates row values $1^{(2)}$ and $3^{(1)}$, outputting $1^{(3)}$ to be stored in the western Cache tile 8, and $3^{(2)}$ to be stored in the southern Cache tile 13.

SVD Block Sweep. Figure 28 combines the computations shown in Figures 24, 25, 26, and 27. The figure also shows the data flows for the remaining steps necessary to complete one block sweep of the Stream Hestenes SVD algorithm for an input matrix containing 6 rows. The odd steps in the figure show the computation of Jacobi rotations. The even steps show the application of the rotations. The first four steps shown in the figure correspond to Figures 24, 26, 25, and 27 respectively. The fifth subdiagram shows the computation of Jacobi values for rows $0^{(3)}$ and $1^{(3)}$ with rows 4 and 5, with the application of the Jacobi rotations illustrated in the sixth subdiagram. The outputs, row values for $0^{(5)}$ and $1^{(5)}$, are stored in the eastern Cache tiles, and updated values for rows $4^{(2)}$ and $5^{(2)}$ are streamed to the southern Cache tiles. At this point rows 0 and 1 have been rotated against all other rows in the input matrix and require no further rotations for this block sweep. After this point, another boundary condition is found in the seventh subdiagram as all remaining rows (2 through 5) are located in the southern Cache tiles. Subdiagram seven shows the computation of Jacobi rotations for the two rows $3^{(2)}$ and $2^{(2)}$. The application of the Jacobi rotations is shown in subdiagram eight where the updated values for rows $3^{(3)}$ and $2^{(3)}$ are streamed to the western Cache tiles. Subdiagrams nine and ten show the rotation of rows $3^{(3)}$ and $2^{(3)}$ with $4^{(2)}$ and $5^{(2)}$, resulting in rows $4^{(4)}$ and $5^{(4)}$ being stored in the northern Cache tiles, and the final values (for this block sweep) for rows 3 and 2 ($3^{(5)}$ and $2^{(5)}$) stored in the eastern Cache tiles. Finally the boundary condition is found for rows 4 and 5 in the remaining two subdiagrams, where the resulting rows, $4^{(5)}$ and $5^{(5)}$, are stored in the eastern Cache tiles as well.

Throughout the block sweep, on each Computation tile a convergence flag is kept and set to *false* if any $|g_{ij}|$ value is greater than δ. At the end of the block sweep, the convergence flags are all sent to tile 0, where a logical *and* is performed on the flags, and the resulting value is passed back to each tile. If the returned flag value is *true*, the resulting singular values are computed and sorted (this is discussed in the next section) then output to the PCA testbed. If the returning convergence

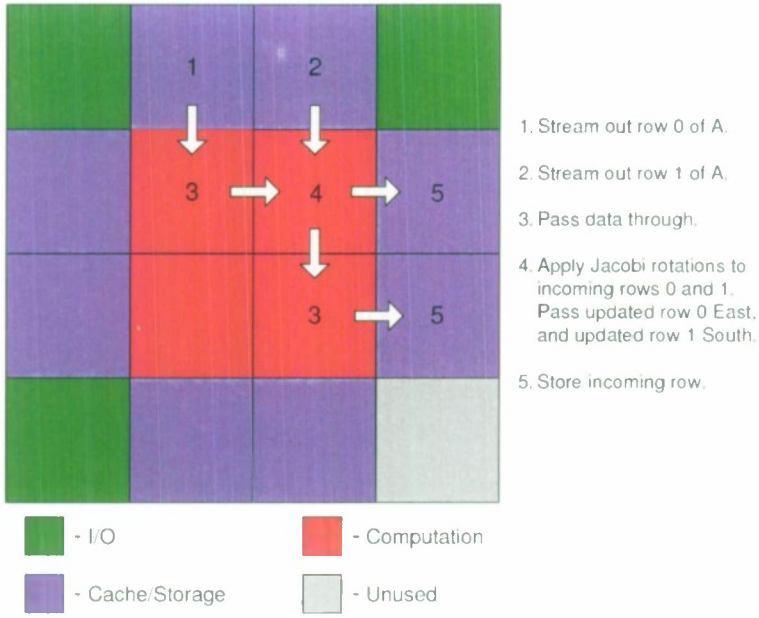

Figure 26. Data flow and computation that occurs while applying Jacobi rotations for a boundary case.

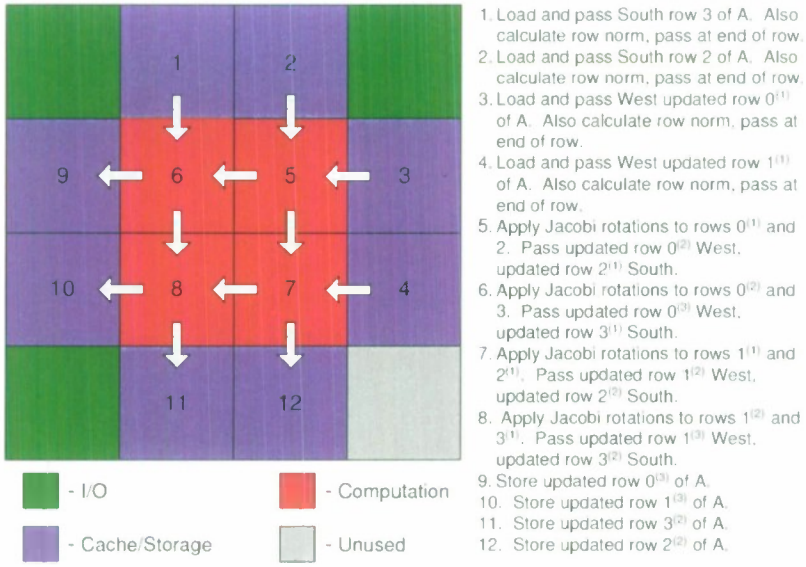

Figure 27. Data flow and computation that occurs while applying Jacobi rotation for a non-boundary case.

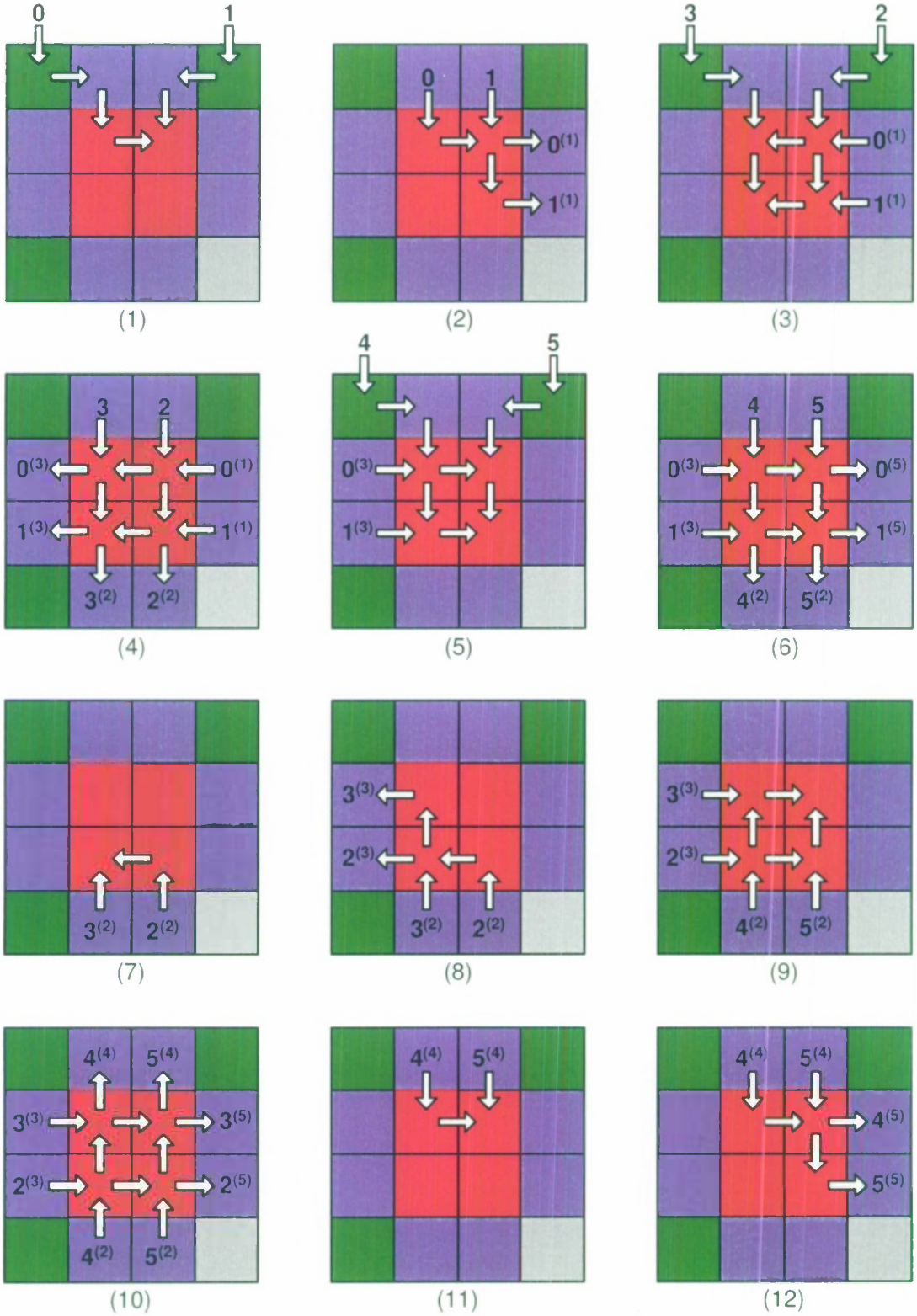

Figure 28. Data flow for a single block sweep of the Stream Hestenes SVD on an input matrix containing 6 rows. Numbers within the subdiagrams represent the row number being read, processed, or stored. Superscript numbers denote the number of times the row has been rotated. This process is repeated until convergence criteria are met.

flag is *false*, the block sweep process is repeated until convergence criteria are met.

The location of the rotated data at the end of a block sweep will be different than the location of the data at the start of the sweep. In Figure 28, the data is located north of the chip in the beginning of the sweep, and in the eastern Cache tiles at the end. Therefore data flows and computations for subsequent block sweeps occur in different directions and locations. This is accomplished by changing each tile's conceptual notion of its location in the grid (virtually rotating the grid), requiring a complex coordination of the computation and directional routing of data. This coordination is also dependent upon data size. In Figure 28, if the input matrix, A, contained 8 rows (or any number divisible by 4), the data would have been located in the western Cache tiles at the end of the block sweep instead of in the eastern tiles.

Computing Σ. Once the convergence criteria has been met for the rotated matrix, Σ values are computed by taking the square root of the norm of each row in the matrix. These values are then sorted using a *bubble sort* algorithm [1], and output to the PCA testbed.

3.3.4 Benchmark Results

Figure 29 shows results obtained from running the SVD on square matrix inputs on the 4×4 Raw cycle-accurate simulator and Handheld board. The performance graph levels off when m, for an $m \times m$ input matrix, A, is equal to approximately 90. This is similar to cache effects seen in results for the G4 [10]. At $m = 90$, we see that,

$$90 \text{ rows } * 90 \text{ columns } * 8 \text{ Bytes per complex element} = 64 \text{ kB (Size of input matrix } A).$$

Because the data is divided between two storage tiles, when $m = 90$ each storage tile holds 32kB of data. A performance drop-off is seen at this point because, for larger values of m, the input matrix will no longer fit into the 32kB data cache contained in each of the Raw tiles.

Results for the Raw cycle-accurate simulator are only provided for data sets up to a 192×192 input matrix due to excessive simulator time requirements for the SVD. However, it is expected that the results should level off similar to the cycle-accurate simulator results seen for the QR kernel benchmark (see Figure 19). Differences seen between the results for the simulator and Handheld board are due to DRAM access penalties not being accurately represented in the simulator. These differences are discussed in detail in section 6.1.

3.3.5 Workload Considerations

The Stream SVD document, [19], defines the total number of multiply-and-add operations of the SVD for processing real data to be

$$((5n + 4) * m(m - 1)/2) * it, \tag{40}$$

where it represents the number of iterations or block sweeps required for the algorithm to converge. Therefore, we define the workload, in flop, for the complex implementation to be

$$8 * ((5n + 4) * m(m - 1)/2) * it, \tag{41}$$

because there are 8 flop required to perform each complex multiply-and-add operation.

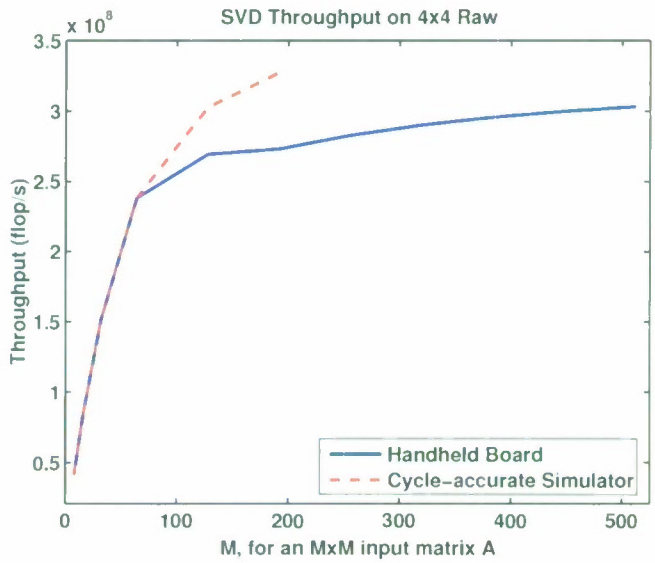

Figure 29. SVD results using the 4 × 4 Raw cycle-accurate simulator and 4 × 4 Handheld board on square matrices. Differences between the simulator and board results are discussed in detail in Section 6.1.

Table 1.

Comparison of SVD Workloads.

	Stream Hestenes SVD Real Data		Stream Hestenes SVD Complex Data		Golub-Kahan SVD Complex Data
Matrix Size	Iterations	Workload (Mflop)	Iterations	Workload (Mflop)	Workload (Mflop)
16 × 16	7	0.14	16	1.29	0.26
32 × 32	7	1.14	16	10.41	2.04
64 × 64	8	10.45	15	78.38	16.20
128 × 128	9	94.22	17	711.88	129.12
256 × 256	9	754.38	15	5029.17	103.10
128 × 32	7	18.66	18	191.95	3.61
128 × 64	9	47.40	18	379.22	20.40
256 × 32	7	74.94	17	728.00	5.71
256 × 64	8	169.21	16	1353.65	28.78
256 × 128	8	336.32	17	2858.74	162.67

Experimentation has shown that using the Stream SVD algorithm to process *complex* single-precision floating-point data requires a significantly higher number of iterations or block sweeps to converge relative to the number required for *real* single-precision floating-point data. Consequently, workloads for processing complex data using the Stream Hestenes algorithm are considerably larger than using a more conventional algorithm such as the Golub-Kahan SVD, described in [5]. Table 1 compares the number of iterations required for convergence, and total workloads for real vs. complex single-precision floating-point data run using the Stream SVD algorithm. A floating-point precision $\epsilon = 1^{-7}$ was used for both real and complex simulations. Also shown in the table are total workloads required for the Golub-Kahan SVD algorithm for processing complex single-precision floating-point data. The workload equation used for the Golub-Kahan SVD algorithm is discussed in [11].

3.4 Constant False Alarm Rate Detection

3.4.1 Algorithm Description

As described in the PCA Kernel Benchmark Report [11], the constant false-alarm rate (CFAR) detection algorithm finds targets in an environment of varying background noise. Assume a data cube whose dimensions are number of beams N_{bm}, number of range gates N_{rg}, and number of dopplers N_{dop}. For each cell in the data cube, a local noise estimate is computed from the $2N_{cfar}$ range gates nearest to the cell $C(i, j, k)$ under test that are at least a certain number of guard cells G away from $C(i, j, k)$. Formally, for each cell $C(i, j, k)$, the value of the noise estimate $T(i, j, k)$ is calculated as

$$T(i, j, k) = \frac{1}{2N_{cfar}} \sum_{l=G+1}^{G+N_{cfar}} |C(i, j + l, k)|^2 + |C(i, j - l, k)|^2. \tag{42}$$

The range cells involved in calculating the noise estimate for a particular vector are shown in Figure 30. For each cell $C(i, j, k)$, the quantity $|C(i, j, k)|^2/T(i, j, k)$ is calculated; this represents the normalized power in the cell under test. If this normalized power exceeds a threshold μ, the cell is considered to contain a target.

3.4.2 Implementation on Raw

For the CFAR kernel, the defined data sets are larger than the buffer size available with the HSIO interface. Therefore, the Raw CFAR kernel has two variants, an HSIO variant and a USB variant. We describe each of these in this section.

The Raw implementation of the CFAR using the High-Speed I/O (HSIO) interface splits a Raw chip of size 4×4 into four quadrants of size 2×2. Each of these quadrants consist of three types of tiles, one I/O tile, one forwarding tile, and two leaf tiles. The I/O tile is adjacent to an external I/O port and handles all the external I/O for the quadrant. The forwarding tile forwards data and results to and from the leaf tile that is not adjacent to the I/O tile. Of the two leaf tiles, one is adjacent to the I/O tile and one is adjacent to the forwarding tile. The I/O tile thus communicates solely with its adjacent leaf tile and forwarding tile, and the forwarding tile communicates solely with the forwarding tile adjacent to it. See Figure 31 for a diagram of this arrangement for a 4×4 chip. Note that only the static network is used for inter-tile communication.

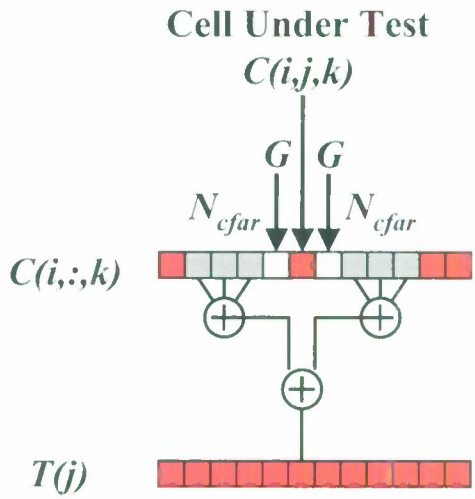

Figure 30. Sliding window in CFAR detection. The example shows the number of guard cells $G = 1$ and the number of cells used in computing the estimate $N_{cfar} = 3$.

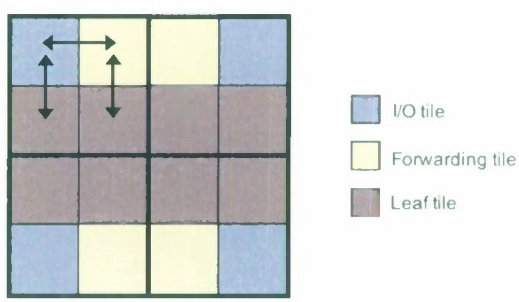

Figure 31. CFAR's use of tiles on the MIT Raw. Thick black lines delineate the quadrants on the chip. The arrows are drawn only on the upper-left quadrant and indicate the tiles which communicate.

This implementation takes the data cube and flattens it into a series of one-dimensional rows of size N_{rg}, where each row represent elements from the same beam and doppler. These rows are then divided among the four quadrants and streamed into the I/O ports. The I/O tile will stream one row to each tile in its quadrant. As data is streamed in, each tile looks for targets. For every cell under test, a 32-bit word is returned. This word equals 0 if there are no detections; otherwise, the word encodes the beam (8 bits), doppler (8 bits), and range gate (16 bits) in which the target was found. Because of this encoding, the data cube can have at most 256 beams, 256 dopplers and 65536 range gates. Note that this can be easily extended by sending the beam, doppler, and range gate as separate numbers; however, current encoding is sufficient for data sets of interest. These detection reports are sent off-chip by the I/O tile, either directly or through the forwarding tile.

The USB version differs in that only the left half of the chip is used. Since the right half has no tiles adjacent to external I/O, those tiles are not used, and the data that would have been passed to them is reallocated to the tiles on the left half of the chip.

As shown in the PCA Kernel Benchmark Report [11], an efficient implementation of the CFAR algorithm can make use of the redundancy in the computation of T:

$$\begin{aligned} T(i,j+1,k) = T(i,j,k) \quad &+ \quad \frac{1}{2N_{cfar}}(|C(i,j+1+G+N_{cfar},k)|^2 \\ &+ \quad |C(i,j-G,k)|^2 \\ &- \quad |C(i,j-G-N_{cfar},k)|^2 \\ &- \quad |C(i,j+G+1,k)|^2). \end{aligned}$$

By using this recursive relationship, the value of T for all N_{rg} range gates can be calculated in $O(N_{rg})$ time. The Raw implementation takes advantage of this by maintaining two buffers. One buffer caches squared values $C(i,j,k)^2$ and is of size N_{rg} words. The other stores the squared sum of a one-sided window and is of size $N_{rg} + N_{cfar}$ words. As values are streamed in for a particular row, these arrays are updated with the necessary values, and $T(i,j,k)$ is calculated accordingly using the two arrays.

3.4.3 Benchmark Results

Figure 32 shows the parameters and the results of running CFAR on a large range of data set sizes. In Figure 32, timing is performed during the entire CFAR algorithm, including when data is streamed in and out. The CFAR algorithm in its current state is able to achieve at most approximately 2.5 operations per cycle for the entire chip. At around 4220 range gates (4220 elements $\times 2$ buffers $\times 4$ bytes/element $\approx$ 32 kB), the two buffers used by the CFAR begin to exceed the Raw's data cache. These buffers are read approximately sequentially, so a drop-off at this point is expected. The performance continues to steadily decline until it reaches approximately 1.77 operations per cycle. There is some fluctuation as the number of range gates increase. However, the fluctuation is relatively small.

The results in Figure 32 were obtained using the HSIO interface. To obtain results on the data sets defined in the kernel benchmark report, we had to use a mixture of HSIO and USB. Data sets 2, 3, and 4 for CFAR consume 82, 22, and 9 MByte of space, respectively. HSIO has a maximum buffer of 8 MByte for sending data, which is only large enough to accommodate CFAR data set 1. By using a special "high-speed input-only" version of the HSIO interface, we were

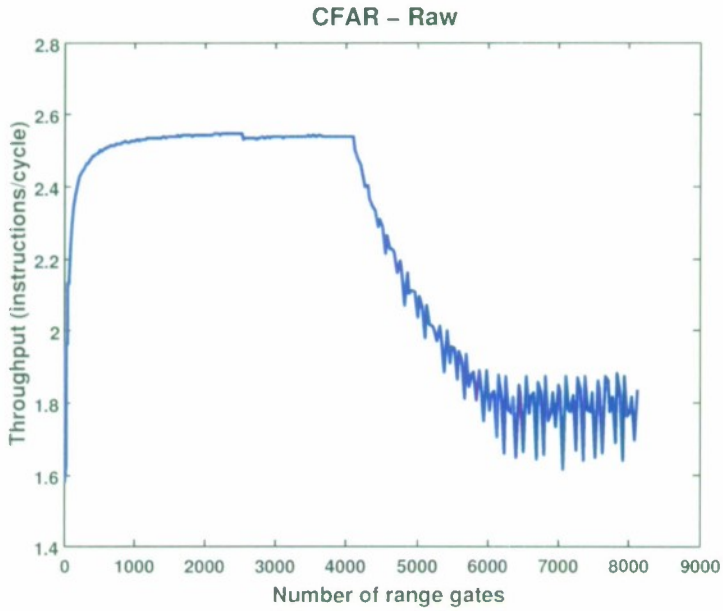

Figure 32. Performance of CFAR on the MIT Raw. $G = 3$, $N_{bm} = 8$, $N_{cfar} = 6$, $N_{dop} = 16$, $20 \leq N_{rg} \leq 8114$, *and* $\mu = 200$.

able to accommodate data sets 1 and 4. Data sets 2 and 3 had to be benchmarked using the USB interface. As the USB interface has a relatively high latency, all of the I/O time for these data sets was subtracted out, making them appear faster than the other data sets.

3.4.4 Other Optimizations

There are a few significant optimizations that can be made to the CFAR. One involves cutting the size of the $C(i, j, k)^2$ array. We observe that the algorithm accesses a squared element only five times, when it is respectively added to the rightmost window, subtracted as part of the rightmost guard cells, compared as a cell-under-test, added as a part of the leftmost window, and subtracted as the leftmost window shifts. Cells that are subtracted as the leftmost window shifts are never accessed again and need not be stored. Thus, instead of having a buffer of size N_{rg}, a lot of memory could be saved by using a rotating buffer of size $2 \times (N_{cfar} + G) + 1$, which is typically far smaller than N_{rg}. As N_{cfar} and G are typically very small numbers, such a buffer would easily fit in cache. This would allow the CFAR to avoid the drop-off shown in Figure 32.

The other major optimization is that the partial window sum buffer is also not necessary. Using the partial sums saves on the number of operations that must be done; the sum of any given window is only done once when it is calculated as the right window, whereas normally it would be implicitly calculated both when it is the left and right window. However, the amount of memory that is required to maintain this buffer makes it relatively expensive, especially when N_{rg} gets large. Thus, using a single value to maintain the current $T(i, j, k)$ could improve performance.

4. Communication Benchmark: Corner Turn

4.1 Algorithm Description

Mathematically, a corner turn or transpose of a matrix can be expressed as

$$B = A^T,$$

where A is a matrix of size $m \times n$ and B is a matrix of size $n \times m$. This operation involves copying elements of A to B in the following way:

$$b_{j,i} = a_{i,j} \text{ where } i = 1, 2, \ldots, m, j = 1, 2, \ldots, n.$$

Thus, the simplest algorithm for implementing the corner turn is to loop over all the elements across the rows and columns as follows:

```
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        B[j][i] = A[i][j];
```

In practice, this simple algorithm will lead to poor performance owing to poor cache utilization. It is more efficient to divide the matrix into blocks and transpose each block individually. The block size depends on the characteristics of the processor, including the cache size.

4.2 Implementation on Raw

In general, a separate corner turn stage should be regarded as something to avoid in programming Raw. If the corner turn is (as is usually the case) occurring between two computational steps, better efficiency can be obtained by overlapping the movement of data with computation at a fine-grained level. However, the corner turn is an interesting benchmark because it shows the performance of the processor in a pure communication operation.

Our approach to the corner turn on Raw is to have each tile be responsible for the transposition of a set of individual blocks. We used as a starting point for the benchmark the C code for a single-processor corner turn developed by Rodric Rabbah of MIT and Jinwoo Suh of USC/ISI East as part of the VersaBench suite (for more information on VersaBench see Rabbah *et al.* [16]). In their code, a block size of 64×64 is used. This allows both the input and the output data associated with a given block to be stored in the processor's 32 kByte cache, because

$$(64 \times 64) \text{ elements } \times (4 \text{ bytes per element }) \times 2 \text{ (input and output)} = 32 \text{ kByte}.$$

Because data movement is the only type of operation involved in the corner turn, the actual input data is irrelevant. Therefore, in contrast to our other benchmarks on Raw, our implementation of the corner turn does not actually use a matrix streamed in from off the board. Instead it takes the size of a matrix as an input. Each tile then allocates space for the blocks of the input and output matrices for which it is responsible, and generates an input matrix of the appropriate size which

it stores in memory. During the timed part of the benchmark, each tile copies and transposes the blocks assigned to it.

For this implementation we access the data purely through the memory network, that is, by reading and writing. We are also using the cache to buffer each block as it is transposed. This implementation is less efficient than an implementation that uses Raw's static network. However, it allows easy scaling of the matrix and the number of tiles involved. The corner turn code is written to allow 1 tile, 4 tiles, or 16 tiles of Raw to be used in the benchmark. In the case of one tile or 4 tiles, the corner turn is performed on the tiles on the east side of Raw, closest to memory.

4.3 Benchmark Results

We began by testing the corner turn for square matrices on a single Raw tile. The achieved throughput is shown in Figure 33. The achieved throughput peaks between 35 and 40 Mbyte/s, but notice that it drops severely for data sizes that are multiples of 256 elements. This performance drop turns out to be dependent on the stride of the source matrix. The corner turn code shown here reads the blocks of the source matrix by columns, and the matrix is stored in row-major order. When the distance between items in the same column is a multiple of 1024 bytes (=256 elements $\times$ 4 bytes per element), the performance drops by more than a factor of 3.

To understand the cause of the performance drop, we must understand that each Raw tile possesses a two-way set-associative cache. Each set consists of $2^9 = 512$ lines of $2^5 = 32$ bytes per line. When two addresses differ by some multiple of $2^{9+5=14} = 16384$, they are mapped to the same two lines in the cache. If we read 64 elements that are 1024 bytes apart, then there will be four lines that map into the same two lines in the cache. By the end of the first column read, the first two lines will have been evicted to make room for the second two lines. Unfortunately, when we read the second column, the same four lines will successively knock each other out of cache again. This competition for the scarce resource of cache lines only increases as the stride grows. We can improve the algorithm by shrinking the row block size when the stride is a multiple of 256. To be precise, we shrink the original block size of 64 by the source-stride divided by 128. In the case where the source stride is 256, for example, this results in a block size of 32 rows.

The results of the improved algorithm running on all 16 tiles of Raw are seen in Figure 34. The bandwidth increases by nearly a factor of 10 when compared to the single tile results. A factor of four comes from the fact that Raw has four "pipes" to memory, one for each row. The remaining increase in performance we attribute to the presence of multiple tiles in each row, each making memory requests, keeping the memory system maximally busy.

There are clearly still performance drops for matrix sizes that are multiples of 1024, that is, when the stride associated with each tile is a multiple of 256. These decreases come from the fact that decreasing the block size does increase the memory traffic. However, the performance drops are clearly not as severe as in Figure 33.

Similar decreases in performance can also be seen in our results on the PowerPC G4 [10], and the cause there was also the cache mapping policy. In fact the drops were in general less severe because the associativity (the number of lines into which any one line may be mapped) of the G4's data cache is larger, but the cause and effect are the same.

We timed the two baseline data sets (corner turn of a 64×5120 matrix and a 768×5120 matrix) on all 16 tiles of the Raw chip. In these cases we map $5120/16 = 320$ columns to each

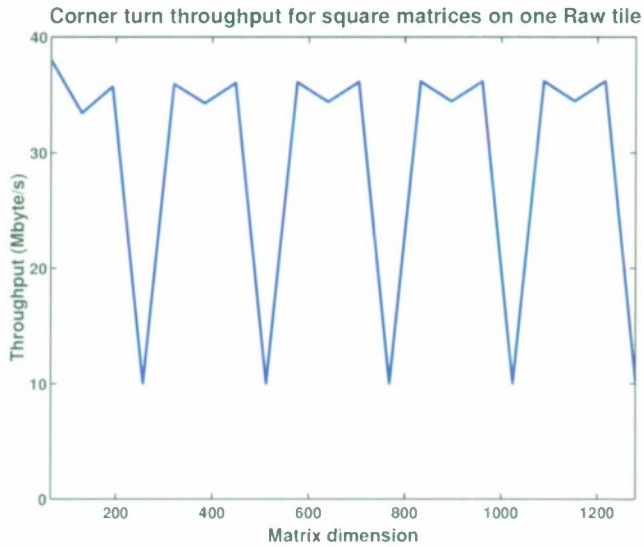

Figure 33. Corner turn results for square matrices on a single tile of Raw.

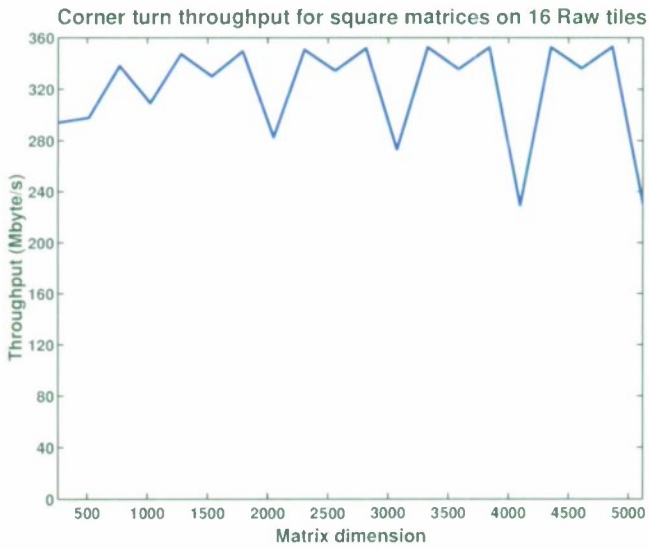

Figure 34. Corner turn results for square matrices on 16 tiles of Raw.

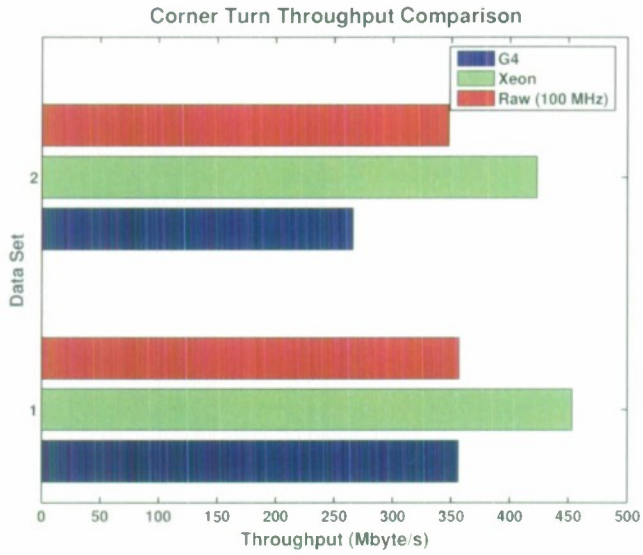

Figure 35. Corner turn throughput for the G4, Xeon, and Raw board.

tile. These are treated as 5 column-blocks of size 64 columns in each block. The results are shown and compared to similar results on the G4 and Xeon in Figure 35. Even though we are using the dynamic network for the corner turn on Raw, the 100 MHz Raw is capable of performing a corner turn at the same rate as the 500 MHz G4. The 2.8 GHz Xeon is only 27% faster in the small case (set 1) and 22% faster in the large case (set 2). This performance is remarkable when it is considered that the G4 and Xeon code was hand-optimized to use SIMD vector instructions (see [10, pp. 21-23]). Furthermore, we note that a static network corner turn in the Raw simulator has been demonstrated to have much better performance than the code presented here (see Section 4.4). The performance of Raw on this benchmark can undoubtedly be attributed to the emphasis on providing network bandwidth in the Raw design.

4.4 Raw Static Network Implementation

Previous work on Raw includes the timing of a fast corner turn of a 1024×1024 matrix using the Raw chip's static network [21] and assuming DRAM connected to all sixteen ports of the chip. It was not used for this benchmark primarily because its assumptions are not compatible with the existing board, which only has DRAM connected to four ports on the east side of the chip. The implementation of this benchmark also has the disadvantage of being harder to scale to different matrix sizes and numbers of tiles. However, it does demonstrate the potential advantage of using the static network and of a different board design.

For the 1024×1024 case, the static network implementation completes in approximately 142,000 cycles, while the memory network implementation takes approximately 2,710,000 cycles on the board (about 20 times as long) and 1,770,000 cycles on the simulator (about 12 times as long). If we presume that placing DRAM on all four sides of the chip speeds up the benchmark by approximately a factor of four, then this benchmark shows the performance of the static network to be about a factor of 3 better than the performance of the dynamic network.

5. Information and Knowledge Processing Benchmarks

The three information and knowledge processing benchmarks defined for PCAs are pattern matching, database operations, and graph optimization via genetic algorithm. The results for these benchmarks are respectively defined in Sections 5.1, 5.2, and 5.3.

5.1 Pattern Matching

5.1.1 Algorithm Description

The pattern matching kernel involves overlaying a test vector a against a library of patterns of length N, and computing the mean square error, MSE, that quantifies the degree to which these two vectors match. Before the two profiles can be overlaid, they may need to be shifted in range to the left or right and the magnitude of the profiles needs to be scaled to match. The optimal shift and gain values can be found through brute force by computing the MSE for each combination of shift and gain values, then taking the minimum MSE. However, by noting that the MSE is a parabolic function of the shift and gain, we can find the optimum shift and gain values at the global minimum by first finding the optimal shift, then finding the optimal gain value.

5.1.2 Mapping

The pattern match kernel can easily be mapped as a threaded kernel due to its underlying data parallelism. Since a MSE will be computed for each pattern template that matches against the test pattern, we can take advantage of that by replicating the pattern match kernel onto all tiles. Data will be distributed accordingly (see Figure 36 below). In detail, the Raw processor is divided into four quadrants or compute units. Each compute tile will retain a copy of the test pattern. The library patterns, which will be evenly distributed across the four compute units, are streamed into the Raw processor from the north and south ports of the four corner tiles. Each unit is responsible for computing the MSE for a quarter of the library patterns. After a tile has finished processing its share, the tiles local minimum MSE will be sent to its corresponding corner tile. The corner tile will then determine the local MSE for the quadrant and send it to tile 0 to find the global minimum. The processing is concluded by sending the index of the closest match out from the west port of tile 0.

5.1.3 Implementation

The pattern match kernel is divided into two stages: range shifting of the test pattern and range and magnitude shifting of the library patterns. Part of the code was parallelized to utilize the four-stage FPU pipeline. In addition, after acknowledging the base 10 power and the logarithmic functions are the bottleneck of the kernel, we have implemented a floating point version of the two functions by unrolling their corresponding Maclaurin series [23].

5.1.4 Results

Figures 37 and 38 show the throughputs and efficiencies in Mflop/s and percent, respectively, obtained for this kernel by varying the pattern length on the 4x4 tile Raw board and the 8x8 tile

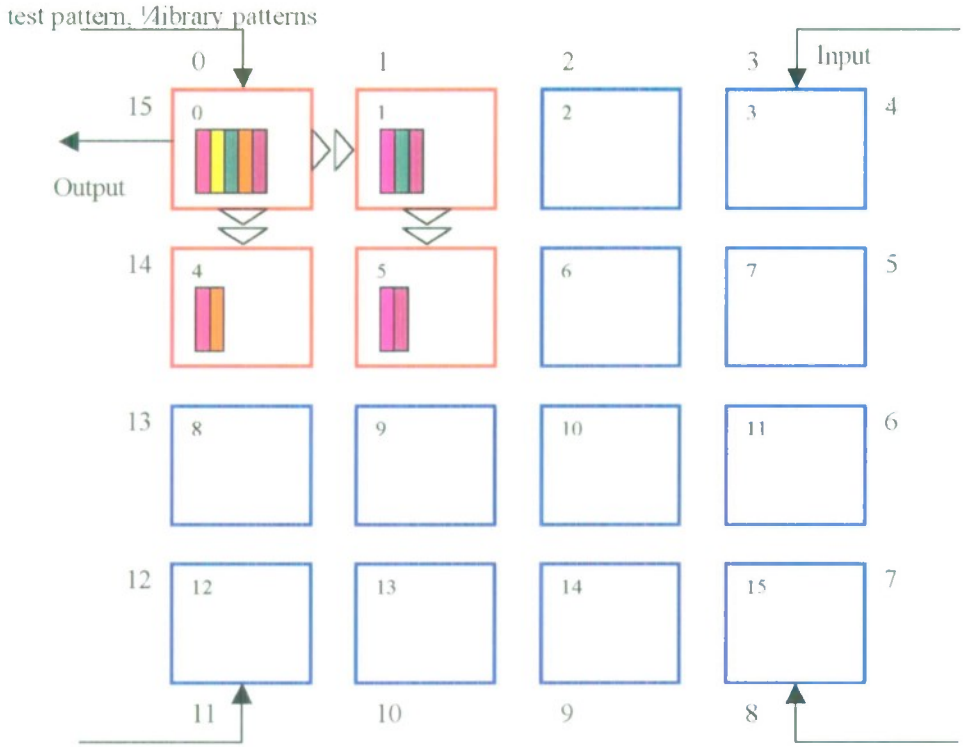

Figure 36. The pattern match kernel mapping. The mapping for one quadrant is shown in detail. The mapping for each of the other quadrants is similar.

simulator. The cycle count is averaged over two trials. The averaged throughputs are 322 Mflop/s and 1240 Mflop/s for the board and simulator results, respectively. When running in serial mode, memory usage for this kernel is relatively large due to the fact that we have to store all the library patterns on one processor. However, when the kernel is distributed across multiple tiles, the amount of memory usage is kept well below the cache boundary. The reason for this effect was that each library pattern was streamed into a tile and discarded once it has finished processing. Essentially, only one library pattern is kept in the cache of a tile at any given time. Thus, the amount of memory required for the kernel is solely dependent of the length of the patterns, and is independent of the number of patterns being matched. The length for which the data will spill over the cache is:

$$32 \text{ kByte}/(4 \text{ byte/pixel})/4 = 2\text{K pixels} \tag{43}$$

Pattern lengths larger than 2K were not benchmarked. For pattern sizes larger than 2K, numerical errors make single-precision implementation of the kernel impractical. Nonetheless, the benchmarked data sizes still do not exhibit behavior consistent with exceeding the size of the cache.

Since data are always available in the cache, the stability of the kernel is expected to be relatively high, and the performance shown in Figures 37 and 38 clearly illustrate this effect. Also supporting this fact is the 3.8×[1] speedup in throughput for quadrupling the number of tiles. The

[1]Computed by dividing the mean throughput of the 8 × 8 simulator results by the mean throughput of the Raw board results.

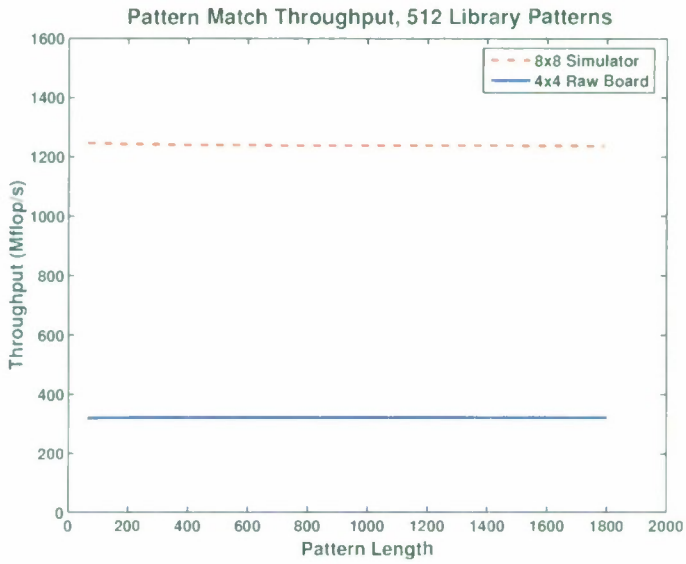

Figure 37. Pattern match throughput on Raw with 512 library patterns.

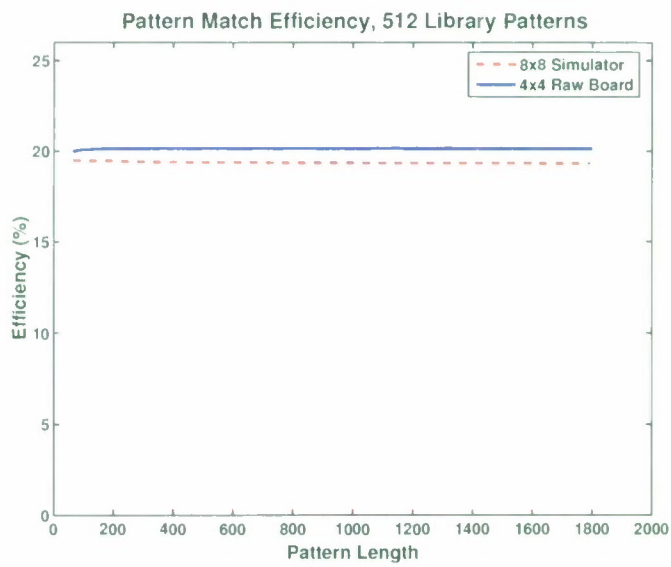

Figure 38. Pattern match efficiency on Raw with 512 library patterns.

speedup is a little below linear, probably due to the fact that the patterns have to travel longer to get to the inner tiles. Overall, the pattern match is a very stable kernel.

5.2 Database Operations

5.2.1 Algorithm Description

The Database kernel benchmark measures the performance of database operations in the context of a tracking application that stores track information in a database [11]. During a discrete time interval, or *cycle*, the tracker application receives target reports from a radar and searches the database for all associated track records. The tracker application may also direct the database to insert new tracks based on target reports that are not associated with any current tracks, and to delete specific tracks.

The database interface therefore receives a stream of instructions from the tracker application in the form of *search*, *insert*, and *delete* operations. Each of these operations are performed on track record indexing structures; the benchmark does not actually alter the contents of any particular track record, nor does it maintain the data associated with the records. Within the database, the following values are stored in each track record index structure: p, a track record pointer value used by the tracker application to locate the track record data; and x and y, coordinate values for the target within an area or grid. The output from the benchmark is a set of record identifiers, or track record pointer values, returned from search operations.

The Database Operations are formatted in the following manner:

$$search\ x_{min}\ x_{max}\ y_{min}\ y_{max},$$

attempts to locate all track records within a specified range of a particular (x, y) coordinate pair;

$$insert\ x\ y,$$

creates a new track record index structure within the database for a target detected at location (x, y); and finally,

$$delete\ x\ y,$$

deletes a specific track record from within the database.

5.2.2 Mapping to Raw

Figure 39 illustrates how the 4×4 Raw chip is used to perform Database kernel benchmark operations. The 16 tiles of the Raw chip are divided into a *Master* tile, an *Insert* tile, and 14 *Search* tiles. The Master tile is responsible for performing off-chip I/O, distributing search, insert, and delete instructions appropriately, and collecting track record pointers returned from search operations. The Insert tile maintains a list of available record pointer values, adding and removing pointer values for insert and delete operations respectively. Finally, Search tiles contain binary trees that hold track record index values. These tiles are responsible for searching the binary trees during search operations and returning valid record pointer values for records matching the search criteria.

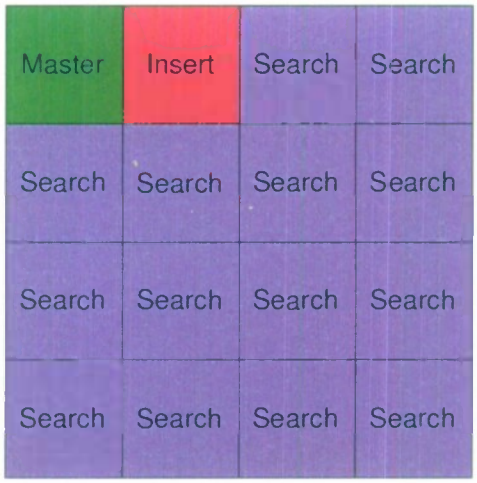

Figure 39. Usage of a 4 × 4 Raw chip for performing Database Operations. The Master tile (tile 0, in green) distributes instructions and collects track record pointer values returned from searches. The Insert tile (tile 1, in red) maintains record pointer values. Finally, the Search tiles (tiles 2-15, in blue) maintain binary trees that contain indexing structures for track records.

5.2.3 Implementation

The following sections give a description of the implementation of the Database kernel benchmark for the Raw processor. All communications described in the following sections are performed using Raw's static network.

Initialization and Timing

The Database benchmark performs all I/O via the northern port of the Master tile (see Appendix A, B). Data set parameter information [11] is passed into the Master tile and distributed to the remaining 15 tiles. All tiles receive the data set parameters and allocate any necessary data structures.

A Matlab instruction generator creates insert instructions to initially populate the database. The Matlab instruction generator also generates the search, insert, and delete instructions for the benchmark. Initialization and benchmark instructions are input into Raw and executed in the same manner; the methods used are described in the following sections. The Master tile begins timing once the initially placed targets, set P, are inserted into the database. The Master tile stops execution after a set number of instructions are performed for a predefined number of cycles. The Master calculates the time taken to perform the benchmark, and outputs the time to the PCA Testbed.

Search Tile Data Structures. Each Search tile holds a binary tree [1] that is used to store track record information. Record information is stored in the binary trees using the target's x coordinate as the key field.

Search Database Operation

Figure 40 shows the data flow patterns for performing search operations on Raw. The search parameters, x_{min}, x_{max}, y_{min}, and y_{max} are distributed from the Master, west→east across the top row, then north→south down columns of the chip. Each Search tile receives the search criteria and parses its binary tree for any track record matching the criteria. All track record pointer values, p, for tracks that match the criteria are returned to the Master north→south, then east→west. Search criteria is met for a track record with (x, y) coordinate values if

$$x_{min} \leq x \leq x_{max} \text{ and } y_{min} \leq y \leq y_{max}.$$

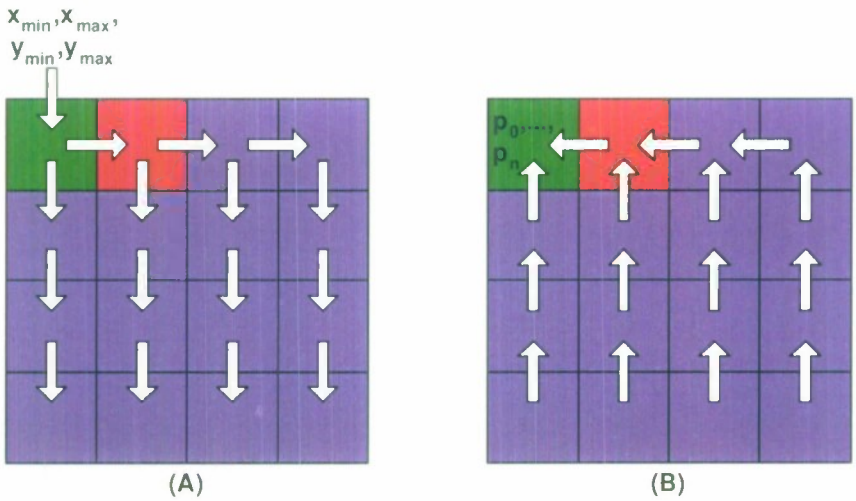

Figure 40. Data flows for performing the search Database Operation. Subdiagram (A) shows the distribution of the search values. Subdiagram (B) shows the data for the return information sent from the Search tiles. Each Search tile will return each track record pointer value for records that match the search criteria.

Insert Database Operation

Upon receipt of an insert instruction, the Insert tile reads the next available track record pointer value from its memory. This value, p, along with the (x, y) coordinate values supplied with the insert instruction are sent to a Search tile that inserts the values into its binary tree. Track record index values are inserted into Search tiles in a round-robin fashion over tiles 15:-1:2. Figure 41 shows example data flows for four insert operation scenarios inserting into tiles 15, 9, 6, and 3, shown in subdiagrams (A), (B), (C), and (D) respectively. In general, for tiles 2 and 3, found in row 0, the values p, x, and y, are passed east to the chosen Search tile. For tiles 4 through 15, or tiles not found in row 0, the value p is passed from the Insert tile west to the Master tile, then values p, x, and y are communicated south and then east to the appropriate Search tile.

Performing the insert operations in this manner allows multiple insert operations to be performed in parallel. Each insert operation requires the allocation of memory as a new node is added

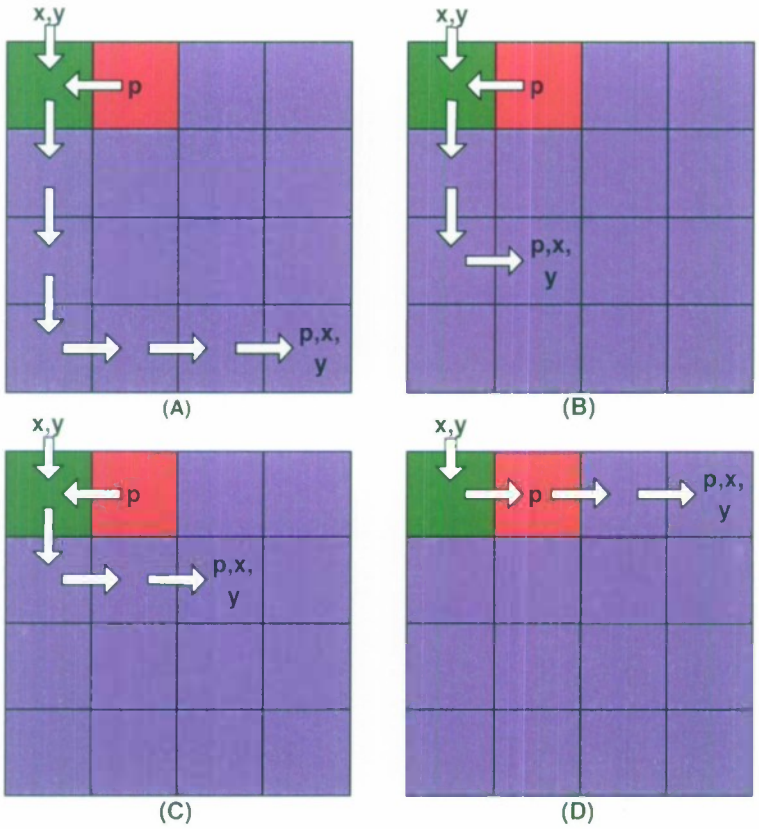

Figure 41. Example data flows are shown for performing insert Database Operations. Track record index values are inserted into Search tiles in a round-robin fashion over tiles 15:-1:2. Examples are shown in subdiagrams (A), (B), (C), and (D) for inserting record index data into search trees on tiles 15, 9, 6, and 3 respectively.

to the binary tree. As one Search tile is performing the allocation, the Master tile will assign insert operations to the remaining Search tiles to be performed in parallel.

Delete Database Operation

Figure 42 shows the data flow required to perform a delete operation. Subdiagram (A) shows the distribution of the instruction and (x, y) coordinate values for the corresponding track record to be deleted. Subdiagram (B) shows the resulting communication as the track record pointer value, p, is returned to the Insert tile to be reinserted into the list of available pointers. When the Search tiles receive the (x, y) coordinate values, each tile searches its binary tree for the corresponding record. If a Search tile locates the record in its binary tree, the pointer value, p, is returned to the Insert tile. Otherwise, the tile sends an invalid pointer value that is ignored by the Insert tile. Invalid values are sent to maintain the structured communication model that is required from using the static network for communication.

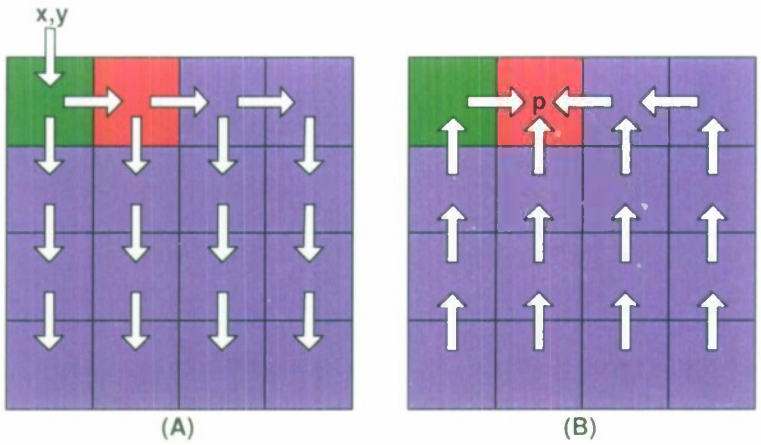

Figure 42. Data flow for performing the delete Database Operation. Subdiagram (A) shows the distribution of the track record values for the record to delete. Subdiagram (B) shows the data flow for the return information sent from the Search tiles. The deleted record's pointer value is returned to the Insert tile.

5.2.4 Benchmark Results

Figures 43 and 44 show the performance results for the Database Operations benchmark run on the 4×4 Raw Handheld board. The results were generated by running 100 cycles, performing 200 search operations, 150 insert operations, and 150 delete operations per cycle, while varying the size of the database. For these data set parameters, Figure 43 shows the throughput in transactions per second, where a transaction is defined as a *search*, *insert*, or *delete* operation. Figure 44 shows the latencies measured for each test.

In Figure 44 an increase in the slope of the latency curve is seen somewhere between database sizes of 10k-20k track records. To examine the causes of the increased latency, we measure the performance of running only search operations. This is done because the search operation is the dominate operation in terms of workload and communication for the database sizes plotted in Figures 43 and 44. Results for running 100 cycles, performing 400 search operations per cycle is plotted in Figure 45. A more dramatic increase in the slope of the latency curve is shown in this figure. For a database size of 16k records, we see that,

$$\frac{16\text{k records} * 28 \text{ Bytes per record}}{14 \text{ Search tiles}} = 32 \text{ kB (Size of binary tree on each Search tile).}$$

As the size of the database approaches 16k records, latencies increase as the size of the binary tree held on each Search tile grows larger than the size of the tile's cache. The red and black dashed lines in Figure 45 respectively show where search operations are performed on target records in the cache of each tile, and where search operations begin to access data outside of the cache on each tile.

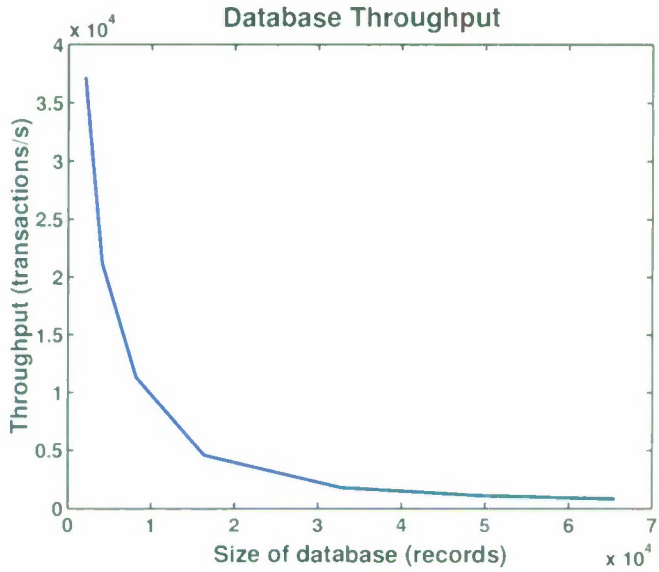

Figure 43. Throughput results are plotted for the Database Operations benchmark run on the 4 × 4 Raw Handheld board. For each test, 100 cycles were run, performing 200 searches, 150 inserts, and 150 deletes per cycle on a grid size of 16 × 16 with a 4 × 4 search area. See [11] for data set parameter details.

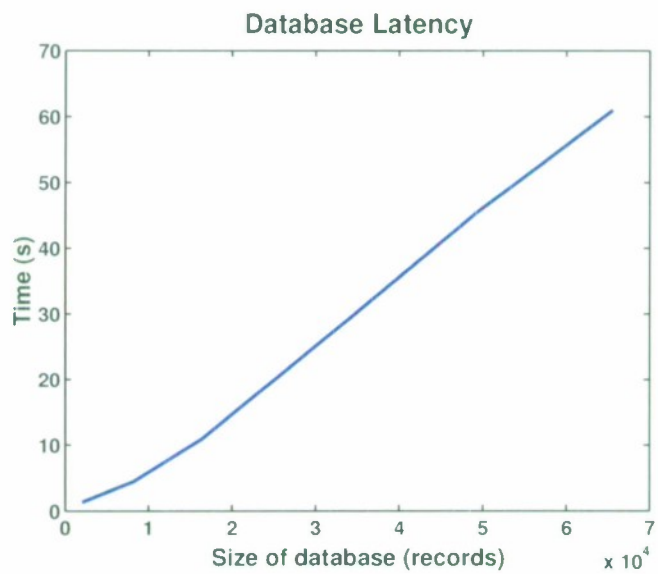

Figure 44. Latency results are plotted for the Database Operations benchmark run on the 4 × 4 Raw Handheld board. See Figure 43 for data set information.

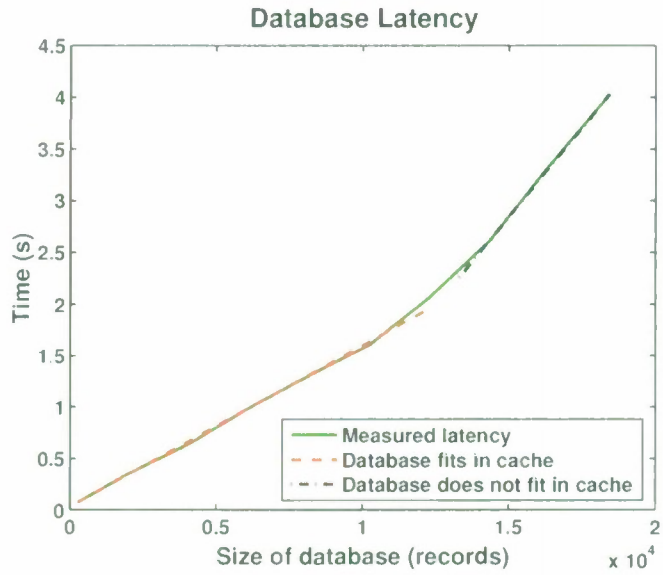

Figure 45. Database latencies are plotted from using the 4×4 *Raw Handheld board for performing search operations only. Latency is plotted in solid green. Red and black dashed lines respectively show where search operations are performed on target records in the cache of each tile, and where search operations begin to access data outside of the cache on each tile. For each test, 100 cycles were run, performing 400 search operations per cycle on a grid size of* 16×16 *with a* 4×4 *search area. See [11] for data set parameter details.*

5.2.5 Further Optimizations

The Database kernel benchmark implementation for Raw could benefit in performance from improvement in the mapping and design of the benchmark, as well as further optimization of the current design. One such mapping change would be to combine the functionalities of the Master and Insert tiles, and perform these responsibilities on a single tile. The current Insert tile could be used as a Search tile providing increased distribution of the target records and reduction of the search operation workload per Search tile.

It may be possible to optimize the insert operation by using the dynamic network for communication[2]. While the current implementation does allow for insert operations to be performed in parallel, there are times where tiles that are in the process of inserting a track record block the transmission of insert commands to other Search tiles. Using the static network requires the explicit programming and control of each switch processor, requiring each Search tile to be finished with its current instruction before proceeding to and passing along the next instruction to its neighbor. The dynamic network could be used to bypass working tiles, better utilizing each Search tile during a block of insert operations.

The performance of the benchmark could also be improved by implementing a memory manager for each of the Search tiles, similar to what was implemented for the G4 implementation of the Database benchmark [10]. A memory manager could reserve a large pool of memory, dol-

[2]At the time of this report we have not evaluated the performance of Raw using the dynamic network in such a context.

ing out pieces upon allocation requests and reassuming responsibility for pieces deallocated by the benchmark. Managing memory in this manner would significantly improve the performance because each cycle performs insert *and* delete operations. For a large enough pool of managed memory, each matching insert and delete operation would remove the necessity of either a *malloc* or *free* system call that requires hundreds of cycles.

Finally, the benchmark performance could also improve through the use of more sophisticated data structures. The binary trees used on the Search tiles could be replaced with red-black trees [1] as was used in the G4 implementation of the benchmark [10]. Using red-black trees would insure that the search tree is *balanced*, potentially reducing tree traversal latencies involved in search, insert, and delete operations.

5.3 Graph Optimization via Genetic Algorithm

5.3.1 Algorithm Description

Genetic algorithms [2, 4, 18] have become a viable solution to strategically perform a global search by means of many local searches. The genetic algorithm used for this kernel benchmark is a fairly simple version [11] that works by first randomly generating an initial population of chromosomes representing a set of possible solutions to an optimization problem. A matrix of scores is also created that determines how "good" a particular code is in a particular gene position within a chromosome.

A typical genetic algorithm usually consists of two tasks: *evaluation* and *selection*. During evaluation, the fitness of a particular chromosome is determined. This score is used in the following selection phase, in which the new generation is created by selecting chromosomes from the current generation. Typically, chromosomes are randomly selected with probability proportional to a chromosome's fitness score. Chromosomes are then potentially subject to *mutation*, in which individual gene values change to a randomly generated code, and *crossover*, where a pair of chromosomes exchanges genes with one another.

A parallel version of the genetic algorithm [3] adds a *migration* stage between evaluation and selection. During migration, certain chromosomes residing locally on each tile are copied to neighboring tiles.

5.3.2 Implementation Features

In the Raw version of the genetic algorithm, each tile runs an independent instance of the genetic algorithm with its own local pool of chromosomes. The only time that tiles actually communicate is during the migration phase. The algorithm proceeds in five steps listed below.

1. The chromosomes in the current generation are evaluated using the scoring matrix. The *elite chromosome*, or chromosome with the highest score, is noted, and the scores are stored for later use.

2. Two copies of the elite chromosome are maintained. One copy is left untouched between generations, and the other is subjected to mutation.

3. The elite chromosome from a particular tile will emigrate to each of the tile's neighbors. Thus, each tile will receive between 2-4 chromosomes from its neighbors. Note that tiles

with less neighbors (such as the edge tiles) will have less immigrants and thus will have more open spots during the selection phase.

4. The remaining spots in the new generation are filled in by randomly selecting pairs of chromosomes from the old generation. The probability that a chromosome will be selected is equal to its fitness score divided by the total fitness of the entire population. Each gene in a selected chromosome is subjected to mutation with a given probability. These pairs are also potentially subjected to crossover, which involves randomly choosing a site along the length of the two chromosomes and exchanging all the genes of the two chromosomes past this point. Note that selection, mutation, and crossover are done simultaneously in order to minimize memory operations.

5. If there remains a leftover spot, it is filled with another copy of the elite chromosome, which is subjected to mutation. This occurs when the local population size minus the number of neighboring tiles is odd and thus cannot be filled by pairs of selected chromosomes.

The parameters of the kernel, including population size, probability of mutation/crossover, and fitness scoring, remain the same for any particular run of the kernel.

The number of chromosomes locally on each tile is calculated as follows:

$$\text{local number of chromosomes} = \max\left\{6, \left\lceil \frac{\text{global number of chromosomes}}{\text{total number of tiles}} \right\rceil\right\}. \tag{44}$$

The global number of chromosomes is specified by the data set, and the total number of tiles is dependent on the Raw chip (16 in the present 4×4 Raw chip). The effect of the max function in equation (44) is to guarantee that each pool can at minimum hold the two elite chromosomes and up to four immigrant chromosomes. The typical net effect, however, is that the global pool is divided among all the tiles.

Each tile allocates two pools with sizes equal to the product of the local number of chromosomes and number of genes per chromosome; these pools hold the current and next generation of chromosomes. In addition, each tile also maintains a copy of the scoring matrix, which maps a particular code in a particular gene position to a score, and uses it to calculate the fitness of a particular chromosome. This matrix, along with all the other parameters, is propagated from tile 0, which is responsible for all the external I/O, to all the other tiles. Finally, each tile maintains a scorecard which represents the actual scores of each chromosome in the current generation; this is used during phase 4.

After a specified number of generations, the tiles propagate their elite chromosome back to the northwest tile. This is done by performing a process similar to the one specified above, except maintaining only one copy of the elite chromosome in step 2 and not performing steps 4 and 5.

The random number generator used in this implementation is the same version of the VSIPL random number generator [17] used in the G4 version of the kernel [10].

5.3.3 Benchmark Results

Timing is performed on each generation as well as the final propagation. The final propagation's time is amortized among all the generations, and the mean time over all generations is reported.

As in the G4 version of the kernel [10], the genetic algorithm relies on a random number generator to control the behavior of certain processes, such as crossover, selection, and mutation. The scoring process's memory accesses also depends on what a particular gene's value is. Thus the memory access pattern during one generation of the genetic algorithm is not completely sequential. However, after scoring has occurred and a particular pair of chromosomes are selected, the values are read and the crossovers performed in a pairwise sequential manner. That is, the accesses alternate between the two patterns, but each pattern is read sequentially. Also, the memory accesses to the new generation's pool are completely sequential.

As a result, the effects of memory subsystem can be seen in the genetic algorithm's performance, as shown in Figure 46. Performance for the entire chip peaks at slightly over 500 Mflop/s, an efficiency of about 30%, until the the size of the two pools plus the scoring matrix and scorecard exceeds the 32 kByte level-1 cache boundary. When the genetic algorithm exceeds cache capacity, performance begins to drop. As the memory usage far exceeds cache, the genetic algorithm's performance asymptotically approaches about 178 Mflop/s, or about an 11% efficiency.

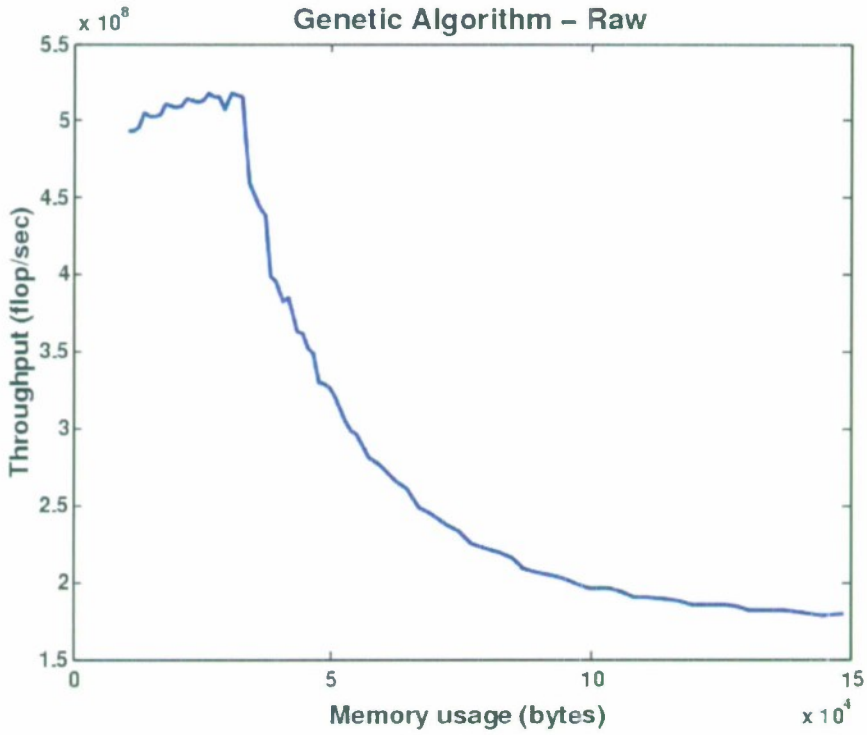

Figure 46. Throughput of the genetic algorithm benchmark on the MIT Raw. The number of genes/population size varies from 10 genes/6 chromosomes per tile (~10.25 kBytes) to 120 genes/30 chromosomes per tile (~145.43 kBytes). The number of codes = 250, number of generations = 5000, $P(mutation) = 0.05$, and $P(crossover) = 0.60$.

5.3.4 Further Optimizations

There are several optimizations that could have been done to achieve better performance on this kernel. One simple optimization involves not re-evaluating chromosomes in later generations whose score is already known by some tile. For example, the unmodified elite chromosome kept between generations is currently re-evaluated every generation. In addition, tiles migrating elite chromosomes could also send the score of the corresponding chromosomes, which increases network traffic by one value per chromosome but saves the destination tile from traversing through the genes of a chromosome and performing lookups into a score matrix. A further optimization could make use of prefetching on the new generation pool to minimize the number of writes that miss cache. This could mitigate the drops in performance at the cache boundaries. Similarly, it is possible to reduce memory accesses further by ordering the memory reads necessary during the random selection of chromosomes. This could be achieved by generating and sorting all the random numbers before performing the reads.

6. Raw Kernel Benchmark Observations

The previous chapters discuss detailed results for individual kernels. This chapter contains general observations about Raw based on the observed results. We begin with a comparison of the Raw simulator to the Raw board in Section 6.1. Section 6.2 compares development effort on Raw to conventional processors. Section 6.3 summarizes throughput, stability, and throughput per unit power results for the baseline kernel data sets on Raw, and compares the Raw results to previous results for the G4 and Xeon.

6.1 Board-Simulator Comparison

Prior to delivery of the Raw board in April 2004, MIT/LL used the cycle accurate simulator for Raw as a means of development and testing of the kernel benchmarks, as well as to obtain initial performance estimates. However, experimentation has shown that the Raw board and the Raw simulator yield considerably different performance results. In this section, we discuss these differences and their causes.

Figure 47 shows performance plots for the QR decomposition kernel benchmark run on the Raw board and the Raw cycle-accurate simulator [7]. A performance drop-off is seen when M (for an $M \times M$ input matrix A) is equal to 64, similar to the cache effects seen in results for the G4 [10]. At $M = 64$, we see that,

$$64 \text{ rows } * 64 \text{ columns } * 8 \text{ Bytes per complex element} = 32 \text{ kB (Size of matrix } Q),$$

and

$$64 \text{ rows } * 64 \text{ columns } * 8 \text{ Bytes per complex element} = 32 \text{ kB (Size of matrix } R).$$

Because the data is divided between two storage tiles, when $M = 64$ each storage tile holds 32kB of data. A performance drop-off is seen at this point because, for larger values of M, Q and R will no longer fit into the 32kB data cache contained in each of the Raw tiles.

Inferring from the cache effects seen in Figure 47 as well as an investigation into the Raw simulator source code, we hypothesize that the memory model used by the simulator does not accurately match that of the board, consequently producing the differences seen between the Raw Board and the cycle-accurate simulator due to off-chip DRAM access delays. One possible cause of the delays is that parameters used by the simulator do not match the specifications of the board (e.g. Raw clock speed, DRAM clock speed). Whatever the cause, the simulator uses inappropriate estimates of DRAM write and read penalties for cache misses. Experimentally adjusting the values $dramReadLatency$ and $dramWriteLatency$ within the simulator code[1] produced performance numbers close to those attained from using the Board. Comparisons are shown in Figure 48. For this plot, the values $dramReadLatency$ and $dramWriteLatency$, which are usually set to the values 6 and 1 respectively, were each set to 16. These results suggest that a near-accurate estimate of the Raw Board's performance can be obtained by adding constant delays to DRAM access penalties within the simulator. It should be noted that the values chosen for the variables

[1]The simulator is written in a language known as "bC": the actual file that was changed was in the subdirectory btl/dev/dram.bc.

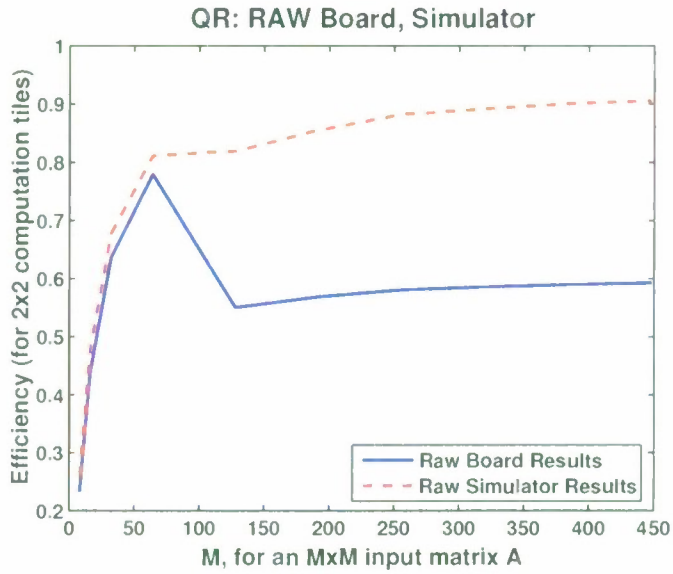

Figure 47. QR Decomposition results using the Raw Board and the Raw cycle-accurate simulator.

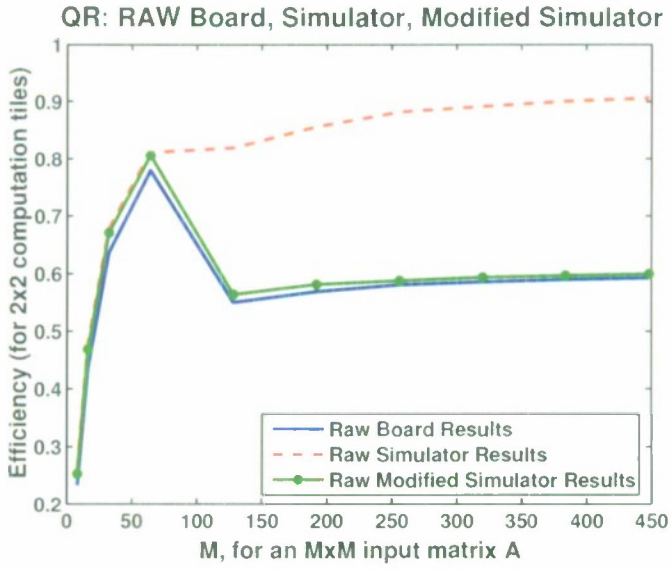

Figure 48. QR Decomposition results using the Raw Board, cycle-accurate simulator, and the cycle-accurate simulator with modified DRAM read and write penalties.

dramReadLatency and *dramWriteLatency* for Figure 48 were found in an experimental fashion. Other kernels with memory usage patterns different from the QR could expose other possible DRAM delays requiring different modifications to the simulator. Therefore, further knowledge and research into the simulator code and Board would be necessary to create a completely accurate simulation model. For this effort, the majority of benchmarking work was performed on the board, making a completely accurate model unnecessary. However, such a model could be a useful tool for examining current limitations in the board design, and for determining requirements for possible future chips containing a larger number of tiles. Also, it may show the need for algorithms that use better data and computation blocking to minimize cache misses.

6.2 Development Effort

In this section, we make comparisons of the development effort required to achieve high performance on Raw versus the effort required on the PowerPC G4. Briefly, achieving performance on the G4 requires tuning the code to the memory hierarchy and adding Altivec instructions. Achieving performance on Raw requires specializing code for each tile and writing code to communicate among tiles. Tile communication code is assembly-language code and so is very tedious to write.

We quantify the amount of effort by comparing the lines of code required. This is in some sense an unfair comparison, as lines of code do not directly translate to development effort. However, it is the best metric we have in the absence of actual time-to-develop statistics.

To begin with, we consider the implementation of the FIR filter on Raw. We have four implementations available to compare. The first is straightforward, portable ANSI C code for the frequency-domain FIR filter, available as part of the HPEC Challenge benchmark suite [6]. This code is a radix-4 FFT-based implementation. We compare this code to the streaming frequency-domain implementation described in Section 3.1. We also compare with an optimized single-tile frequency-domain FIR filter bank developed by Jinwoo Suh of USC/ISI. This implementation is capable of performing up to 16 FIR filter operations simultaneously, one per tile.

Table 2.

Lines of code for four FIR filter implementations.

Name	Developer	Language	SLOC
HPEC challenge benchmark	MIT/LL	ANSI C	400
Single-tile	USC/ISI	C,Assembly	1525
Stream FFT	MIT/LL	C,Assembly,Network	3450

We can see the large jump in code size required to implement the single-tile optimized version versus the straightforward C code version. However, adding the code to make a multi-tile version adds even more lines of code, more than doubling the code required for the single-tile version (which was already almost four times as large as the C version).

As another data point, we consider the implementation of the corner turn kernel. The AltiVec implementation of this kernel was previously described [10]. The Raw static network implementation referred to in Section 4.4 can serve as a point of comparison, since these perform the same operation. The G4 corner turn code including the AltiVec assembly language routines comprises about 260 lines of code. By comparison, the Raw static network version comprises 4750 lines of

code. Of these, about 4000 lines of code are assembly language code used to route the data. The Raw code is nearly a factor of 20 larger. This roughly corresponds to the fact that Raw can be considered to have a factor of 16 times as many instruction streams as the PowerPC.

The AltiVec corner turn, though it imposes some limitations on data size, can be used with a range of matrix sizes. About 3000 lines of the Raw static code are actually generated by a Perl script and are matched to the specific number of tiles on Raw and the size of the matrix. Automatic generation of this code is clearly very desirable.

Certainly, the ideal tools for tiled architectures are not yet well-defined. It would be unrealistic to expect mature versions of such tools to emerge fully formed from an academic project like Raw. However, by pointing out the amount of effort required to achieve high performance on these architectures, we hope to motivate the expenditure of more time and effort on such tools.

6.3 Baseline Results and Platform Comparison

In this section, we summarize the performance of Raw on the PCA kernel benchmarks, and compare with the performance of the Xeon and the G4. The parameters of the three chips are summarized in Table 3. In particular, it is important to point out that the Xeon is implemented in newer technology than the other two chips.

Table 3.

Processor Parameters

Parameter	Raw	G4	Xeon	Units
Clock speed	100	500	2800	MHz
Peak throughput	1.6	4	11.2	Gflop/s
On-chip level-1 cache	16 × 32	32	8	kbyte
On-chip level-2 cache	–	–	512	kbyte
Off-chip bandwidth	160	1	4.3	Gbyte/s
Typical chip power	5	5.3	74	W
Technology generation	0.18	0.18	0.13	micron
Chip die size	330	52	146	mm^2
Transistors	118	10.5	108	million

With one exception, the performance numbers we cite in this section are for the implementations described in this report. The exception is the FIR filter, where we made use of an optimized single-tile implementation provided by Jinwoo Suh of USC/ISI East. When multiple filters are available to be distributed to each tile, this implementation gives good performance.

The original kernel benchmark description [11] gave several baseline parameter sets, based on real application parameters, for each of the kernel benchmarks. The throughput for the baseline data sets for each kernel is summarized in Figure 49. Throughput for the corner turn kernel benchmark is given in Mbyte/s, while throughput for the database kernel benchmark is given in transactions per second. Throughput for all the other kernels is in floating-point operations per second.

We compare the absolute performance of Raw to that of the G4 and Xeon in Figure 50. In that figure, we include both actual throughput measured on the Raw board at 100 MHz, and scaled

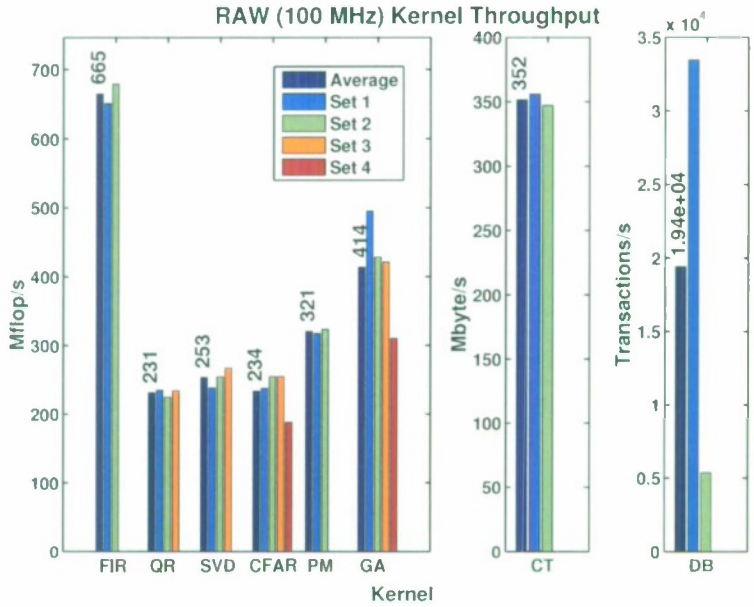

Figure 49. Achieved throughput of kernels on defined data sets on the Raw board.

performance numbers for an improved board. For the improved board, we assume a 425 MHz clock frequency (MIT's best estimate of the maximum frequency of the Raw chip) and linear scaling. The results are very impressive. The 100 MHz Raw is competitive with the G4 despite a 5× difference in clock frequency and a 2.5× difference in peak performance. The 425 MHz Raw is competitive with the Xeon operating despite being from an earlier technology generation and operating at a much slower clock frequency. In addition, Raw's performance averages are more consistent across kernels than either of the conventional architectures.

In Figure 51 we compare the performance per unit watt for the Raw, G4, and Xeon. As discussed in the original kernel benchmark specification, we consider only the power of the processor and not the power associated with other components necessary to put together a system [11]. This is done because the number of type of such components vary with the system purpose and it is reasonable to assume that the components might be similar for any processor employed for a specific purpose. Performing the calculations using only the processor power exaggerates the importance of the processor but also highlights differences among architectures.

Raw compares very well to the other architectures in Figure 51, despite the fact that it is an academic design that is not power-optimized. The Xeon does not perform well by this metric, as it is designed for use in server environments rather than embedded systems. But Raw's performance is very close to that of the embedded G4 system, and on some kernels it outperforms the G4 in terms of throughput per watt.

Based on the throughput measurements, stability for each kernel can be calculated. The data set stability, that is, the stability over all data sets for a particular kernel, is shown in Figure 52 for each kernel, for the Raw, G4, and Xeon. On a per-kernel basis, Raw's stability is similar to that of the other platforms. It shows better stability for the SVD and genetic algorithm, and worse stability for CFAR detection.

Also shown in Figure 52 is the stability over all the floating-point kernels, that is, the minimum

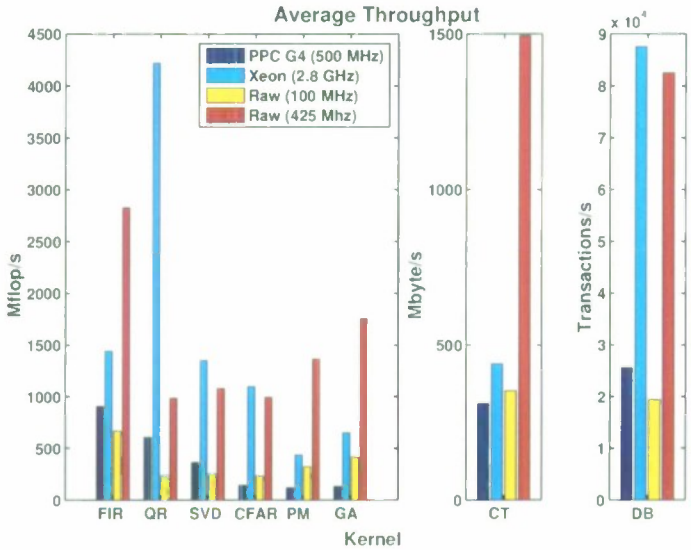

Figure 50. Average throughput for each kernel on the Raw board and on the Raw board scaled to 425 MHz, compared to the G4 and Xeon.

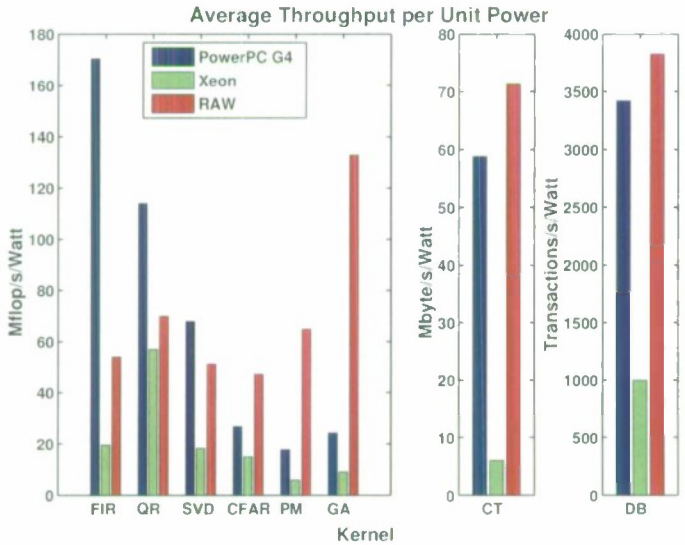

Figure 51. Average throughput per unit power for each kernel on the Raw board compared to the G4 and Xeon.

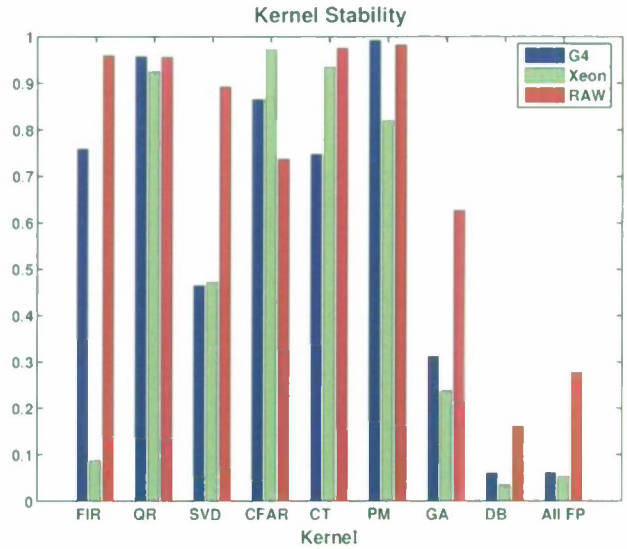

Figure 52. Achieved data set stability for kernels on the Raw board compared with the G4 and Xeon.

achieved throughput for the floating-point kernels divided by the maximum achieved throughput for those same kernels (FIR, SVD, CFAR, pattern match, and genetic algorithm). Notice that this number is much lower for each of the platforms than the individual stability for each kernel. However, Raw has a higher stability over all the floating-point kernels than either the G4 or the Xeon. The ratio between the best and worst performance on Raw is about 4 to 1, whereas on conventional architectures it is about 20 to 1. One way to interpret this result is to say that Raw's tiled architecture can be more consistently used to achieve high performance across kernels than the SIMD instructions present on the G4 and Xeon.

The actual stability numbers are given in Table 4, where they are compared to the numbers for the G4 and Xeon. Another interesting metric to compute that is an indicator of overall stability is the geometric average of the data set stabilities (the seventh root of the product of the stability scores shown in Table 4). This metric gives an indicator of the machine's stability over all kernel types. For the Raw with the baseline kernels and data sets, the geometric average of the data set stabilities is 0.700. Corresponding numbers for the G4 and Xeon are respectively 0.677 and 0.490. This also shows an advantage for Raw, though not as large as that shown by the floating-point kernel stability.

In summary, it is worth observing that a 425 MHz version of Raw shown in Figure 49 is expected to do well both in terms of performance and performance per watt. The average performance and average performance per watt for all three chips is shown in Table 5. While the G4 delivers the best average performance per watt, and the Xeon delivers the best overall performance, Raw is very close to the best in both categories. Combined with the knowledge that Raw also gives more consistent performance than the other two architectures, this is a strong endorsement of Raw's design and capability.

Table 4.

Kernel stability numbers for the Raw, G4, and Xeon.

Kernel Name	Stability		
	Raw	G4	Xeon
FIR	0.959	0.759	0.878
QR	0.934	0.956	0.924
SVD	0.892	0.464	0.472
CFAR	0.737	0.864	0.971
CT	0.974	0.747	0.933
PM	0.981	0.958	0.819
GA	0.626	0.311	0.238
DB	0.161	0.058	0.040
All floating-point kernels (FIR, SVD, CFAR, PM, GA)	0.277	0.062	0.053

Table 5.

Average Performance and Performance Per Watt for the Raw, G4, and Xeon

Chip Name	Clock Rate (MHz)	Average Throughput (Mflop/s)	Average Throughput Per Watt (Mflop/s/Watt)
Raw	425	1.5	71
G4	500	0.37	71
Xeon	2800	1.53	21

7. Conclusions

We have presented a set of kernel benchmark measurements on the MIT Raw processor, an early Polymorphous Computing Architecture. The results show that despite being an academic design, Raw is a scalable, flexible architecture capable of delivering consistent high performance and performance per watt. In contrast, our two conventional architectures show less consistent performance and either high performance or high performance per watt but not both.

Our major area of concern for Raw is the programmability of the architecture. Optimizing code for Raw is a very labor-intensive process. It takes approximately an order of magnitude more code to program Raw for high performance than to program at an equivalent level on the PowerPC. Tools to automate the development of high-performance code are sorely needed.

As more tiled architectures appear in industry and academia, high-performance programming of these architectures will continue to be an issue. In our evaluation of Raw, we have shown the potential of these architectures. We have also demonstrated techniques for high-performance programming that are scalable to future, larger tiled arrays. We believe these will be a good foundation for further work on these architectures.

APPENDIX A
Testbed hardware design

A.1 Introduction

This appendix describes the design of the PCA high speed I/O (HSIO) system. This system is designed specifically for the MIT Raw architecture with hopes that it can be easily extended to the other architectures being explored by the PCA program. The testbed consists of three main parts: the hardware, the firmware, and the software. Each of these parts are discussed in a section below. Following this introduction, Section A.2 describes the hardware setup including a specification of each hardware component and of the data flow on the testbed. Next, the software executing on the host is described in Section A.3. Finally, Section A.4 describes the firmware designs of the customizable FPGA components of the testbed.

A.2 Hardware Setup

The hardware portion of the testbed is made up of six major pieces: a computer to host the WildStar II/PCA, a WildStar II/PCI board, two WildStar Data Port-Euro daughter cards, the Raw Handheld board, and a computer to host the Raw Handheld board. The list below outlines the functionality of each of these pieces and their interfaces with other components. These relationships are depicted graphically in Figure 53.

1. **WildStar Host Computer** — Connects to the WildStar II board.
   - Interface to WildStar II: PCI 32 bits @ 33 MHz
   - Processor: Pentium 4 @ 2.53 GHz
   - Operating System: Redhat Linux (Custom Kernel 2.4.20 modified for WildStar PCI driver)
   - Compiler: GCC 3.2.2
2. **WildStar II PCI** — Connects to the host and to the WildStar Data Port daughter cards.
   - Interface to host: PCI 32 bits @ 33 MHz
   - Interface to Data Port: 2 × 153 bit MICTOR connector
   - Processors: 2 × Virtex II 6000-5 FPGAs for custom processing
   - Memory: 12 Mbytes DDR II SRAM (6 ports/PE)

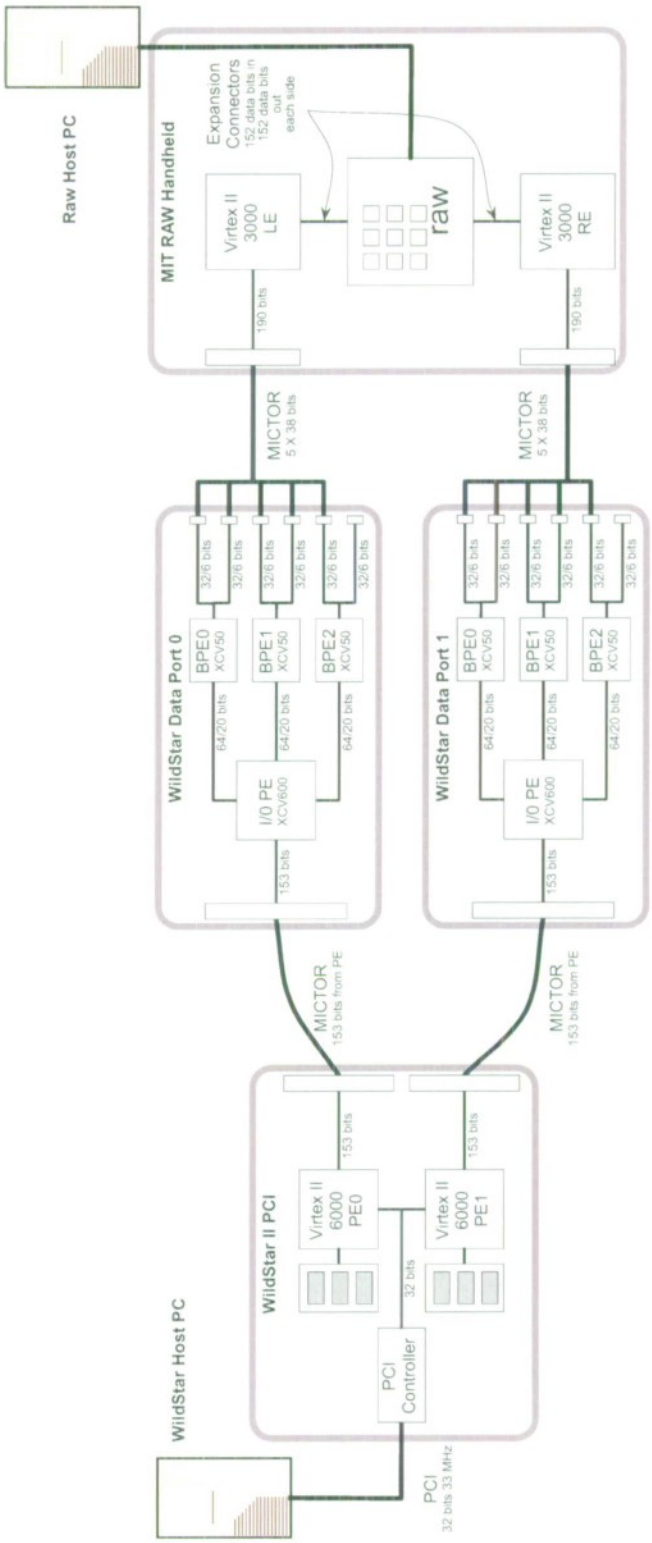

Figure 53. PCA testbed system components.

3. **WildStar Data Port × 2** — Connect to the WildStar II board and to the Raw Handheld board.
   - Interface to WildStar II: 153 bit MICTOR connector to expansion FPGA
   - Interface to Raw Handheld: 5 × 38-bit MICTOR ribbon connectors
   - Processors: 1 × Virtex 600E-7 and 3 × Virtex 100E-7 FPGAs
4. **Raw Handheld** — Connects to both WildStar Data Port cards.
   - Interface to each Data Port: 190 bit MICTOR connector
   - Interface to Raw: 64 bits in + 64 bits out + 24 control bits
   - Processors: 2 × Virtex II 3000 expansion FPGAs + Raw 16-tile processor @ 100 MHz
5. **Raw Host Computer** — Connects to the Raw Handheld board.
   - Interface to Raw Handheld board: USB 2.0
   - Processor: Pentium 4 @ 2.80 GHz
   - Operating System: Redhat Linux (Custom Kernel 2.4.26 modified for USB 2.0 driver)
   - Compiler: GCC 3.2.2

The HSIO can be used to both send and receive data. The information flows through the HSIO touching all of the hardware components described above, except for the Raw Host computer. This path is explained below:

1. The data are generated on the host.
2. The data are transmitted to the WildStar II and placed in the board's local memory through the two processing elements (PEs). The PEs are the start and end points for the data paths in the I/O system.
3. The host gives the WildStar II a signal beginning data transmission from each PE to its respective Data Port daughter card.
4. The daughter cards pass the data to the expansion FPGAs on the Raw Handheld board.
5. The expansion FPGAs send the data to the Raw and forward the results back to the Data Port daughter cards.
6. The daughter cards, in turn, pass the results back the WildStar PEs.
7. The WildStar PEs store the results and wait for the signal from the host that the benchmark has completed.
8. The host software gathers the results from the WildStar, verifies their correctness, and generates performance statistics.

A.3 Software Design

Software is executed on both the host and the Raw processor. This document discusses the control for the High Speed I/O System. A description of the overall testbed software can be found in an accompanying appendix.

Host Software — Format data, download data, send control signals, upload results

- Language: C/C++
- API: Use the Annapolis supplied programming interface and driver for writing data to and reading data from the board. The data transfer can be performed before and after the timed test.
- Task 1: Format test data.
- Task 2: Download data to the board.
- Task 3: Send "stream" signal to the WildStar II.
- Task 4: Await "done" signal from the Testbed software.
- Task 5: Upload results.
- Task 6: Format results and forward them to the Testbed software.

A.4 Firmware Design

The firmware design refers to the design of the FPGA modules that are downloaded to the WildStar II PEs, the Data Port PEs, and the Raw Expansion FPGAs. The firmware design combines modules developed at MIT/LL, modules provided by Annapolis Micro Systems (AMS), and modules provided by the MIT Raw group. In the figures below the off-the-shelf modules are gray and the custom modules are white. In general, the FPGAs are configured to simply pass the data from the memory on the WildStar II to the Raw and record the results that the Raw generates. The FPGAs do not aid in the computation of the benchmark. Each of the firmware designs are described in more detail below:

1. **WildStar II PE** — Interface to the memory controller and send data to the Data Port daughter card. See Figure 54.
    - PCI Interface @ 33 MHz (Annapolis): Manage the communication with the host.
    - Memory Interface @ 120 MHz (Annapolis): Manage the memory transfers between the on board RAM and Custom Design 1. Two memories (one for send and one for receive) are used for each port on the Raw. This leaves two of the six memories unused.
    - Custom Design 1 @ 33/120 MHz: Stream data from the memory controller to the external interface. Make a clock transition from 33 MHz (PCI) to 120 MHz.

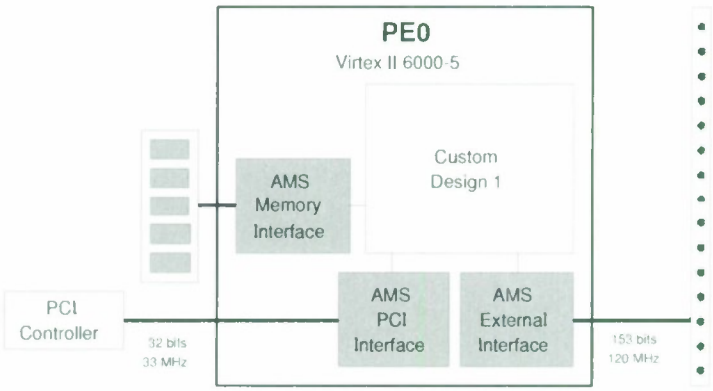

Figure 54. WildStar II Processing Element (PE) 0. PE 1 has an identical design.

- External Interface @ 120 MHz (Annapolis): Buffer and send data through MICTOR connector to the daughter card.

2. **Data Port I/O PE** — Pass the data along. See Figure 55.

   - External Interface @ 120 MHz (Annapolis): Send to and receive from the WildStar II PE via the external MICTOR connector.
   - Custom Design 2 @ 120 MHz: Perform any formatting necessary. Send data to and receive data from the External Interface to the Raw Handheld.
   - External Interface @ 120 MHz (Annapolis): Send to and receive from the Raw Handheld via the small MICTOR ribbon connectors.

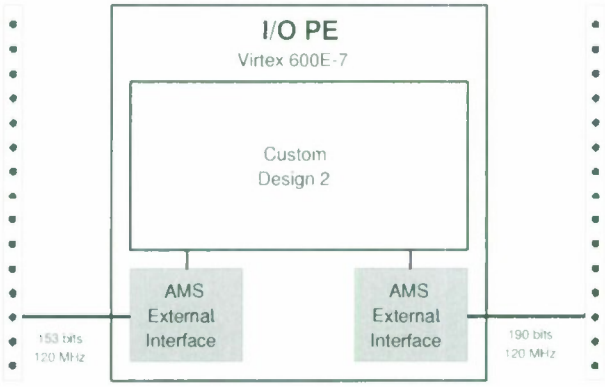

Figure 55. WildStar Data Port Daughter Card I/O Processing Element (PE).

3. **Raw Handheld Left and Right Expansion FPGAs** — Interface the High Speed I/O System to the Raw Processor. The data will only be transmitted via four of the the Raw static network ports. The I/O system will stream and retrieve data from both expansion FPGAs on the Raw Handheld board giving an effective data input and output rate of 8 words per cycle at 100 MHz (3.2 GBytes/s), with each expansion FPGA working at half that rate. See Figure 56.

- Custom External Interface @ 120 MHz: Send to and receive from the Data Port daughter card via the small MICTOR ribbon connectors.
- Custom Design 3 @ 120 MHz: Transmit to and receive data from the Speed Gasket module.
- Speed Gasket @ 100 MHz (Raw Team): Interface with the Raw static network. Multiplex and decode the values presented according to the control signals from the custom design.

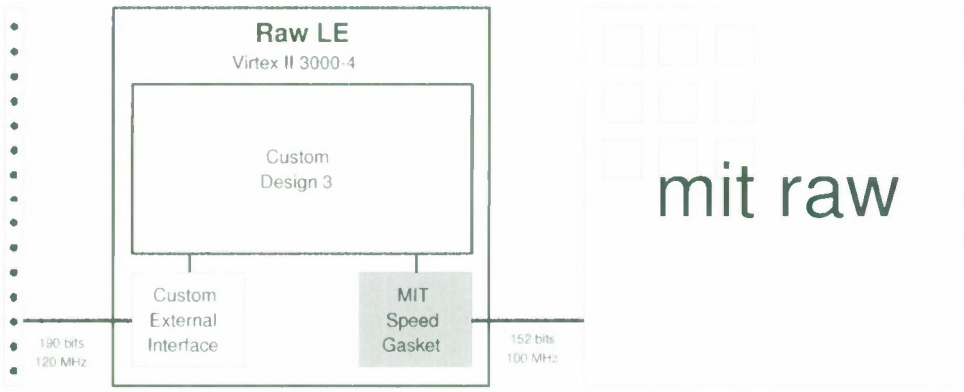

Figure 56. Raw Handheld Left Expansion FPGA. The Right Expansion FPGA has an identical design.

A.5 Summary

The PCA testbed HSIO interface allows input data to be sent in a streaming mode from the host computer to Raw and results to be sent back in the same fashion. It uses a combination of custom firmware on both the Annapolis WildStar II board and on the MIT Raw board to achieve this goal. The system described here has been implemented and was used to produce the kernel results given in this report.

APPENDIX B
Testbed software design

B.1 Introduction

The goal of the Polymorphous Computing Architecture (PCA) project is to evaluate new "morphing" computer architectures. Through the use of a variety of benchmarks and interfacing hardware, testing can be done to determine the benefits of these new processors. In order to facilitate the enormous task at hand, however, there is a need for a standard software interface to allow testing to be done easily on a variety of platforms.

This appendix describes the design and implementation of software used to interface a host computer with an architecture under test. The interface allows a program running on the host computer to communicate with a specific platform under test, sending it code and data and retrieving the results of an execution. The software is based on architecture modules that share similar interfaces so that they can be swapped in and out, making it easy for programs to be launched and tested on a wide variety of platforms with few modifications to actual code.

The first section is this introduction. Section B.2 is a quick-start guide to allow users to pick up the standard testbed configuration quickly and use it with their code. Section B.3 goes over the general design and introduces some of the concepts used by the testbed. Section B.4 covers in more detail the interactions between the tasks and the testbed. Finally, Section B.5 covers the interactions between platforms and the testbed.

Some commonly used terms are defined below:

Application MATLAB code that, in this context, contains tasks that are launched through the testbed.

Platform An environment (hardware and software) on which a task can run.

Platform interface object A MATLAB object used to interface with a given platform.

Platform map A file used by the testbed to determine which platform interface objects to use with which platforms.

Platform name The name given to a particular environment; this name is used to refer to specific platforms in both the task and platform maps.

Task A MATLAB script or executable binary that is used and launched on various platforms.

Task map A file that the testbed uses to determine what tasks are available, where they should be run, and what options are used for a particular task/platform pair.

B.2 Tutorial

This section gives a quick introduction to using the PCA software testbed given the standard testbed and PCA kernel distribution configuration. This should allow a user to get the testbed running relatively quickly.

<tb-root> will designate the path to the PCA testbed software; unless you are using a custom installation of the testbed, <tb-root> should be /data/pca/testbed. <task-root> will be used to designate where you choose to install the PCA kernel distribution. Finally, <task-path> will denote the path holding the kernel code you write.

B.2.1 Set up the task hierarchy

1. Get a copy of all the tasks:
   Latest: cvs -d/data/cvsroot co -d<task-root> pca/tasks
   Stable: cp -rf /data/pca/tasks <task-root>

2. Install your own copy of Raw's Starsearch software. The install should be located at /data/pca/raw_dist/starsearch.tar.gz. Consult New_Users.README within the Starsearch directory to finish setting it up. Finally, edit <task-root>/config.mk and set the variable STARSEARCH_PATH to point to it.

B.2.2 Creating a new task

1. Add your new task to the task hierarchy.

   - Create <task-path>, which should be of the form <task-root>/name_of_kernel/platform_name.
   - Add your task to <task-root>/task.map. Add your entry between .begin(taskoptions) and .end(taskoptions). Your entry must follow this format:

     <kernel name> <platform> <options>

     Valid options and examples are documented in the task mapping. At the very least, you want path=<task-path> specified. If <task-path> is specified as a relative path, it is relative to the location of the task map. This is also where you want to put setupIn and setupOut options if you are using the MIT Raw processor.
   - By convention, all code related to this Raw kernel (including any related MATLAB scripts) are placed in <task-path>.

2. Setup the build system.

   - Get a copy of depend.mk, which is needed to build and launch your task. For the MIT Raw processor:

     cp <task-root>/templates/dependRaw.mk
     <task-path>/depend.mk

     Afterwards, open it and follow the documentation inside to customize it accordingly.
   - Finish up the setup by typing:

     cd <task-root>; ./setup.pl <tb-root> -f

B.2.3 Running the task

1. In MATLAB, create a testbed object by calling its constructor. Make sure the testbed is in the path.

```
addpath <tb-root>
tb = Testbed;
```

2. Run your task with the testbed object as the first parameter. For a function `taskFunction()` that takes in two values and outputs two values, you would call it in the following way:

```
[out1, out2] = taskFunction(tb, in1, in2);
```

   If timing needs to be done, it is called as follows:

```
[out1, out2, time] = taskFunction(tb, in1, in2);
```

If everything is set up correctly, you should see your task run.

B.3 Overview

The testbed is designed to fulfill the following requirements:

1. Allow tasks to be launched on different platforms from MATLAB, whether directly or in an application.
2. Make it easy to change where tasks are launched.
3. Provide facilities that will make it easy to time tasks.
4. Make the testbed extensible, e.g. make it easy for platforms to be added.

The testbed achieves these goals by acting as a common layer between the MATLAB application which launches the tasks and the various underlying platforms on which the tasks run. For each platform, an interface object exists that the testbed uses to launch a task with the appropriate inputs and outputs. These objects follow a standard API; thus, the testbed's functionality can be easily extended to other platforms. Tasks require minimal modifications to run on the testbed, and maps make it easy to change where to launch a task or what options to use.

The following steps (corresponding to Figures 57 and 58) describe the process of launching a task on a remote platform. Relevant sections are listed in brackets.

1. The user application reaches a invocation for a particular task. This call takes an instantiated testbed as an extra parameter. [Section B.4.1]
2. The testbed intercepts the call and looks up the platform specified for this task. [Section B.4.2]

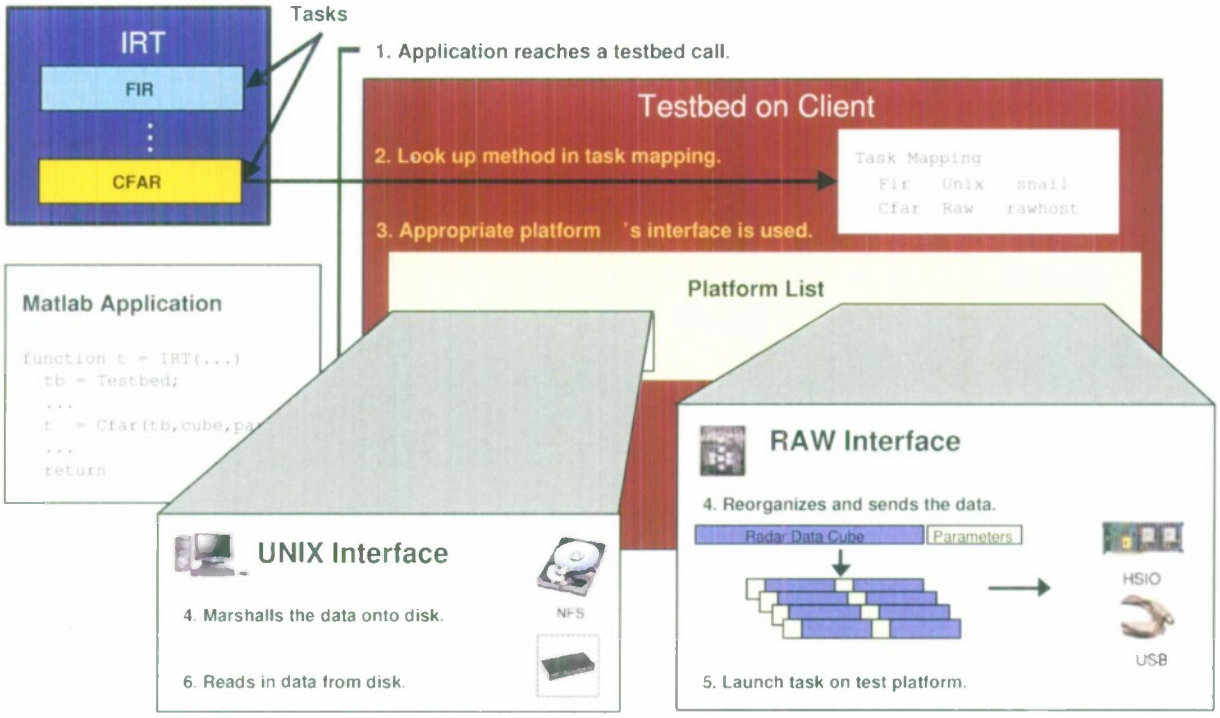

Figure 57. The client side of the testbed software shown running the CFAR kernel.

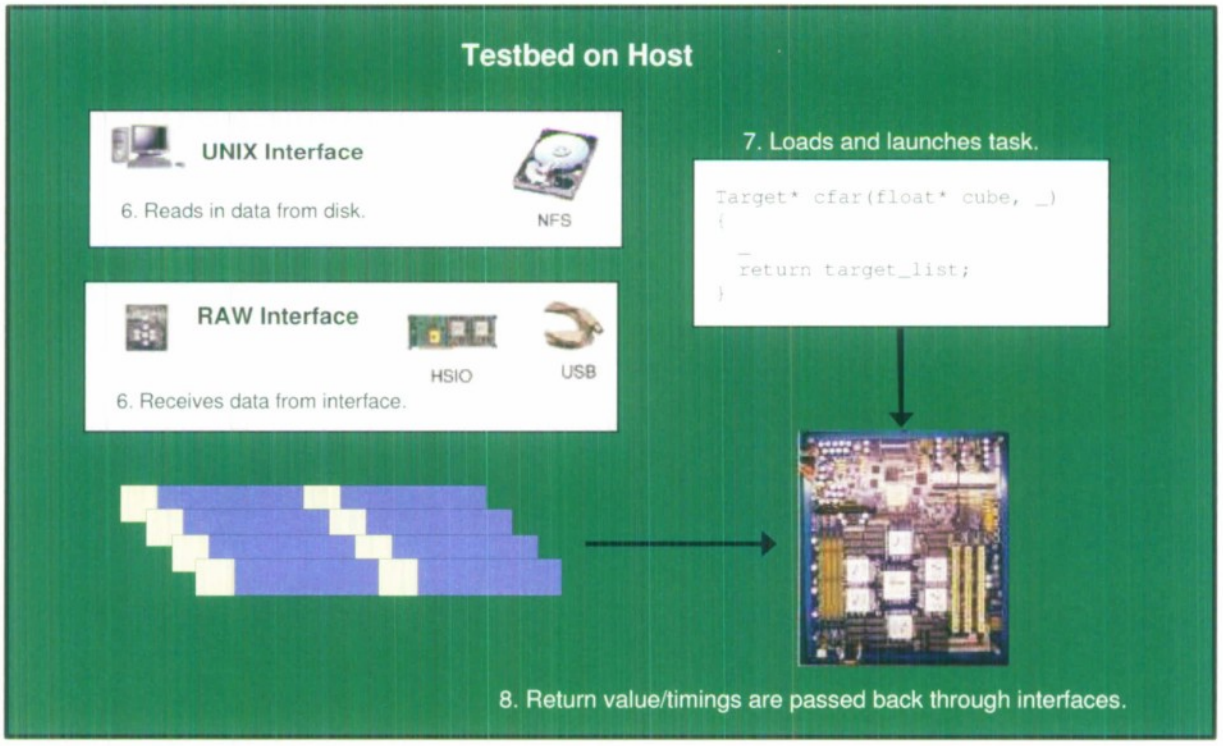

Figure 58. The host platform's side of the testbed software.

3. The testbed uses a platform interface object corresponding with that particular platform. [Section B.5.1, B.5.2]

4. The platform interface object manipulates the data as needed for communication to the platform. [Section B.5.3]

5. The testbed launches the task's code on the remote platform. [Sections B.5.3]

6. The running task loads or receives the data from the testbed. [Section B.5.3]

7. Return values and timings are passed back through similar communication channels. [Section B.5.3]

8. The testbed receives this and returns it to the caller.

B.4 Tasks

This section covers the interaction between the application's tasks and the testbed.

B.4.1 User Application Requirements

A user uses the testbed by calling tasks in a testbed-specific fashion. In order for data communication between the client application and the platform to work correctly, the application must be running on a machine that can directly send data to and receive data from the platform. UNIX machines, the Raw simulator, and the Raw USB interface all use data files to communicate; thus, the client and platform must be able to write and read data from a shared disk. The Raw High-Speed I/O interface (HSIO, described in Appendix A) uses function calls to perform I/O; thus, the client application must be running on the machine on which the Raw HSIO interface is connected.

The user program must first create a testbed object by calling `Testbed` and assigning it to a variable (e.g. `tb = Testbed`). This sets up the testbed, including instantiating platform interface objects, and loading the task map. If the task map changes, the testbed must be re-created to reflect the change.

From this point, for every task that needs to be run, the user application calls the method with the testbed object as its first parameter. For example, if the original call was

```
Targets = Cfar(C, Params),
```

the call should now be

```
Targets = Cfar(tb, C, Params).
```

This allows the testbed to intercept the invocation and handle it accordingly.

```
#######################################################################
# TASK-PLATFORM MAPPING
#######################################################################
.begin(taskplatform)

Cfar        RawSim      rawhost
GenAlg      Unix        ville

.end(taskplatform)


#######################################################################
# TASK-OPTIONS MAPPING
#######################################################################
.begin(taskoptions)

Cfar        RawSim      path=cfar/raw;
                        inPort=0,3,8,11; outPort=0,3,8,11
Cfar        RawUSB      path=cfar/raw-usb;
                        inPort=12,15; outPort=12,15
Cfar        *           inType=int32,float32; outType=int32; repeat=10
                        setupIn=setupIn; setupOut=setupOut; verbose=1

GenAlg      Unix        path=genalg/unix
GenAlg      RawHSIO     path=genalg/raw; inPort=0; outPort=15
GenAlg      *           inType=float32,float32; outType=int32;
                        setupIn=setupIn; setupOut=setupOut; verbose=1
.end(taskoptions)
```

Figure 59. A sample task map with two tasks: CFAR and Genetic Algorithm.

B.4.2 Task Map

In order for the testbed to understand what tasks are available for running, what options to use, and where to run a particular task, the testbed uses a task map file. Every task that is run on the testbed must be specified in the task map. By maintaining an external map file, the actual source code does not need to be changed in order to run tasks on a different platform or change task metadata. By default, the testbed uses a task map either located on the MATLAB path or in `<task-root>/task.map`. The following paragraphs will provide a brief summary of the grammar used in the file; Section B.6 contains the complete specification including the definition of the tags used here.

To determine where to run a particular task, the map contains a mapping from each task to some platform and hostname. These entries are contained within an area denoted by a `.begin(taskplatform)` and `.end(taskplatform)`. The format of each mapping is:

```
<TaskName> <PlatformName> <Hostname>
```

Note that the mapping is whitespace-delimited.

The task map is also used to determine specific options and metadata corresponding with a particular task and platform. For example, the user could specify that when the CFAR runs on the Raw simulator, the task's inputs should be streamed into ports 0 and 3. This area is denoted by a `.begin(taskoptions)` and `.end(taskoptions)`; each entry consists of a task, a platform, and the platform-specific options for this task in the following format:

```
<TaskName> <PlatformName> <Options>
```

Note again that the components are whitespace-delimited.

Multiple-line entries should leave whitespace in the first column of each subsequent line. Task options that should be affiliated to all platforms (for example, the types of the inputs and outputs) should be labelled with a `*` as the platform. Finally, comments in the task map are designed by placing a `#` in the first column of a line.

Figure 59 shows an example task map. The specific supported options and needed metadata are platform-dependent and are documented in the `task.map` included in the testbed distribution as well as in Section B.5.4.

In addition to the task map, the testbed provides a method for setting/overriding the mappings specified in the task map at run-time:

```
testbed = setTaskMap(testbed, task, platform, [hostname])
```

where `testbed` is an instantiation of the testbed, `task` is the name of the task, `platform` is the name of the platform to run on, and `hostname` is an optional parameter specifying which machine to use. The testbed will use the task/platform options specified in the task map.

Timing

One of the requirements of the testbed is to be able to get timings from task executions. This is accomplished by adding an extra return value in the invocation of a task. This extra return value should not be reflected in the task map (e.g. as one of `outType`'s values). For example, if

```
Targets = Cfar(tb, C, Params),
```

was an untimed CFAR invocation, the new invocation would be

```
[Targets, Timings] = Cfar(tb, C, Params).
```

If `Targets` is of type `float32`, the specified `outType` should just be `float32` regardless of whether the first or second invocation of `Cfar()` is used.

B.4.3 PCA Kernel Distribution

The implementations of the PCA kernels were written to use the PCA testbed. These tasks follow a relatively straightforward interface that may be useful for use in future tasks. See section B.2.1 on how to obtain a copy of the PCA kernel distribution.

File Hierarchy

The included tasks are located in `<task-root>`. They are organized as subdirectories named after the task name. Within each task subdirectory, there is one subdirectory for every platform's implementation.

Build/Launch System

The standard tasks included with the testbed utilize a `Makefile` system to simplify building and launching tasks. The following is a list of files used by the standard task structure:

- `<task-root>/build.mk` - The base `Makefile` used. All the standard tasks create a symbolic link named `Makefile` in their local directories that point to this file, which in turn links to the testbed's standard remote execution system (see Section B.5.3). By linking to this, all standard tasks share a common interface.

- `<task-root>/config.mk` - A global configuration file for all the tasks using `build.mk`. This allows build flags and other options to be specified at a global level.

- `<task-root>/depend.mk` - Each task for each platform specifies what files are needed to build the needed executable. Depending on which platform is being used, the `depend.mk` file may need to define different variables. For ease of use, standard templates for the testbed-provided platforms (see Section B.5.4) are located in `<task-root>/templates/depend<platform>`.

To further simplify this process, a setup script (`setup.pl`) is available in the task's root directory; this script sets up all the needed symbolic links for the tasks listed in the task map. This script also affiliates the task distribution with a particular testbed installation. Thus, to use a new testbed distribution with an existing task distribution, one merely has to rerun the setup script and tell it to set up the build system to use the new testbed.

By using this system of `Makefiles`, the testbed easily builds and launches tasks. More information on how the actual standard platform interfaces use this system is available in Section B.5.4.

B.4.4 MIT Raw: Data Reorganization

In order to achieve maximum efficiency on Raw-based platforms, the data may need to be split up, interleaved with other parameters, and manipulated in various ways before being sent to specific ports. To accommodate this, `setupIn=<`MATLAB `function>` and `setupOut=<`MATLAB `function>` options exist for the task map. This option allows the user to specify two functions, one which reorganizes the input and the other the output. These functions are called right before input is marshalled and after output is unmarshalled. The interface of the input setup function must be:

```
function out = taskSetupIn(inData, ports)
```

where `inData` is a cell array of the passed-in parameters (not including the testbed object), `ports` are the input ports being used, and `out` is a cell array where `out{x}` should correspond to data going to input port `ports{x}`. As an example, calling `Cfar(testbed, C, params)`, `inData{1} = C` and `inData{2} = params`. The port numbers are documented in `depend.mk` file.

Several issues of note:

- Data is written out in a column-major format.
- If multiple input types are specified, `out{x}{y}` will be serialized using the types one after another, repeating them once all are used. For example, if there are two types specified, `inType=int32, float32`, and there are four elements `out{x}{1},...,out{x}{4}` being sent to port `ports{x}`, `out{x}{1}` will be serialized as type `int32`, `out{x}{2}` as `float32`, `out{x}{3}` as `int32` and `out{x}{4}` as `float32`. If `out{x}` is a non-cell array, it will be serialized as the first type specified. The testbed will output an error message if the number of elements being sent to a specific port is not a multiple of the number of types.
- To send complex data to the Raw, merely pass it back as one of the `out`'s values. The testbed sends in complex data by interleaving the real and imaginary parts of each element. Note that the testbed uses MATLAB's `iscomplex()` function to determine whether to send the data in as real or complex numbers. It is advisable to ensure that the data sent is always either real or complex (see MATLAB's `complex()` function to mark a matrix as complex regardless of whether there is an imaginary part or not).

The interface of the output setup function must be:

```
function rv = taskSetupOut(fileData, ports)
```

where `fileData` is a cell array where `fileData{x}` corresponds to output coming from port `ports{x}`, `ports` is a listing of the output ports and `rv` is a cell array where `rv{y}` should correspond to the yth return value of the original MATLAB call. For example, if the original MATLAB call was `(A, B) = Kernel(tb)`, `rv{1}` corresponds to `A` and `rv{2}` to `B`. Again, add the `setupOut` key to the appropriate entry in the task map (e.g. `setupOut=taskSetupOut`).

There are several issues of note:

- The cycle count is *not* counted as a return value. This means that the cycle count should be returned as the $n+1^{th}$ return value if the function normally has `n` return values.
- The testbed currently only supports *one* output type. If multiple output types are specified for a kernel, it will automatically default to single-precision (32-bit) floating point. Make sure that all output (including the cycle count) is of the same type.

B.5 Testbed

This chapter will detail the components of the platform interface objects and the associated platform map. All these files are located within the `<tb-root>` subdirectory. These objects are used to facilitate the launching of tasks on and sending/receiving of data to/from remote platforms.

```
##########################################################
# PLATFORM-IFO MAPPING
# platform  interface    options
##########################################################
RawHSIO   RawIFO       mode=hsio
RawUSB    RawIFO       mode=usb
RawSim    RawIFO       mode=sim
Unix      UnixIFO      path=/usr/bin/X11:/tools/gnu/bin
```

Figure 60. A sample platform map for a testbed that supports four different platforms.

B.5.1 Platform Map

The platform map is located in `<tb-root>/platform.map`. Each supported platform contains a line in the platform map specifying the name of the platform, the actual MATLAB class to use to interface with it, and platform-specific metadata. Each entry must follow the following format:

```
<PlatformName> <PlatformInterfaceObject> <Options>
```

See figure 60 for an example of a platform map and chapter B.6 for a complete description of the grammar used, including a definition of `<PlatformName>`, `<PlatformInterfaceObject>`, and `<Options>`. Note that multiple platforms may use the same platform interface object.

When a testbed object is instantiated, each platform listed is constructed by calling by the listed MATLAB object's constructor. Thus, if new platform interfaces are created or the platform map changed, a new testbed object must be created.

B.5.2 Platform Interface Objects

Platform interface objects form the basis of the testbed. Through the use of these objects, the testbed is capable of running on any platform that has an associated object. Platform interface objects are MATLAB objects located within `<tb-root>/pifo` that must have two methods defined: a constructor and `pifoRun()` that runs a task using that particular object. The constructor is passed a single argument: a structure filled with the option keys and values listed in the platform map. `pifoRun()` is called with five parameters:

1. The platform interface object itself.
2. A string containing the name of the task as specified in the task map and as called by the user in MATLAB.
3. A cell array containing the arguments passed by the user during invocation.
4. A structure containing the options specified in the task map.
5. The name of the host on which to run the task.

It returns the values returned by the given task, plus timing if that has been enabled.

By forcing all platform interface objects to follow a standard API, it makes the testbed easily extensible: a new platform can be interfaced by creating an object with the same API.

B.5.3 Common Functionality

The methods by which code is launched on remote platforms and data is sent and received from a particular platform heavily depend on the platform itself. A platform interface object can do whatever is necessary to perform the needed operation as long as it follows the API. However, there is some commonly used functionality that can be shared between various platform interface objects. Some of these have been provided by the testbed and are described below. Note that a custom platform interface object need not use these; they are merely provided for convenience.

Remote Execution

It is expected that tasks will often be launched on different platforms and not on the same machine that the application is running on. Thus, there is a definite need for remote launching capability.

The testbed provides two shell scripts that help provide the needed functionality. They are located in `<tb-root>/pifo` and are named `runRsh.sh` and `runPlatform.sh`. These two scripts set up the remote execution as well as parse passed-in options and create the necessary command-line to run the task.

In addition, a set of `Makefiles` are used in order to provide building and launching capability. Together with the shell scripts, they provide the ability for platform interface objects to `rsh` into remote machines, communicate various options, and build/launch a given task. In order for a platform interface object to use this, the task to be launched must have a `Makefile` that interfaces correctly with `runPlatform.sh`. This can be ensured by linking to the testbed interface's `Makefiles`, which are located in `<tb-root>` and are named `make<platform>.mk`. The PCA kernel benchmarks discussed in Section B.4.3 demonstrate how this interfacing is done.

Interprocess Communication

In order to facilitate I/O with remotely launched tasks, the testbed provides two interprocess communication objects, both of which are located in `<tb-root>/ipc`. This directory also contains common functions used by these two objects. The `FileIO` object provides file I/O serialization. Given some data, it generates a data file with a small header that describes the data and allows for proper unmarshalling. Note that data is written in column-major order. The `RawHSIO` object utilizes the High-Speed I/O interface and associated libraries developed to provide high-speed, streaming I/O to the MIT Raw board. Both objects provide the same API function calls:

`FileIO()` or `RawHSIO()` - Constructs the appropriate IPC object.

`initIO(ipc, in, out)` - Initializes the input and output accordingly. This should be called once per task but does not need to be called every iteration.

`sendIn(ipc, in)` - Sends in the data into the kernel. This should be called every iteration.

`data = recvOut(ipc, out)` - Receives output from the kernel.

`destroyIO(ipc, in, out)` - Cleans up the I/O.

B.5.4 Provided Interface Objects

This section describes the standard interface objects that are provided by the testbed. They are reference implementations for several platforms of interest.

The following values must be defined for every task by all the interface objects:

path The path of the task; if a relative path is specified, it must be specified relative to the location of the task map.

inType The input types. Valid values are `int32` and `float32`.

outType The output types (not including the timing). Valid values are `int32` and `float32`.

The following is an additional option supported by all interface objects:

repeat Number of iterations to repeat a particular task (default 1).

MatlabIFO

This object is used by the platform named `Matlab`. This interface is rather simple and essentially just passes the call on to the appropriate MATLAB function. It also performs timing as needed.

UnixIFO

This interface is appropriate for machines with UNIX-like interfaces and with access to a shared disk (such as Linux and Mercury). The platforms that use this object are named `Unix` and `Mercury`.

Machines utilizing this interface generally share a shared disk (e.g. on NFS) with the client machine. Thus, code is launched by merely using `rsh` to connect to the machine and launch the task. Sending data is done by serializing the data and writing it to a shared location, followed by launching the task with a pointer to the data files. Output is done in a similar way.

Note that most kernels launched in this way are written in C++ and use the templated `KernelDemo` interface to handle I/O and setup. This interface specifies which public methods a kernel must have as well as providing support for timing and running multiple iterations. See `<tb-root>/include/KernelDemo.h` for more details.

The following are additional task options supported by this interface object:

verbose Turn on verbose mode, i.e. show all output from using the testbed. 0 disables this, any non-zero value enables it (default 0).

debug Turn on debug mode, i.e. runs the appropriate debugger with the task. 0 disables this, any non-zero value enables it (default 0).

clean Perform a `make clean` to clean up a path before building/launching. 0 disables this, any non-zero value enables it (default 0).

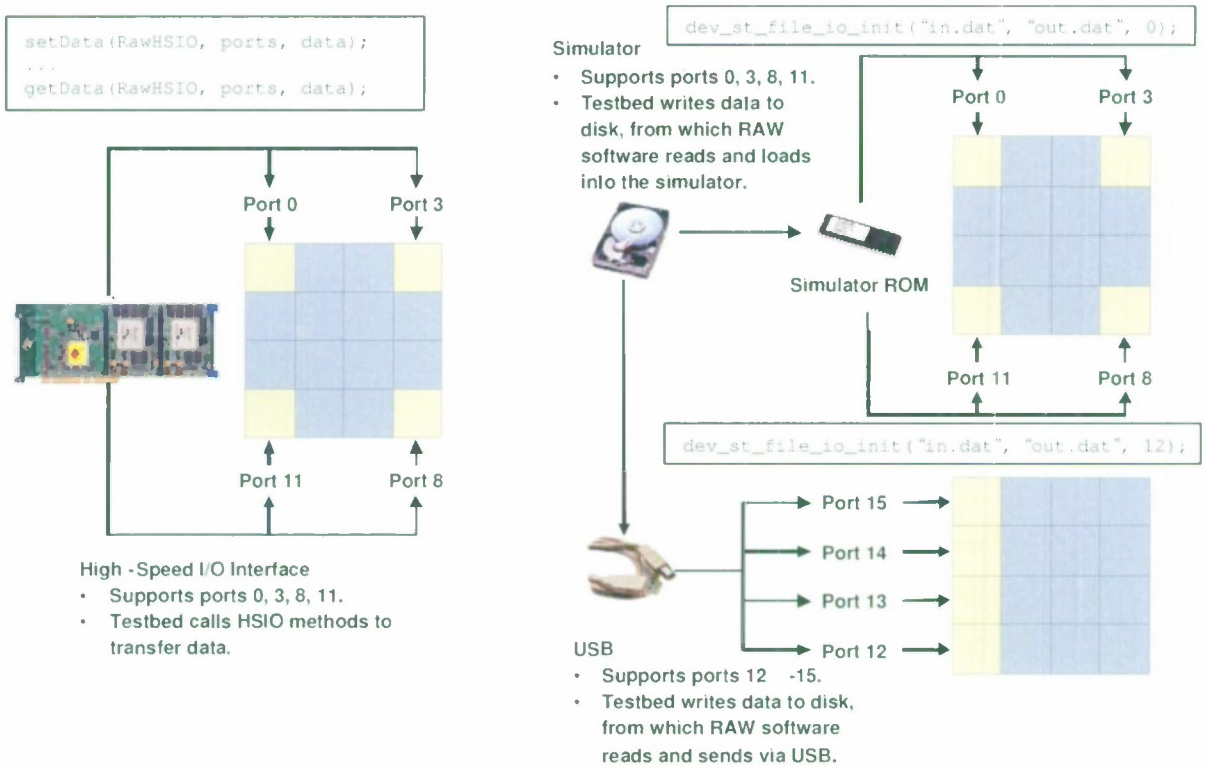

Figure 61. Methods used to send data to the various Raw instantiations.

The platforms that use this object are named `RawSim`, `RawUSB`, and `RawHSIO`. This interface is for use with the MIT Raw processor. `rsh` is used to access the desired machine. For `RawSim`, this machine is used to run a simulator with the given code. To send and receive data, the simulator uses `bC` code that can simulate an I/O device in order to communicate with its environment. The platform interface object communicates then by writing the data to files and connecting these I/O devices to particular files and ports in the simulator. Afterwards, these "devices" are read in. For `RawUSB`, MIT-provided binaries allow us to launch tasks on the board as well as perform I/O using a similar interface as the simulator. For `RawHSIO`, instead of using the simulator interface for I/O, we use a different High-Speed I/O interface.

Note that due to the distributed nature of the MIT Raw architecture, the `RawIFO` allows the user to provide MATLAB functions in order to reorganize both the Raw's inputs and outputs. This also forces certain restrictions on the types of the outputs. Usage is discussed in further detail in Section B.4.4.

The following are additional task options supported by this interface object:

verbose Turn on verbose mode, i.e. show all output from using the testbed. 0 disables this, any non-zero value enables it (default 0).

debug Turn on debug mode, i.e. runs the appropriate debugger. 0 disables this, any non-zero value enables it (default 0).

clean Perform a `make clean` to clean up a path before building/launching. 0 disables this, any non-zero value enables it (default 0).

inPort The input port(s) to use on the MIT Raw. The supported ports depend on the mode.

outPort The output port(s) to use on the MIT Raw. The supported ports depend on the mode.

setupIn The function used to reorganize input (data being sent from the MATLAB application to the task). Valid values are any legal MATLAB function name.

setupOut The function used to reorganize output (data being returned from the task to the MATLAB application). Valid values are any legal MATLAB function name.

metadata Indicates which parameter is metadata. This is used to support datasets on the Raw platform but isn't fully implemented. Avoid using this functionality.

The following are additional platform options supported by this interface object:

mode Specifies what mode to run the Raw interface in (`sim` for simulator, `usb` for USB, `hsio` for high-speed I/O).

In addition, because of the relative difficulty in programming the MIT Raw, the interface also provides in `pca.h` some useful functions/macros for programming the MIT Raw:

`pca_test_done(x)` - Equivalent to C's `exit(x)` function; `x` is the return value.

`pca_tiles_side()` - The number of tiles on the side of the Raw processor.

`pca_get_x()` - Gets the x-coordinate of the calling tile.

`pca_get_y()` - Gets the y-coordinate of the calling tile.

`pca_get_id()` - Defined to be

$$\texttt{pca_get_id()} = \texttt{pca_get_y()} \cdot \texttt{pca_tiles_side()} + \texttt{pca_get_x()}$$

`pca_init_switch()` - Initializes the switch processor; either `pca_init_switch_pc()` or this must be called before any switch calls (e.g. `pca_set_switch_pc()`, `pca_barrier()`).

`pca_init_switch_pc(label)` - Initializes the switch processor and sets its PC to `label`. It is safe to use this function to set the switch PC at any time. Either this or `pca_init_switch()` must be used before any switch calls.

`pca_set_switch_pc(label)` - Sets the switch PC to `label`.

`pca_barrier()` - Sets up a barrier by which all tiles must reach before continuing. This function changes the PC on the switch processor. Note that all incoming data should be cleared from a tile's switch before this is called.

`pca_init_io(port)` - Used to initialize I/O at port `port`; must be called before that port is used. This can be safely called from all tiles. `port` can either be a number or `PORT_NW`, `PORT_NE`, `PORT_SW` or `PORT_SE`.

`pca_init_io_dir(dir)` - Used to initialize I/O in a given direction; `dir` is the bitwise-or of `NORTH`, `SOUTH`, `WEST` and/or `EAST`.

`pca_sync_io(port)` - Used to finish up I/O at the completion of the task. This is commonly called several times at the end of a kernel with the different ports used and is usually followed by a `pca_barrier()`. Make sure you read in all the data waiting at `port`; otherwise, I/O may not flush completely before this call returns.

`pca_sync_io_dir(dir)` - Same as `pca_sync_io()`, except takes in a direction like `pca_init_dir()`.

B.6 Map Grammar

The testbed maintains a simple and consistent grammar to specify both the task and platform maps, which are essential structures in the operation of the testbed. This chapter serves as a reference for the grammars used in those files. Note that `<Tag>*` represents one or more distinct instances of `<Tag>` objects, and parentheses are used for grouping and are not actually part of the actual grammar.

B.6.1 Whitespace, Continuations, and Comments

`<ws>` is defined to be spaces and tabs. `<EmptyLine>` is one that is filled with zero or more `<ws>` objects. Overly long lines can be broken into multiple individual lines; each line that contains `<ws>` in the first column is considered a continuation of the previous line. All other non-empty lines must contain some form of text in the first column.

Comments are lines ignored by the testbed. Note that the # must be in the first column.

```
CommentLine := # Comment Text
IgnoredLine := <CommentLine> | <EmptyLine>
```

B.6.2 Options

Options are specified very simply as semicolon-separated `<Key>=<ValueList>` pairs, where `<Key>` follows MATLAB naming conventions and the `<ValueList>` is a comma-separated list of `<Value>` objects, which in turn must be valid MATLAB strings or numbers.

```
ValueList   := (<Value>,)*<Value>
Options     := <EmptyLine>
               | (<Key> = <ValueList>;)* <Key> = <ValueList>
```

B.6.3 Task Map

The task map contains two sections: one that represents the mapping of tasks to specific platforms and hosts, and one that represents the options used for a particular task/platform pair. The following are lines that are legal in the task mapping and task option sections, respectively.

```
TaskMapLine := <TaskName><ws><PlatformName><ws><Hostname>
               | <IgnoredLine>
TaskOptLine := <TaskName><ws><PlatformName><ws><Options>
               | <IgnoredLine>
```

where `<TaskName>` must follow MATLAB naming conventions, `<PlatformName>` must follow MATLAB naming conventions or be a `*`, and `<Hostname>` represents a legal hostname for a machine. Note that `<PlatformName>` in the task map should match up to some `<PlatformName>` in the platform map. If `*` is specified as the `<PlatformName>`, those options are applied to all platforms. If, for a particular task, a specific key is defined for both a specific platform and all platforms, the values specified for a particular platform take precedence.

Finally, we can specify the format of the entire task map from these constructs. Note that each line should be separated by a newline.

```
TaskMap :=
<IgnoredLine>*
.begin(taskplatform)
<TaskMapLine>*
.end(taskplatform)
<IgnoredLine>*
.begin(taskoptions)
<TaskOptLine>*
.end(taskoptions)
<IgnoredLine>*
```

B.6.4 Platform Map

The platform map has a similar structure to the task map. This is the only legal line in the platform map:

```
PlatformMapLine := <PlatformName><ws><PlatformInterfaceObject>...
                   <ws><Options> | <IgnoredLine>
```

where `<PlatformInterfaceObject>` follows MATLAB naming conventions.

We can use this line to specify the format of the entire platform map. Note that each line should be separated by a newline.

```
PlatformMap :=
<IgnoredLine>*
<PlatformMapLine>*
<IgnoredLine>*
```

B.6.5 Example

See Figure 59 for an example of a task map and Figure 60 for an example of a platform map.

REFERENCES

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

2. Lawrence Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

3. José L. Ribeiro Filho, Philip C. Treleaven, and Cesare Alippi. Genetic-algorithm programming environments. *IEEE Computer*, 27(6):28–43, June 1994.

4. David E. Goldberg, editor. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Van Nostrand Reinhold, New York, 1991.

5. Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, 1996.

6. Ryan Haney, Theresa Meuse, Jeremy Kepner, and James Lebak. The HPEC challenge benchmark suite. In *Proceedings of the Ninth Annual High-Performance Embedded Computing Workshop (HPEC 2005)*, Lexington, MA, September 2005.

7. Henry Hoffmann. Stream Algorithms and Architecture. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2003.

8. Intel Corporation. *Intel Xeon Processor with 512-KB L2 Cache at 1.80 GHz to 3 GHz*, March 2003.

9. Jason Sungtae Kim, Michael Bedford Taylor, Jason Miller, and David Wentzlaff. Energy characterization of a tiled architecture processor with on-chip networks. In *International Symposium on Low Power Electronics and Design (ISLPED 2003)*, pages 424–427, Seoul, Korea, August 2003. Association for Computing Machinery.

10. James Lebak, Hector Chan, Ryan Haney, and Edmund Wong. Polymorphous computing architectures (PCA) kernel benchmark measurements on the PowerPC G4. Project Report PCA-Kernel-2, MIT Lincoln Laboratory, Lexington, MA, January 2004.

11. James Lebak, Albert Reuther, and Edmund Wong. Polymorphous computing architectures (PCA) kernel-level benchmarks. Project Report PCA-Kernel-1, MIT Lincoln Laboratory, Lexington, MA, January 2004.

12. Mercury Computer Systems. *500 MHz PowerPC 7410 Daughtercard product data sheet*, March 2002.

13. Jason Miller. Private communication, August 2004.

14. Motorola Semiconductor Products. *AltiVec Technology Programming Interface Manual*, June 1999.

15. Motorola Semiconductor Products. *MPC7410 RISC Microprocessor Hardware Specifications*, January 2002.

16. Rodric M. Rabbah, Ian Bratt, Krste Asanovic, and Anant Agarwal. Versatility and VersaBench: A new metric and a benchmark suite for flexible architectures. Technical Memo MIT-LCS-TM-646, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, MA, June 2004.

17. David A. Schwartz, Randall R. Judd, William J. Harrod, and Dwight P. Manley. Vector, signal, and image processing library (VSIPL) 1.0 application programmer's interface. Technical report, Georgia Tech Research Corporation, 2000. http://www.vsipl.org.

18. M. Srinivas and Lalit M. Patnaik. Genetic algorithms: A survey. *IEEE Computer*, 27(6):17–26, June 1994.

19. Volker Strumpen, Henry Hoffmann, and Anant Agarwal. A stream algorithm for the SVD. Technical Memo MIT-LCS-TM-641, Massachusetts Institute of Technology Laboratory for Computer Science, 2003.

20. Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–36, March/April 2002.

21. Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw microprocessor: an exposed-wire-delay architecture for ILP and streams. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA 2004)*, pages 2–13, Munich, Germany, 19-23 June 2004. IEEE Computer Society.

22. Charles F. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.

23. Dale Varberg, Edwin J. Purcell, and Stephen Rigdon. *Calculus*. Prentice-Hall, 8th edition, 1999.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD-MM-YYYY)*
14 June 2006

2. REPORT TYPE
Project Report

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Polymorphous Computing Architecture (PCA) Kernel Benchmark Measurements on the MIT Raw Microprocessor

5a. CONTRACT NUMBER
FA8721-05-C-0002

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
R.J. Haney, J.M. Lebak, M.A. Alexander, H. Chan, P.A. Jackson, E.L. Wong

5d. PROJECT NUMBER
1084

5e. TASK NUMBER
0

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420-9108

8. PERFORMING ORGANIZATION REPORT NUMBER
PCA-Kernel-3

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA/IPTO
3701 Fairfax Drive
Arlington, VA 22203-1714

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
ESC-TR-2006-063

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

ADA 440246

14. ABSTRACT

The DARPA Polymorphous Computing Architecture (PAC) program is a research initiative aimed at developing new computer architectures with a high degree of flexibility. Unlike current computer architectures that are rigid in nature, PCAs will have the capability to adapt ("morph") to match the problem being solved. This flexibility will allow higher overall system performance in a broad range of applications.

MIT Lincoln Laboratory has defined a set of kernel benchmarks for the PCA program. The kernel-level benchmarks have been chosen to stress both computation and communication aspects of the architecture. The particular benchmarks chosen are based on the frequency of their use in current and future applications. They are drawn from the areas of signal and image processing, communication, and information and knowledge processing. Each of these areas imposes different processing requirements on the architecture in terms of operations performed and memory bandwidth required.

This document describes a set of measurements of the PCA kernel benchmarks on a prototype PCA chip.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE	Same as report	108	19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified			