

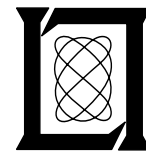
**Project Report
PCA-KERNEL-1
Revision 1**

Polymorphous Computing Architecture (PCA) Kernel-Level Benchmarks

**J. Lebak
A. Reuther
E. Wong**

13 June 2005

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.


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, ITO, 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 is given to destroy this document
when it is no longer needed.

Massachusetts Institute of Technology
Lincoln Laboratory

**Polymorphous Computing Architecture (PCA)
Kernel-Level Benchmarks**

*J.M. Lebak
A. Reuther
E. Wong
Group 102*

Project Report PCA-KERNEL-1, Revision 1

13 June 2005

Approved for public release; distribution is unlimited.

Lexington

Massachusetts

Acknowledgments

The authors acknowledge contributions from the MIT Lincoln Laboratory PCA team, including Bill Coate, Janice McMahon, Masahiro Arakawa, Robert Bond, Hector Chan, Ryan Haney, Matt Alexander, and Jeanette Baran-Gale.

The PCA community suggested some of the additional kernel benchmarks referred to but not described in this report. Image processing kernel benchmarks were suggested by Mark Richards of the Georgia Tech Research Institute. The exception is the image compression benchmark, which is based on work done by Baxter and Seibert [1]. The incomplete gamma function was suggested as a kernel by James Lyke of the Air Force Research Lab.

Helpful input on the metrics was provided by members of the Morphware Forum, including Steve Crago, Dennis Cattel, Mark Richards, Randy Judd, and others. In particular, Jinwoo Suh of the University of Southern California/Information Sciences Institute (USC/ISI) pointed out an error in the CFAR section of the document.

TABLE OF CONTENTS

Acknowledgments	iii
List of Illustrations	vii
List of Tables	ix
1. Introduction	1
2. Metrics	3
3. Signal Processing Benchmarks	7
3.1 Finite Impulse Response Filter Bank	7
3.2 QR Factorization	8
3.3 Singular Value Decomposition	8
3.4 Constant False Alarm Rate Detection	10
4. Communication Benchmark	13
5. Information and Knowledge Processing Benchmarks	17
5.1 Pattern Matching	17
5.2 Database Operations	17
5.3 Graph Optimization via Genetic Algorithm	20
6. Further Kernel Benchmarks	25
REFERENCES	27
APPENDIX A – Revisions	29

LIST OF ILLUSTRATIONS

Figure No.		Page
1	Sliding window in CFAR detection.	11
2	C corner turn example.	14
3	VSIPL corner turn example.	15
4	Outline of the pattern match kernel.	18
5	Structure of a simple genetic algorithm.	21

LIST OF TABLES

Table No.		Page
1	Benchmark metrics	5
2	FIR filter bank input parameters	7
3	QR input parameters	8
4	SVD input parameters	8
5	Parameter sets for the CFAR Kernel Benchmark	12
6	Corner turn parameters	16
7	Pattern matching parameters	18
8	Tracking parameters	19
9	Parameter sets for the Genetic Algorithm Kernel Benchmark	23

1. Introduction

This document describes a series of kernel benchmarks for the PCA program. Each kernel benchmark is an operation of importance to DoD sensor applications making use of a PCA architecture. Many of these operations are a part of the composite example application described elsewhere [11].

The kernel-level benchmarks have been chosen to stress both computation and communication aspects of the architecture. “Computation” aspects include floating-point and integer performance, as well as the memory hierarchy, while the “communication” aspects include the network, the memory hierarchy, and the I/O capabilities. 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 processing, communication, and information and knowledge processing. The specification of the benchmarks in this document is meant to be high-level and largely independent of the implementation.

2. Metrics

For each benchmark, a set of problem sizes are defined. 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 $i = 1, 2, \dots, N_k$, and N_k varies from kernel to kernel. We assume that the data for the problem begins in a “staging area” accessible to the PCA computation units (“main memory” or “an I/O stream”) and must be moved into local memory. For the initial realization of the benchmarks, the staging area can be main memory. Final measurements in the next phase of the PCA program should have an I/O stream as the staging area.

There are two major metrics of interest for each problem size. The first is the total time or latency, $L_1(k, d_i)$, to perform kernel k for a data set size d_i using a *single PCA prototype integrated circuit* (hereafter, a *PCA chip*). This measurement should include both computation time and the time to move the data for the problem from the staging area (off the PCA chip) to a computation or operation area (on the PCA chip).

The second major metric of interest is the sustained achievable throughput, $T(k, d_i)$. For each kernel k and problem size d_i , a measure of the *workload*, $W(k, d_i)$, is defined in an operation-dependent and system-independent way. (For floating-point computation operations, W is the floating-point operation count, while for communication operations, W is the number of bytes transferred.) The sustained achievable throughput is

$$T(k, d_i) = \lim_{n \rightarrow \infty} \frac{nW(k, d_i)}{L_n(k, d_i)}, \quad (1)$$

where $L_n(k, d_i)$ is the total time to solve n problems of the given type using the PCA chip. As above, $L_n(k, d_i)$ includes the time to move the data from the staging area to an operation area.

There are clear trade-offs between throughput and latency. If the entire PCA chip is being used to solve kernel k for data set d_i , then $L_n = nL_1$ and $T = W/L_1$. In some cases, however, an operation will be able to take advantage of *pipelining* and perform multiple computations of the same type at the same time, resulting in higher throughput. Obviously, the extent to which this can be accomplished will depend on the input bandwidth of the PCA chip. To measure the throughput for our purposes, it is sufficient to measure L_n for a value of n that is sufficiently large (at least $n > 10$, and preferably $n > 100$).

For embedded systems we are interested in the *efficiency* of the operation, that is, the use of resources relative to the potential of the device. In general the efficiency $E(k, d_i)$ is defined as

$$E(k, d_i) = \frac{T(k, d_i)}{U(k)} \quad (2)$$

where $U(k)$ is the kernel-dependent upper bound or peak performance of the chip. The definition of $U(k)$ is linked to the definition of the workload. When W is in floating-point operations, $U(k)$ is the theoretical peak floating-point computation rate (based on the clock rate and the number of floating-point units). For a communication operation, where workload is defined in bytes, $U(k)$ is the theoretical peak bandwidth between the communicating units. For benchmarks other than the signal processing and communication benchmarks, efficiency is difficult to calculate because peak performance for the corresponding workloads cannot easily be defined. For example, the workload

for the database benchmark is in transactions, and there does not exist an easily calculable peak performance for the number of transactions performed. In these situations, efficiency cannot be calculated.

One of the key metrics for the PCA program is *stability* of the performance. Kuck [9, p.168ff] defines stability as the ratio of the minimum achieved performance to the maximum achieved performance over a set of data set sizes and programs. Stability is defined in two senses for the kernel benchmarks, a per-kernel sense and an overall sense. The per-kernel stability is reflected by a metric called *data set stability*, S_d , defined as the stability for a particular kernel over all data sets for that kernel,

$$S_d(k) = \frac{\min_{d_i} T(k, d_i)}{\max_{d_i} T(k, d_i)}. \quad (3)$$

Stability across all kernels poses a problem, as the workloads and thus the throughput calculations are different for different kernels. However, a good indication of the overall stability can be gleaned from the geometric mean of the kernel stabilities,

$$\bar{S}_d = \sqrt[8]{\prod_{k=1}^8 S_d(k)}. \quad (4)$$

Finally, for embedded systems, an important metric is the achieved performance per unit power consumed by the PCA chip,

$$C(k, d_i) = \frac{T(k, d_i)}{P(k, d_i)}, \quad (5)$$

where $P(k, d_i)$ is the overall power consumed during the operation. This normalized quantity C gives some indication of the “cost” of executing the benchmark on the given PCA chip. Obviously, this metric ignores power consumed by other elements of the system, but allows comparison with commercially available processors using the same metric. Performance metrics per unit size and weight are omitted, as the processing unit is perceived to be less of a driver for either of these quantities than for power consumed by the system.

Along with the specified measurements, developers are asked to provide their implementation of the algorithm and a description of chip resource usage. The specific implementation of the benchmark used to achieve the measured latency and throughput is of great interest for several reasons. One important reason is to compare the throughput and latency achievable by different approaches. For this comparison, the availability of multiple implementations of the same algorithm on the same architecture using different approaches would be ideal.

Another important reason to examine the implementation has to do with the workload, $W(k, d_i)$, which is defined for a particular implementation of the kernel. This standard workload and implementation allows comparison of different architectures. If an additional algorithm with a significantly different workload is also implemented, the value of W must be adjusted or the workload is invalid. Therefore, developers are asked to provide an estimate of the workload for implementations that are substantially different from those described in this report.

In summary, developers are requested to measure the latency, throughput, and power consumed for each kernel benchmark k and data set d_i . The theoretical peak floating-point, operation, and

communication rates should be reported for the chip.¹ All other metrics (efficiency, stability over problem size, stability over all kernels, and performance per unit power) are derived from chip parameters and the measured quantities. Other statistics such as variance may be appropriate also and may be calculated from these results. The desired quantities are summarized in Table 1.

Table 1.

Benchmark metrics.

Parameter	Description	Calculation
$L_j(k, d_i)$	Total latency for j problems	measured
$W(k, d_i)$	Workload	given
$T(k, d_i)$	Throughput	$\lim_{n \rightarrow \infty} \frac{nW(k, d_i)}{L_n(k, d_i)}$
$U(k)$	Performance upper bound for operation type:	
	floating-point	clock rate * floating-point units
	communication	bandwidth between communicating units
	integer	clock rate * integer units
$E(k, d_i)$	Efficiency	$\frac{T(k, d_i)}{U(k)}$
$S_d(k)$	Stability over data sets	$\frac{\min_{d_i} T(k, d_i)}{\max_{d_i} T(k, d_i)}$
\bar{S}_d	Mean of data set stabilities	$\sqrt[8]{\prod_{k=1}^8 S_d(k)}$
$P(k, d_i)$	Power consumed	measured
$C(k, d_i)$	Performance-power efficiency	$\frac{T(k, d_i)}{P(k, d_i)}$

¹These rates should be specific to an agreed-upon technology as discussed at the Morphware forum meeting in July 2002.

3. Signal Processing Benchmarks

Four signal processing kernels are included in the benchmark set. They present a range of a different characteristics in terms of operation counts and memory references.

3.1 Finite Impulse Response Filter Bank

The finite impulse response (FIR) filter is one of the basic operations in signal processing. This kernel implements a set of M FIR filters. Each FIR filter m , $m \in \{0, 1, \dots, M - 1\}$, has a set of impulse response coefficients $w_m[k]$, $k \in \{0 \dots K - 1\}$. 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 :

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

The filter is often implemented using fast convolution with the fast Fourier transform (FFT). The most efficient implementation depends on various factors including the size of the filter response vector (for more details, see [12]). We define two data sets in Table 2, one for a short filter and one for a long filter. We define the workload for this kernel as the minimum workload of the time-domain and frequency-domain implementations. For the long filter, the frequency-domain implementation operation count ($M(10N \log_2 N + 8N)$) is given. For the short filter, the time-domain implementation operation count ($8MNK$) is given.

Table 2.

FIR filter bank input parameters.

Parameter Name	Description	Values	
		Set 1	Set 2
M	Number of filters	64	20
N	Length of input and output vectors	4096	1024
K	Number of filter coefficients	128	12
W	Workload (Mflop)	34	1.97

All operation counts for the FIR filter assume complex input and output data. In addition, the frequency-domain operation count assumes that an FFT and inverse FFT are implemented using a radix-2 algorithm. Many other implementations of the FFT are possible: see Van Loan for a discussion of algorithms for performing the FFT [16]. Because such a wide variety of FFT implementations exist, and the particular algorithm implemented in a given library may not be known or may change with data size, it is common to use the radix-2 operation count to calculate throughput even when a different algorithm may be in use.

3.2 QR Factorization

The QR factorization is a linear algebra operation that factors a matrix into an orthogonal component, which is a basis for the row space of the matrix, and a triangular component. In adaptive signal processing, the QR is often used in conjunction with a triangular solver.

The QR factorization of an $m \times n$ matrix A with $m \geq n$ is a pair of matrices $A = QR$, where the unitary matrix Q is of size $m \times m$ and the upper-triangular matrix R is of size $m \times n$. There are many ways of calculating the QR factorization, as discussed in Golub and Van Loan, including the Householder transformation method [7, Algorithm 5.2.1], the Modified Gram-Schmidt algorithm [7, Algorithm 5.2.5], and the Fast Givens method [7, Algorithm 5.2.4]. Of these, we chose to specify the Fast Givens method for this kernel benchmark. This was primarily done because the Fast Givens method consists of a number of fine grain calculations. This structure is very suitable for implementation as a *stream algorithm* on PCAs such as the MIT RAW machine. For more details, see Hoffmann [8]. The data matrix sizes that we define for this kernel, and the corresponding workloads for calculating the QR factorization, are given in Table 3.

Table 3.

QR input parameters.

Parameter Name	Description	Values		
		Set 1	Set 2	Set 3
m	Matrix rows	500	180	150
n	Matrix columns	100	60	150
W	Workload (Mflop)	397	30.5	45.0

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.

Table 4.

SVD input parameters.

Parameter Name	Description	Values		
		Set 1	Set 2	Set 3
m	Matrix rows	500	180	150
n	Matrix columns	100	60	150
W_{r1}	Fixed workload (Mflop)	101	15	72
W_{r2}	Workload per iteration (Mflop)	0.24	0.88	0.54

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

$$A = U\Sigma V^H, \quad (6)$$

where U is a unitary matrix of size $m \times m$, Σ is an $m \times n$ matrix in which the upper $n \times n$ portion is a diagonal matrix with all entries real and sorted in descending order, and V is an $n \times n$ unitary matrix. If $m > n$, then define

$$U = \begin{bmatrix} U_a & U_b \end{bmatrix},$$

$$\Sigma = \begin{bmatrix} \Sigma_a \\ 0 \end{bmatrix},$$

where U_a is size $m \times n$, U_b is size $m \times (m - n)$, and Σ_a is of size $n \times n$. Then $A = U_a \Sigma_a V^H$ is called the *reduced SVD* of the matrix A . In this context the SVD defined in equation (6) is sometimes referred to as the *full SVD* for contrast. Notice that U_a is not unitary, but it does have orthogonal columns. When $m < n$, the reduced SVD can be similarly defined by partitioning V instead of U .

For signal processing applications, we are typically most interested in the reduced decomposition, in the matrix U , and in the singular values (the values on the diagonal of Σ). We provide operation counts for the reduced decomposition, assuming that all three matrices are produced. The data matrix sizes of interest and associated operation counts are given in Table 4.

There are three major steps to the full SVD algorithm, which are described in more detail in Golub and Van Loan [7]. First, if the $m \times n$ matrix A has many more rows than columns, a QR factorization is performed. This step is done if $m > 5n/3$ [7, p. 252], which is typically the case in signal processing.

Define

$$Q = \begin{bmatrix} Q_a & Q_b \end{bmatrix}, \quad (7)$$

$$R = \begin{bmatrix} R_a \\ 0 \end{bmatrix}, \quad (8)$$

where Q_a is size $m \times n$, Q_b is size $m \times (m - n)$, and R_a is size $n \times n$. The decomposition $A = Q_a R_a$ is referred to as the *reduced QR* decomposition of A . Matrix Q_a is not unitary, but it has orthogonal columns. The reduced QR factorization can be obtained by the modified Gram-Schmidt algorithm described in Golub and Van Loan [7, Algorithm 5.2.5]. If a full SVD is being performed, the full QR is computed: if a reduced SVD is being performed, a reduced QR is computed. In the remainder of this exposition, we describe the reduced SVD algorithm for a matrix with $m \geq n$. We assume that in the first step, we perform a reduced QR decomposition via the MGS algorithm to produce

$$A = U_1 R,$$

where R is an $n \times n$ upper-triangular matrix and U_1 is an $m \times n$ matrix with orthogonal columns. (Notice that the QR factorization described in Section 3.2 is a full QR; hence, a different algorithm is used here.)

In the next step, R is reduced to *bi-diagonal* form, to consist of the main diagonal and a single diagonal of entries above that, with the remainder of the entries in the matrix being zero [7, p. 253].

This is accomplished with Householder transforms, producing

$$R = U_2 B V_2^H,$$

where U_2 and V_2 are unitary and of size $n \times n$, and the $n \times n$ matrix B is bi-diagonal. The matrix B produced at this step is no longer complex, but real, though matrices U_2 and V_2 are complex.

The final step is an iterative reduction of B to diagonal form and the ordering of the singular values. This is accomplished with Givens rotations [7, p. 454]. At the end of this step we have produced matrices $n \times n$ orthogonal matrices U_3 and V_3 such that

$$B = U_3 \Sigma V_3^H,$$

so that the singular value decomposition of the original matrix A can be expressed as

$$\begin{aligned} A &= U_1 U_2 U_3 \Sigma (V_3 V_2)^H \\ &= U \Sigma V^H, \end{aligned}$$

with $U = U_1 U_2 U_3$ and $V = V_3 V_2$.

As the exact number of iterations required to produce the SVD is based on the data, an efficiency measurement must take into account the actual number of iteration steps performed. We account for this by defining the workload W as a linear function of two numbers W_1 and W_2 given in Table 4,

$$W = W_1 + d * W_2, \tag{9}$$

where d is the number of iteration steps performed in the reduction of B to diagonal form, W_2 is an estimate of the number of floating-point operations per iteration step, and W_1 is an estimate of the number of floating-point operations in the remainder of the algorithm. The estimates W_1 and W_2 for complex matrices are given separately for the full and reduced SVD algorithms in Table 4. The number d for a given data set must be “discovered” in the course of the execution of the benchmark.

There are many implementation details associated with achieving high performance in the singular value decomposition. Examples of such details include the use of block Householder transforms [7, p. 213] and the storage of the Householder transforms and Givens rotations that produce U and V rather than the matrices themselves [3].

3.4 Constant False Alarm Rate Detection

The constant false-alarm rate (CFAR) detection benchmark is an example of data-dependent processing designed to find targets in an environment of varying background noise. The benchmark subjects a subset of a radar data cube to this algorithm.

Assume a data cube consisting of real (as opposed to complex) data whose dimensions are beams, range, and dopplers. During CFAR detection, a local noise estimate is computed from the $2N_{cfar}$ range gates near the cell $C(i, j, k)$ under test. A number of guard gates G immediately next to the cell under test will not be included in the local noise estimate (this number does not affect the throughput). For each cell $C(i, j, k)$, the value of the noise estimate $T(i, j, k)$ is calculated as

$$T(i, j, k) = \frac{1}{2N_{cfar}} \sum_{l=G+1}^{G+N_{cfar}} |C(i, j+l, k)|^2 + |C(i, j-l, k)|^2. \tag{10}$$

The range cells involved in calculating the noise estimate for a particular vector are shown in Figure 1. 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.

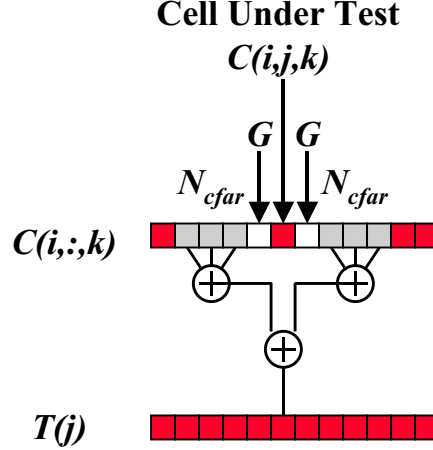


Figure 1. Sliding window in CFAR detection. The example shows the number of guard cells $G = 1$ and the number of cells used in computing the estimate $N_{cfar} = 3$.

An efficient implementation of the CFAR algorithm makes use of the redundancy in the computation of T according to the formula given in (10). Note that the relationship between $T(i, j, k)$ and $T(i, j + 1, k)$ is

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

Using this relationship, the value of T for a particular set of N_{rg} range gates can be calculated in $O(N_{rg})$ time, that is, independent of the values of G and N_{cfar} . Note that some variations of this formula and equation (10) occur at the boundary areas. For the most part, these are handled in a straightforward fashion: if a computed index would cause reference to a cell outside the cube's boundaries, we ignore that term in the computation.

The parameter sets for the CFAR benchmark are shown in Table 5.

Table 5.

Parameter sets for the CFAR Kernel Benchmark.

Name	Description	Set 0	Set 1	Set 2	Set 3	Units
Nbm	Number of beams	16	48	48	16	beams
Nrg	Number of range gates	64	3500	1909	9900	range gates
Ndop	Number of doppler bins	24	128	64	16	doppler bins
Ntgts	Number of targets that will be pseudo-randomly distributed in Radar data cube	30	30	30	30	targets
Ncfar	Number of CFAR range gates	5	10	10	20	range gates
G	CFAR guard cells	4	8	8	16	range gates
mu	Detection sensitivity factor	100	100	100	100	
W	Workload	0.17	150	41	18	Mflop

4. Communication Benchmark

Many signal and image processing applications operate on multi-dimensional data in multiple stages, with operations focusing on a different dimension in each subsequent stage. If the host platform is a parallel processor, the data are usually distributed across the nodes to exploit data parallelism, so that each node can operate in parallel on its portion of the data as the algorithm transitions from one stage to the next. For efficiency reasons, it is desirable to perform a *corner-turn* of the data. A corner turn operation is defined as a copy of the object with a change in the storage order of the underlying data. This may or may not imply transposition of the computation object, depending on the implementation. In this section, we describe this operation in more detail and describe in general terms an abstracted, high-level application that requires a corner-turn operation (already presented in an earlier document [10], some of which is repeated here for completeness).

An application that requires a corner turn works first on the *rows* of an input matrix, and then on the *columns* of the intermediate result matrix. Mathematically, one of the most basic examples of such an operation is a multiplication of three matrices,

$$Z = B^H X A, \quad (11)$$

where B and A are application-dependent matrices, X is the matrix of input data, and Z is the matrix of output data. An example situation where this might occur would be a filtering operation followed by a beamforming operation.¹ Suppose, as in the previous description [10], that we perform the operations of Equation (11) into two stages, the first producing $Y = XA$ and the second producing $Z = B^H Y$. Then the first stage is an operation in which an entire row of X is desired, and the second is an operation in which an entire column of Y is desired. Thus, the two stages suggest different optimal data layouts.

The idea behind the corner-turn operation is to preserve data locality in the dimension being operated on. Whether or not a mathematical transpose is performed is implementation-dependent. In multi-dimensional arrays in the C programming language, the last array index is continuous in memory. In order to perform a corner turn of a C language array, a transpose is required; that is, the order of some of the dimensions must be reversed (see Figure 2).

Now consider an implementation with the property that storage order is independent of the order of the indexes. In such an implementation, it would be possible to do a corner turn without requiring a transpose of the computation object. The vector, signal, and image processing library (VSIPL, [13]) is an example of a library where this is possible: the stride parameters of a VSIPL view allow the VSIPL copy operation to re-arrange the underlying data without changing the mathematical properties (see Figure 3).² When using objects with this property, storage order may be considered to be a mapping issue, whereas when using standard C and C++ arrays, storage order is explicitly embedded in the application program.

The discussion above does not consider the distribution of data over processors. Distribution effectively adds a level of memory hierarchy to the performance of a corner turn: data must be

¹A filtering operation can be represented as a matrix-matrix multiply, but would usually not be implemented in such a fashion. Thus, the use of matrix multiplication here is an additional level of abstraction about the application.

²It would also be possible to implement standard C/C++ data structures with this property: use of VSIPL here is purely a matter of convenience.

The C code to perform a corner turn of two-dimensional array A into two-dimensional array B is

```
// Notice dimensions of B are the reverse of those of A
int A[NX][NY], B[NY][NX];

for (i = 0; i < NX; i++)
    for (j = 0; j < NY; j++)
        B[j][i] = A[i][j];
```

If NX and NY were defined at compile time to be 4 and 3, respectively, and

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \end{bmatrix},$$

then the memory area underlying A is

A = { 0 1 2 3 4 5 6 7 8 9 10 11 }.

Mathematically

$$B = \begin{bmatrix} 0 & 3 & 6 & 9 \\ 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \end{bmatrix} = A^T,$$

and after execution of the above code the memory area underlying B is

B = { 0 3 6 9 1 4 7 10 2 5 8 11 }.

Figure 2. C corner turn example. In matrix B, the data are stored in corner-turned fashion compared to matrix A, and B is the transpose of A.

The C code to perform a corner turn from VSIPL matrix view A into VSIPL matrix view B is

```
// Notice dimensions of B are the same as those of A:
// storage order is different
vsip_mview_i *A = vsip_mcreate_i(NX,
                                NY,
                                VSIP_ROW,
                                VSIP_MEM_NONE);
vsip_mview_i *B = vsip_mcreate_i(NX,
                                NY,
                                VSIP_COL,
                                VSIP_MEM_NONE);

vsip_mcopy_i_i(A, B);
```

If NX and NY were defined at compile time to be 4 and 3, respectively, and

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \end{bmatrix},$$

then the block underlying A is

$$A = \{ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \}.$$

After execution of the above code, B is mathematically the same as A, and the block area underlying B is

$$B = \{ 0 \ 3 \ 6 \ 9 \ 1 \ 4 \ 7 \ 10 \ 2 \ 5 \ 8 \ 11 \}.$$

Figure 3. VSIPL corner turn example. Matrix views A and B are mathematically the same even though the underlying data in B are stored in corner-turn fashion compared to matrix A.

copied to a new processor as well as re-arranged on the new processor. Frequently, an all-to-all communication operation, in which every processor communicates with every other processor, is required as part of a distributed corner turn.

Parameters for two corner turn sizes are given in Table 6. These corner turn sizes are based on current applications: they assume 32-bit data, either integers or single-precision floating-point data. The workload is twice the overall matrix size since the data is being copied and must therefore pass into and out of the processor.

Table 6.

Corner turn parameters.

Parameter Name	Description	Values	
		Set 1	Set 2
M	Matrix rows	50	750
N	Matrix columns	5000	5000
k	Element size (bytes)	4	4
	Matrix size (Mbyte)	1	15
W	Workload (Mbyte)	2	30

For this particular kernel benchmark, the idea of timing throughput and latency based on a single processor is acceptable, but it would be preferable to get a sense of the throughput and latency possible for a multi-processor corner turn of the data. In the most frequent occurrences of a distributed corner turn in signal processing, the source and destination processor groups are either identical or are completely disjoint. The derived statistics of most interest for multi-processor transfers are the stability of the throughput over the number of processors used, and the efficiency of the transfer versus the theoretical peak bandwidth.

5. Information and Knowledge Processing Benchmarks

5.1 Pattern Matching

The pattern matching kernel is extracted from the feature-aided tracking portion of the integrated radar-tracker application [2]. Fundamentally, this step entails overlaying two length- N vectors a and t and computing a metric that quantifies the degree to which these two vectors match. In general, the vector t is chosen from a set of reference vectors referred to as the *template library*. The metric used for matching is the weighted MSE (mean square error) ϵ ,

$$\epsilon = \frac{\sum_{k=1}^N (w_k * (a_k - t_k)^2)}{\sum_{k=1}^N w_k}, \quad (12)$$

where w_k , $k = 1, 2, \dots, N$ is the vector of weights. The optimal weights for the feature-aided tracker have been computed empirically. In the kernel benchmark, we provide a generic weighting vector.

The calculation done in equation (12) is performed on data that has been converted to decibels (the “dB domain”). This is done because the raw power output from a signal processing system can vary by many orders of magnitude. However, conversion of patterns between the power domain and the dB domain is performed during the course of the benchmark: this requires the use of a number of logarithm and exponentiation functions. The operation count for these functions is implementation-dependent, and so the workload we give has three components: a count of the number of calls to the exponent function, a count of the number of calls to the logarithm function, and a count of the operations in the rest of the benchmark.

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 may need 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. A summary of this procedure is shown in Figure 4.

The parameters of interest for the pattern matching benchmark are the length of the pattern vectors, the size of the template pattern library, and the number of shift and scale operations performed. These parameters are given for two data set sizes in Table 7.

5.2 Database Operations

We consider measuring the performance of database operations in the context of a tracking application that stores track information in a database. Tracks are indexed using their location (spatial coordinates). The tracker operates in discrete time intervals called *cycles*. During each cycle, the tracker receives a set of target reports from a radar. It asks the database to search for all

```

for each of  $K$  patterns
  for each of  $S_r$  shift values
    calculate MSE value with shifted pattern
  Choose shift value with smallest MSE
  for each of  $S_m$  magnitude values
    calculate MSE value with scaled pattern
  Choose gain value with the smallest MSE

```

Figure 4. Outline of the pattern match kernel.

Table 7.

Pattern matching parameters.

Parameter Name	Description	Values	
		Set 1	Set 2
N	Pattern length	64	128
K	Number of patterns	72	256
S_r	Number of shifts	21	43
S_m	Number of magnitude scalings	21	21
W_1	Workload: \log_{10} function calls	4.61×10^3	32.8×10^3
W_2	Workload: 10^x function calls	4.61×10^3	32.8×10^3
W_3	Workload: Other floating-point ops (flops)	1.20×10^6	13.6×10^6

tracks that could be associated with each target report, based on location. The tracker may direct the database to insert new tracks based on target reports that are not associated with any tracks, and to delete specific tracks.

The database interface receives a stream of instructions from the tracker consisting of the search, insert, and delete operations to be performed. Its output is a set of record identifiers which are presumably used to look up the actual records in memory. As the actual database does not exist, the numbers are essentially random 32-bit integers. Our goal is to measure the performance of the search, insert, and delete operations, *without ever altering the contents of any particular record*. The major motivation for this is to avoid generating the large amount of data necessary for the database. A typical record from a feature-aided track application is on the order of 650 bytes per record (see document PCA-IRT-4, [2]), and test cases of interest may require up to 100,000 such records. Thus, in the benchmark, we do not actually generate and maintain the contents of the database itself, only the indexing structures. Therefore, the structures used only store the values we need: x and y coordinates, and a record identifier, which is an 32-bit integer index of or pointer to a data record.

For an application of this type, the three database operations needed are:

search Look up and retrieve all items whose characteristics fall into a given range. In this case, a search is done for all targets within a specified range of a particular (x, y) coordinate pair, where x and y are floating-point numbers. Given a set of sector bounds $\{x_{Min}, x_{Max}, y_{Min}, y_{Max}\}$,

Table 8.

Tracking parameters.

Parameter Description	Values	
	Set 1	Set 2
Cycles to run	100	100
Total existing targets ($P + U$)	500	102,400
Number of placed targets (P)	450	92,160
X total area, M (km)	5	32
Y total area, N (km)	5	32
X search area, Δx (km)	2	2
Y search area, Δy (km)	2	2
Overall target density, d (targets/km ²)	20	100
Search operations per cycle, ns	400	100
Matches found per search k	80	400
Insert operations per cycle, ni	20	300
Delete operations per cycle nd	20	300
Workload per cycle (transactions)	440	700

this search can be expressed in a fashion approximating the structured query language (SQL) as

```
select * from TrackDatabase
where ( $x > x_{Min}$  AND  $x < x_{Max}$  AND  $y > y_{Min}$  AND  $y < y_{Max}$ ).
```

insert Add a new item to the database. This can be expressed in an SQL-like fashion as
insert into TrackDatabase values(id, x_u, y_u).

delete Remove an item from the database, expressed in an SQL-like fashion as
delete from TrackDatabase where ($x = x_u$ AND $y = y_u$).

Database workloads are provided based on two scenarios: a kinematic tracking scenario similar to the parameters proposed for the integrated radar-tracker benchmark, and a multi-hypothesis tracking scenario in which the database is allowed to become much larger. For each scenario, the frequency of each operation (search, insert, delete) is specified. The parameters that define these two scenarios are given in Table 8.

5.2.1 Test Data

Test data for the database is drawn from a tracker scenario with stationary targets which will appear and disappear from the database. The targets for the scenario will be distributed roughly evenly on a grid of size $M \times N$ km². These targets will be divided into a set of placed targets, P , and a set of unplaced targets, U . Targets in set P will have corresponding records in the database (they have previously “appeared”) and targets in set U will not (they have “disappeared”). The size

of set P is defined to be sufficient that searches in an area of size $\Delta x \times \Delta y$ will expect to find k targets on average; that is, the target density is $d = k/(\Delta x \Delta y)$ targets/km². The size of set U is chosen to allow sufficient insertions and deletions with additional targets remaining to allow some measure of differences between searches. We have chosen to make P 90% of the total targets and U the remaining 10%.

The database generator creates a sequence of commands for the database. To do this, the generator must keep track of the sets P and U . For each cycle, the generator must choose ns uniformly random (x, y) pairs, nd targets from P , and ni targets from U . The ns pairs will be passed to the database, which will search the records and return those within Δx and Δy of the random pairs. The nd targets from P will be passed to the database to have their records deleted (they will “disappear”). The ni targets from U will be passed to the database and corresponding records will be inserted into the database (thus will “appear”). The time reported for each cycle is the total time to execute the search, insert, and delete commands. The command generation occurs before any of the actual benchmarking and therefore is not timed.

5.2.2 Workload

For a workload value for each scenario, we count each transaction (search, insert, delete) as an operation to be performed. This workload value, given in Table 8, can be used to compute throughput for the database kernel (in transactions per second) and compare among different architectures. However, an efficiency for the database kernel benchmark is not defined. Peak performance for the database kernel benchmark would be calculated from the rate at which the PCA chip can perform each database operation, which in turn is related to the memory hierarchy of the entire system.

5.3 Graph Optimization via Genetic Algorithm

Genetic algorithms [4, 6, 14] have become a viable solution to strategically perform a global search by means of many local searches. The basis of the genetic algorithm methods is derived from the mechanisms of evolution and natural genetics. The genetic algorithm that is being used as one of these kernel benchmarks is a fairly simple version. Many modifications are possible that can enhance the performance for a given application, and some small enhancements have been made to enhance the performance of this benchmark.

A genetic algorithm works by building a population of chromosomes which is a set of possible solutions to the optimization problem. Within a generation of a population, the chromosomes are randomly altered in hopes of creating new chromosomes that have better evaluation scores. The next generation population of chromosomes is randomly selected from the current generation with selection probability based on the evaluation score of each chromosome. The simple genetic algorithm follows the structure depicted in Figure 5. Each of these operations will be described in the following subsections.

5.3.1 Initialization

Initialization involves setting the parameters for the algorithm, creating the scores for the simulation, and creating the first generation of chromosomes. In this benchmark, seven parameters are


```

Simple Genetic Algorithm ()
{
  Initialization;
  Evaluation;
  while termination criterion has not been reached
  {
    Selection_and_Reproduction;
    Crossover;
    Mutation;
    Evaluation;
  }
}

```

Figure 5. Structure of a simple genetic algorithm.

set:

- the genes value (`Genes`) is the number of variable slots on a chromosome;
- the codes value (`Codes`) is the number of possible values for each gene;
- the population size (`PopSize`) is the number of chromosomes in each generation;
- crossover probability (`CrossoverProb`) is the probability that a pair of chromosomes will be crossed;
- mutation probability (`MutationProb`) is the probability that a gene on a chromosome will be mutated randomly;
- the maximum number of generations (`MaxGenerations`) is a termination criterion which sets the maximum number of chromosome populations that will be generated before the top scoring chromosome will be returned as the search answer; and
- the generations with no change in highest-scoring (elite) chromosome (`GensNoChange`) is the second termination criterion which is the number of generations that may pass with no change in the elite chromosome before that elite chromosome will be returned as the search answer.

The scores matrix for the simulation, which is generated in the `GenAlgGen` script, is the set of scores for which the best solution is to be found. The attempted optimization is to find the code for each gene in the solution chromosome that maximizes the average score for the chromosome. Finally, the first generation of chromosomes are generated randomly.

5.3.2 Evaluation

Each of the chromosomes in a generation must be evaluated for the selection process. This is accomplished by looking up the score of each gene in the chromosome, adding the scores up, and

averaging the score for the chromosome. As part of the evaluation process, the elite chromosome of the generation is determined.

5.3.3 Selection and Reproduction

Chromosomes for the next generation are selected using the roulette wheel selection scheme [14] to implement proportionate random selection. Each chromosome has a probability of being chosen equal to its score divided by the sum of the scores of all of the generation's chromosomes. In order to avoid losing ground in finding the highest-scoring chromosome, elitism [14] has been implemented in this benchmark. Elitism reserves two slots in the next generation for the highest scoring chromosome of the current generation, without allowing that chromosome to be crossed over in the next generation. In one of those slots, the elite chromosome will also not be subject to mutation in the next generation.

5.3.4 Crossover

In the crossover phase, all of the chromosomes (except for the elite chromosome) are paired up, and with a probability `CrossoverProb`, they are crossed over. The crossover is accomplished by randomly choosing a site along the length of the chromosome, and exchanging the genes of the two chromosomes for each gene past this crossover site.

5.3.5 Mutation

After the crossover, for each of the genes of the chromosomes (except for the elite chromosome), the gene will be mutated to any one of the codes with a probability of `MutationProb`. With the crossover and mutations completed, the chromosomes are once again evaluated for another round of selection and reproduction.

5.3.6 Termination

The loop of chromosome generations is terminated when certain conditions are met. When the termination criteria are met, the elite chromosome is returned as the best solution found so far. For this benchmark, there are two criteria: if the number of generation has reached a maximum number, `MaxGenerations`, or if the elite solution has not changed for a certain number of generations, `GensNoChange`.

5.3.7 Implementation Notes

Genetic algorithms are being used around the world for an enormous variety of applications. However, this benchmark of genetic algorithms has been designed with two specific purposes in mind: matching computational tasks with processing units in a general independent task environment and in signal processing pipeline tasks. For the general independent task environment, the genes of the chromosomes are the computational tasks which have arrived in the order of their gene's number, and the codes are the possible processing units upon which the computational tasks

Table 9.

Parameter sets for the Genetic Algorithm Kernel Benchmark.

Name	Description	Set 1	Set 2	Set 3	Set 4	Units
Codes	Number of code types for a gene	4	8	100	1000	codes
Genes	Number of genes on a chromosome	8	96	5	10	genes
PopSize	Number of chromosomes in a generation	50	200	100	400	chromosomes
CrossoverProb	Probability of crossing over a pair of chromosomes	0.01	0.002	0.02	0.03	
MutationProb	Probability of mutating a chromosome	0.60	0.60	0.60	0.30	
MaxGenerations	Maximum number of generations	500	2000	500	5000	generations
GensNoChange	Maximum number of generation with no change in elite chromosome	50	150	50	500	generations
	Ops per generation	1750	77400	2300	17200	operations
	Random numbers per generation	898	38798	1198	8798	numbers
	Ops for random number gen.	9878	426778	13178	96778	operations
	Total Ops	11628	504178	15478	113978	operations

can be executed. Similarly, for the signal processing pipeline tasks, the genes of the chromosomes are the pipelined tasks that constitute a signal processing chain, and the codes are the computational unit mappings upon which the pipeline stage tasks can be run. The scores for each code in a given gene position represents a goodness factor (for computational efficiency, execution time, or some other measure) ranging from zero to one (one being best). The goal then is to find the mapping of tasks onto processors that yields the best score, in this case a perfect score of one.

The random number generator for the genetic algorithm is assumed to be the one defined in VSIPL [13, p.245]. This generator is used because the implementation is available and the workload is easily calculable (based on the discussion in the standard, we assume 11 ops per random number generated). Use of this random number generator is not required. However, if a different random number generator is used, the workload given in Table 9 should be altered accordingly.

For this kernel benchmark, four parameter sets have been included. These sets are shown in Table 9. Sets 1 and 2 are sample parameters for the general independent task scenario while sets 3 and 4 are sample parameters for the digital signal processing pipeline task scenario. For this kernel, we define the workload in operations (rather than floating-point operations) per generation. The workload is related to the number of genes and the population size per generation.

Parts of the genetic algorithm code are embarrassingly parallel, including the crossover and mutation sections. Most of the evaluation section is also embarrassingly parallel, except for the elite chromosome determination portion. However, the selection and reproduction section cannot be conducted in a parallel manner since a view of the entire population is necessary. More discussion on parallelizing and distributing genetic algorithms can be found in [5].

6. Further Kernel Benchmarks

These are benchmarks that might be considered for a second set of kernel benchmarks, but are not included with this first set.

Image encoding. Compress a synthetic aperture radar (SAR) image for storage or transmission. The dynamic range of SAR imagery is such that it is not well represented using conventional image processing standards such as that defined by the joint photographic experts group (JPEG) for image compression. The JPEG 2000 standard addresses the problems that JPEG presents for SAR images. For more details on JPEG 2000, see Taubman and Marcellin [15]. Baxter and Seibert have performed an analysis of the desirable features of an encoding algorithm for SAR images [1]. The major features of the encoding algorithm as they describe it are:

- wavelet packet transforms with a Gabor-like tree structure and smooth biorthogonal wavelet filters,
- trellis-coded quantization, and
- a bit-allocation procedure based on minimizing distortion (perceptual distortion, based on the human visual system) and rate.

Some of these features (particularly trellis-coded quantization) are in “part 2” of the JPEG 2000 standard, and are therefore not yet present in most publicly available implementations of the standard.

Secure network protocol. Transmit a message of a given size with authentication and encryption, based on IPSec or a similar protocol.

Synchronization. Mark a value on a different processor or in a remote memory for exclusive access (lock/unlock operations).

Image processing: morphological operations. Take an image and perform an “opening” or “closing” operation on it. These are integer convolution operations.

Image processing: edge detection. Perform edge detection in both the x and y dimensions of a two-dimensional image.

Giga-updates per second. Read, then write a sequence of random memory locations in a large memory space. A description of the benchmark is at

`<http://iram.cs.berkeley.edu/~brg/dis/gups/>.`

Incomplete Gamma function. Calculate values of the incomplete gamma function.

REFERENCES

1. Robert Baxter and Michael Seibert. Synthetic aperture radar image coding. *MIT Lincoln Laboratory Journal*, 11(2):121–158, 1998.
2. W. Coate and M. Arakawa. Preliminary design review: Feature-aided tracking for the PCA integrated radar-tracker application. Project Report PCA-IRT-5, MIT Lincoln Laboratory, Lexington, MA, October 2004.
3. J. J. M. Cuppen. The singular value decomposition in product form. *SIAM J. Sci. Stat. Comput.*, 4(2):216–222, June 1983.
4. Lawrence Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.
5. José L. Ribeiro Filho, Philip C. Treleaven, and Cesare Alippi. Genetic-algorithm programming environments. *IEEE Computer*, 27(6):28–43, June 1994.
6. David E. Goldberg, editor. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Van Nostrand Reinhold, New York, 1991.
7. Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, 1996.
8. Henry Hoffmann. Stream Algorithms and Architecture. Master’s thesis, Massachusetts Institute of Technology, Cambridge, MA, 2003.
9. David J. Kuck. *High Performance Computing: Challenges for Future Systems*. Oxford University Press, New York, NY, 1996.
10. James M. Lebak. Polymorphous computing architectures (PCA) example application 4: Corner-turn. External report, MIT Lincoln Laboratory, Lexington, MA, October 2001.
11. James M. Lebak. Preliminary design review: PCA integrated radar-tracker application. Project Report PCA-IRT-1, MIT Lincoln Laboratory, Lexington, MA, April 2002.
12. Alan V. Oppenheim and Ronald W. Schaffer. *Discrete-time signal processing*. Prentice-Hall, Inc., 1989.
13. 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>.
14. M. Srinivas and Lalit M. Patnaik. Genetic algorithms: A survey. *IEEE Computer*, 27(6):17–26, June 1994.
15. David S. Taubman and Michael W. Marcellin. *JPEG 2000: Image Compression Fundamentals, Standards, and Practice*. Kluwer Academic Publishers, 2002.

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

APPENDIX A

Revisions

This is the first public revision of the document, dated 13 June 2005. It introduces the following changes from the original public version:

- references to kernel benchmark code and its availability dates were deleted;
- the database kernel benchmark description was revised to remove implementation details;
- the CFAR workload and description were revised to reflect *real* (as opposed to complex) data;
- an outline of the calculations performed in the pattern match kernel was added, and some errors in the operation count were corrected;
- the QR factorization section was added;
- the SVD section was revised to reflect a reduced SVD;
- the corner turn workload was revised to clarify the data size and why the workload is twice the matrix data size;
- the FIR workload was slightly changed and some notes on the operation counts were added; and
- the “Revisions” chapter that you are reading now was re-ordered so that the most recent changes were at the start.

The first public version of the document was issued 23 January 2004. It introduced the following changes from the program-private version:

- an error in the CFAR workload table (Table 5) was corrected;
- a description of the reduced SVD algorithm, including workload estimates, was added;
- the description of the database kernel benchmark was heavily revised and the workload was changed;
- a description of the random number generator used for the genetic algorithm was added and its workload was updated in a corresponding way;
- the pattern match benchmark description and workload were updated to reflect a more efficient implementation and to introduce a second data set; and
- the discussion of metrics (in particular, of efficiency and program stability) was updated to reflect that efficiency is not always easily calculable for the different kernels.

The original version of this document was distributed to PCA program participants (only) at sponsor direction, July 31, 2002.

REPORT DOCUMENTATION PAGE				<i>Form Approved</i> 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) 13 June 2005		2. REPORT TYPE Project Report		3. DATES COVERED (From - To)	
4. TITLE AND SUBTITLE Polymorphous Computing Architecture (PCA) Kernel-Level Benchmarks				5a. CONTRACT NUMBER FA8721-05-C-0002	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) J. Lebak, A. Reuther, E. 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-1, Revision 1	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) DARPA/ITO 3701 Fairfax Drive Arlington, VA 22203-1714				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S) ESC-TR-2005-067	
12. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT This document describes a series of kernel benchmarks for the PCA program. Each kernel benchmark is an operation of importance to DoD sensor applications making use of a PCA architecture. Many of these operations are a part of the composite example application described elsewhere. The kernel-level benchmarks have been chosen to stress both computation and communication aspects of the architecture. "Computation" aspects include floating-point and integer performance, as well as the memory hierarchy, while the "communication" aspects include the network, the memory hierarchy, and the I/O capabilities. 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 processing, communication, and information knowledge processing. The specification of the benchmarks in this document is meant to be high-level and largely independent of the implementation.					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT Same as Report	18. NUMBER OF PAGES 39	19a. NAME OF RESPONSIBLE PERSON
a. REPORT Unclassified	b. ABSTRACT Same as Report	c. THIS PAGE Same as Report			19b. TELEPHONE NUMBER (include area code)

