# A Performance Analysis of PIM, Stream Processing, and Tiled Processing on Memory-Intensive Signal Processing Kernels

Jinwoo Suh, Eun-Gyu Kim, Stephen P. Crago, Lakshmi Srinivasan, and Matthew C. French
University of Southern California/Information Sciences Institute
3811 N. Fairfax Drive, Suite 200, Arlington, VA 22203
{jsuh, eungyu, crago, lakshmi, mfrench}@isi.edu

## Abstract

*Trends in microprocessors of increasing die size and clock speed and decreasing feature sizes have fueled rapidly increasing performance. However, the limited improvements in DRAM latency and bandwidth and diminishing returns of increasing superscalar ILP and cache sizes have led to the proposal of new microprocessor architectures that implement processor-in-memory, stream processing, and tiled processing. Each architecture is typically evaluated separately and compared to a baseline architecture. In this paper, we evaluate the performance of processors that implement these architectures on a common set of signal processing kernels.*

*The implementation results are compared with the measured performance of a conventional system based on the PowerPC with Altivec. The results show that these new processors show significant improvements over conventional systems and that each architecture has its own strengths and weaknesses.*

## 1. Introduction

Microprocessor performance has been doubling every 18-24 months for many years [7]. This increase has been possible because die size has increased and feature size has decreased. However, the increasing die size combined with fast clock speeds have made the maximum distance as measured in clock cycles between two points on a processor longer.

To solve this problem, pipelining has been used widely. However, increasing pipeline depth increases various latencies, including cache access and branch prediction penalties, and increases the complexity of processor design. Techniques for exploiting ILP without exposing parallelism to the instruction set have also reached a point of diminishing returns.

Another problem in the recent processors is the growing gap between the processor speed and memory speed. The performance improvement of microprocessors has not been matched by DRAM (main memory) latencies, which have only improved by 7% per year [7], or pin bandwidths. These growing gaps have created a problem for data-intensive applications.

To bridge these growing gaps, many methods have been proposed such as caching, prefetching, and multithreading. However, these methods provide limited performance improvement and can even hinder performance for data-intensive applications. Caching has been the most popular memory latency tolerating technique [10][12]. Caching increases performance by utilizing temporal and spatial locality, but it is not useful for many data-intensive applications since many of them do not show such locality [11].

Recently, several architectural approaches have been explored that promise to hide memory latency for applications that include data-intensive applications while improving scalability. This study is an attempt to demonstrate and compare some of the advantages and disadvantages of processor-in-memory (PIM), streaming, and tiled architectures approaches by implementing a common set of memory-intensive signal processing kernels.

We implemented the corner turn, beam steering, and coherent side-lobe canceller (CSLC) kernels and measured the performance using cycle accurate simulators developed by each architecture group.

The rest of the paper is organized as follows. In Chapter 2, a PIM, a stream processor, and a tile-based processor are briefly described. Chapter 3 describes the three kernels we implemented: the corner turn, coherent side-lobe canceller, and beam steering. Also, the techniques that we used to exploit each platform are described. In Chapter 4, the implementation results and analysis are shown. Chapter 5 concludes the paper.

## 2. VIRAM, IMAGINE, and RAW

In this section, the VIRAM, Imagine, and Raw chips are briefly described. We also describe the performance models that will be used to understand performance of the application kernels.

### 2.1 VIRAM

In conventional systems, the CPU and memory are implemented on different chips. Thus, the bandwidth between CPU and memory is limited since the data must be transferred through chip I/O pins and copper wires on a PCB. Furthermore, much of the internal structure of DRAM, which could be exploited if exposed, is hidden because of the bandwidth limitation imposed by the pins.

Processor-In-Memory (PIM) technology is a method for closing the gap between memory speed and processor speed for data intensive applications. PIM technology integrates a processor and DRAM on the same chip. The integration of memory and processor on the same chip has the potential to decrease memory latency and increase the bandwidth between the processor and memory. PIM technology also has the potential to decrease other important system parameters such as power consumption, cost, and area.

The VIRAM chip [5] is a PIM research prototype being developed at the University of California at Berkeley. A simplified architecture of the chip is shown in Figure 1. The VIRAM contains two vector-processing units in addition to a scalar-processing unit. These units are pipelined. The vector functional units can be partitioned into several smaller units, depending on the arithmetic precision required. For example, a vector functional unit can be partitioned into 4 units for 64-bit operations or 8 units for 32-bit operations. Some operations are allowed to execute on ALU0 only. It has 8K vector register file (32 registers).

It has 13 Mbytes of DRAM. There is a 256-bit data path between the processing units and DRAM. The DRAM is partitioned into two wings, each of which has four banks. It can access eight sequential 32-bit data elements per clock cycle. However, since there are four address generators, it can access only four strided 32-bit or 64-bit data elements per cycle.

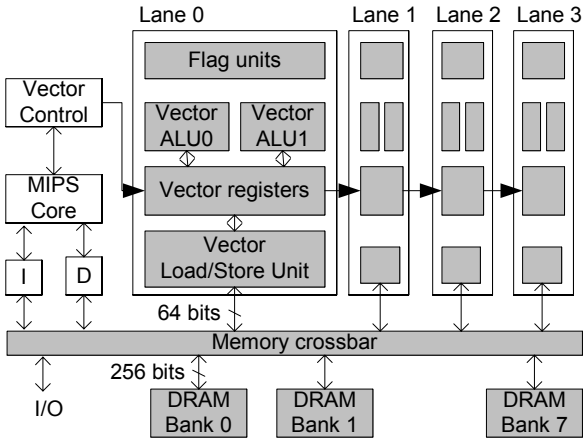


**Figure 1. Block diagram of VIRAM**

There is a crossbar switch between the DRAM and the vector processor. The target processor speed is 200 MHz, which would provide a peak performance of 3.2 GOPS (= 200 MHz x 2 ALUs x 8 data per clock) for 32-bit data. If 16-bit data is processed, the performance is 6.4 GOPS. Its peak floating point performance is 1.6 GFLOPS for 32-bit data. The power consumption is expected to be about 2 W. The EEMBC (Embedded Microprocessor Benchmark Consortium) benchmarks have been implemented on VIRAM [17]. VIRAM's performance is 20 times better (as measured by geometric mean normalized by clock frequency) than the K6-III+ x86 processor.

### 2.2 Imagine

Another approach for handling the growing processor-memory gap is stream processing. In this approach, the data is routed through stream registers to hide memory latency, allow the re-ordering of DRAM accesses, and minimize the number of memory accesses. The Imagine chip [4][11] is a research prototype stream processor developed at Stanford University. It contains eight clusters of arithmetic units that process data from a stream register file. The processor speed is currently 300 MHz, which provides a peak performance of over 14 GOPS (32-bit integer or floating-point operations). Performance results for Imagine have been presented for application kernels such as MPEG, and QRD [9]. ALU utilization between 84% and 95% is reported for streaming media applications.

Figure 2 shows the block diagram of Imagine. The stream processing is implemented with eight ALU clusters (with 6 ALUs each) with a large stream register file (SRF), and a high-bandwidth interconnect between them. The size of SRF is 128 Kbytes. A stream can start at the start of any SRF 128-byte blocks. Data is transferred to and from the SRF from off-chip memory

or the network interface. The eight ALU clusters operate on data from the SRF. Up to eight input or output streams can be processed simultaneously. The data is sent to clusters in round-robin fashion, i.e., the *i*-th data is sent to cluster (*i* mod 8). All clusters perform the same operations on their data in SIMD style. Each cluster has 6 arithmetic units (three adders, two multipliers, and one divider) and one communication interface that is used to send data between ALU clusters.

The Imagine prototype implementation has two memory controllers, each of which can process a memory access stream. The memory controller reorders accesses to reduce bank conflicts and to increase data access locality. The processor speed in the lab is currently near 300 MHz, which would provide a peak performance of 14.4 GFLOPS (= 300 MHz x 8 ALU clusters x 6 arithmetic units per cluster) for 32-bit data.

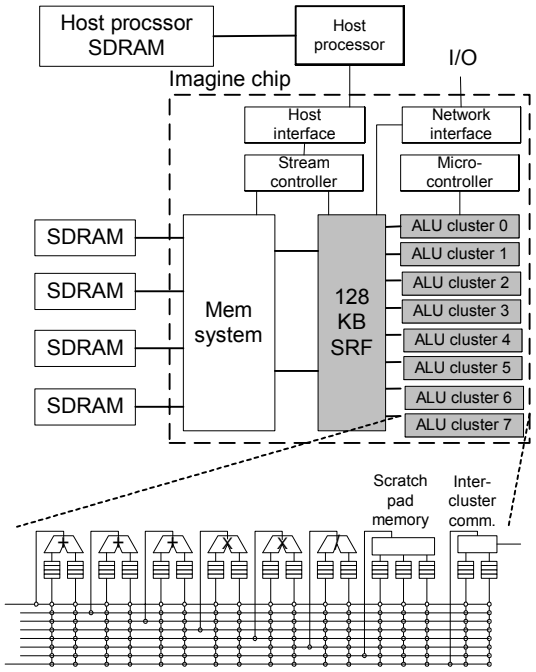


**Figure 2. Block diagram of Imagine**

## 2.3 Raw

Another approach for a scalable microprocessor that addresses issues of continued technology scaling is tile processing. Instead of building one processor on a chip, several processors (tiles) are implemented and connected in a mesh topology using a scalar operand network [16]. Then, each tile occupies a fraction of the chip space, so it is easier to make a faster processor since the signals need to travel only a short distance. One example is the Raw chip implemented at MIT [15] and shown in Figure 3. The current Raw implementation contains 16 tiles on a chip connected by a very low latency 2-D mesh network. The Raw prototype has been tested up to 199MHz and is expected to operate at 300MHz. Peak performance is 4.8 GOPS.

Each tile has a MIPS-based RISC processor with floating-point units and a total of 128 KB of SRAM, which includes switch instruction memory, tile (processor) instruction memory, and data memory. Raw uses general parallelism, which includes streaming, ILP, and data parallelism.

The Raw has four networks: two static networks and two dynamic networks. Communication on the static networks is performed by a switch processor in each tile [15]. The switch processor is located between the computation processor and the network and provides throughput to the tile processor of one word per cycle with a latency of three cycles between nearest neighbor tiles. One additional cycle of latency is added for each hop in the mesh through the static networks. When the dynamic network is used, data is sent to another tile in a packet. A packet contains header and data. If the data is smaller than a packet, dummy data is added to make a packet. If the data is larger than the packet, multiple packets are sent. The memory ports are located at the 16 peripheral ports of the chip. All tiles can access memory either through the dynamic network or through the static network.

Several kernels including matrix multiplication are implemented on Raw and the results are reported in [16]. The results show that Raw obtains speedup of up to 12 relative to single-tile performance on ILP benchmarks. Speedups greater than 16 can be achieved on streaming benchmarks when compared to a single-issue load/store RISC architecture because of a tile's ability to operate on data directly from the networks.

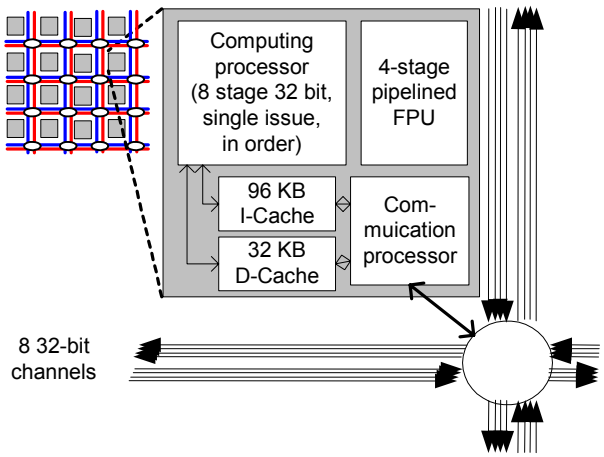


**Figure 3. Block diagram of Raw**

## 2.4 Programming methodologies

The programming methodologies and tools for each of these architectures are evolving. However, each architecture has inherent properties that affect the programming model and programmability of the architecture.

The VIRAM's programming model is that of a traditional vector architecture. An application is described as single instruction stream that contains scalar and vector instructions. There are two primary difficulties to programming the VIRAM architecture. First, the C programming language makes automatic parallelization of many loops difficult or impossible without making assumptions about the independence of pointer and array accesses. Simple loops or computations marked by user hints can be vectorized, but kernels with complex access patterns (e.g. FFT) are still difficult to automatically vectorize. Languages that are more restricted will facilitate automatic vectorization. The second factor that complicates the programmability of VIRAM is the impact of the DRAM organization on performance. Much of the performance of VIRAM is achieved by exploiting properties of DRAM organization (e.g. banks, rows, columns, and wings). Currently, the user must understand the DRAM organization to optimize performance. However, it is feasible that a compiler could organize memory references based on memory organization while it is vectorizing, especially given a language that makes this analysis feasible. For this study, a C compiler was used to compile the kernels, and then inner loops were hand-vectorized using assembly code.

The programming model of Imagine has two significant characteristics. First, the programming model is based on streams. Streams are similar to vectors, but streams can be explicitly routed between the stream register files and the ALU clusters without going through the memory system. This property is important for reducing the impact of the bandwidth bottleneck between DRAM and the processor chip. The second significant characteristic of the Imagine programming model is that a program is described in two languages, one for the host (or control) thread written in C and one for the stream processing unit written in kernel-C. Again, new programming languages may allow this distinction to be hidden from the programmer. However, the programming model used in this paper forces the programmer to think explicitly about streams and their control. This explicit streaming model has the disadvantage that a programmer must think about the application in a new way, but has the advantage that the programmer is forced to think about issues that are important to performance anyway. Applications must contain SIMD parallelism to see significant performance improvements on the Imagine architecture. For this study, inner loops were carefully scheduled to maximize performance.

The Raw architecture is the most flexible of the three architectures addressed in this paper. The tile-based organization with the low-latency, high-bandwidth network and memory interface supports a variety of programming models. The primary programming models used in the kernels described in this paper are the MIMD and stream models. The CSLC and beam steering kernels have plenty of independent parallelism to allow each tile to execute independently. We report results on two modes of using Raw: an easy-to-program but less efficient MIMD mode, in which data is routed to local memories through cache misses (CSLC), and a stream mode, in which data is routed in a stream mode without going through local memories by thinking explicitly about data placement and streams (beam steering).

However, the low-latency, high-bandwidth networks of Raw also allow ILP to be mapped efficiently to Raw. Raw's peak performance can be achieved when data can be operated on without going through local memories in the tiles. For this study, we used standard C to program the kernels. Assembly code was inserted only where necessary to access streaming data through the network. In beam steering, the codes between two instructions accessing streaming data through network are also written in assembly language. Other programming models, such as decoupled processing, are being developed for Raw and have the potential to improve performance of applications such as those described in this paper.

## 2.5 Performance models

In this section, simple performance models used to estimate the upper bound of the performance of the kernels on each architecture are described. We model computation and memory bandwidth. Memory latency is not modeled since these architectures can generally hide memory latency on the kernels used in this study.

Table 1 shows the DRAM memory and ALU throughput for 32-bit data elements that each architecture can support. It should be noted that both memory and ALU throughput are functions of these particular implementations and are not functions of the architectures themselves. However, the architectures provide the means to exploit the throughput supported by the implementation. It should also be noted that memory bandwidth reported is for the nearest DRAM. For VIRAM, DRAM is on-chip, while the nearest DRAM is off-chip for Imagine and Raw.

**Table 1. Peak throughput (32-bit words per cycle)**

| | VIRAM | Imagine | Raw |
|---|---|---|---|
| On-chip DRAM Read/Write | 8 | 16 (SRF) | 16 (Cache) |
| Off-chip DRAM Read/Write | 2 (Using DMA) | 2 | 28 |
| Computation | 8 | 48 | 16 |

## 3. Kernel Implementations

In this section, three data-intensive kernels are described. Also, the techniques used to improve the performance on VIRAM, Imagine, and Raw are presented. The descriptions of the techniques are brief due to the space limitation.

### 3.1 Corner turn

The corner turn is a matrix transpose operation that tests memory bandwidth. The data in the source matrix is transposed and stored in the destination matrix. The matrix size used for this paper, which was chosen to be larger than Imagine's SRF (128 KB) and Raw's internal memories (2 MB), but smaller than VIRAM's on-chip memory (13 MB), is 1024 x 1024 with 4-byte elements.

Naive implementations of the corner turn can have poor performance because cache performance can be bad and strided data accesses degrade DRAM bandwidth. In conventional cache-based processor systems, tiling is used to reduce cache misses.

Our VRAM corner turn uses a blocking algorithm with a 16 x 16 element matrix. Blocking allows the vector registers to be used for temporary storage between the loads and stores. We used strided load operations with padding added to the matrix rows to avoid DRAM bank conflicts. Initial load latencies are not hidden. Stores are done sequentially from the vector registers to the memory.

On the Imagine processor, we divide the matrix into multi-row strips that allows us to use the stream register files. We use four input streams and one output stream simultaneously. Since the rows within a stream are read sequentially, we maximize memory bandwidth during the reading. The Imagine clusters are used to route data in the correct output order. The output data is transferred to memory in one stream. The output data is partitioned into 128 eight-word blocks. The eight words in a block are written sequentially, but the blocks are written with a non-unit stride.

Our corner turn on Raw uses one load and one store operation for each DRAM-to-DRAM transfer. The algorithm, designed at MIT and implemented at USC/ISI, was developed to ensure that all 16 Raw tiles are doing a load or store during as many cycles as possible and to avoid bottlenecks in the static networks and data ports. The algorithm operates on 64x64 word blocks that fit in a single local tile memory. Main memory operations are all done sequentially to maximize memory bandwidth since the transpose can be done in local memories, where all accesses are done in a single cycle.

### 3.2 Coherent side-lobe canceller (CSLC)

CSLC is a radar signal processing kernel used to cancel jammer signals caused by one or more jammers. Our CSLC implementation consists of FFTs, a weight application (multiplication) stage, and IFFTs. Most of the computation time is spent on the FFT and IFFT operations.

There are four input channels: two main channels and two auxiliary channels. Each channel has 8K samples per processing interval. All computations are done using single-precision floating-point operations. The data is partitioned into 73 overlapping sub-bands, each of which contains 128 samples, so 128-sample FFTs are used.

Since the majority of computation time on the CSLC is spent on the FFT operation, we improved the performance of the FFT by using the appropriate FFT algorithms for each architecture. In this study, a parallelized hand-optimized radix-4 FFT is used for VIRAM and Imagine. Note that since the size of the FFT for the CSLC is 128, which is not power of four, we used three radix-4 stages and one radix-2 stage. We did not hand-optimize our Raw FFT implementation. Rather, a C implementation of the radix-2 FFT is used for Raw because it provided better performance than the radix-4 FFT because of register spilling in the radix-4 FFT. The Raw implementation does independent data-parallel FFTs.

### 3.3 Beam steering

Beam steering is a radar-processing kernel that directs a phased-array radar without physically rotating the antenna. The computation of the phase for each antenna element stresses memory bandwidth and latency because large tables are used for calibration tables. Arithmetic operations are additions and shift operations.

In our implementation, the following parameters are used. The number of antenna elements is 1608. Each element can direct the signal up to 4 directions per dwell where a dwell is a period. The phase needs to be calculated for each direction using calibration data.

As for other kernels, we used hand-vectorization of the main portion of the beam steering on VIRAM. Since the same processing is performed for each data, the data is fed to the vector unit, which computes output data.

For the Imagine, a manually optimized kernel was written to maximize cluster ALU utilization. The input data streams are loaded into the stream register file and supplied to the clusters. The results are written back to memory through the register file.

The beam steering processing on each data is independent. Thus, on Raw, we partition the data among 16 tiles and each tile processes its own data. Input data is streamed through the static network and is operated on directly from the network.

## 4. Experimental results and analysis

### 4.1 Overview

In this section, the implementation results are presented. Performance of these kernels is obtained by using cycle-accurate simulators provided by the VIRAM, Imagine, and Raw teams.

For comparison purposes, actual measurements of performance were taken using a single node of a 1 GHz PowerPC G4-based system (Apple PowerMac G4) [1]. An implementation using AltiVec technology was used for speedup comparison. The Apple cc compiler was used with timing done using the MacOS X system call mach_absolute_time(). We manually inserted Altivec vector instructions.

Table 2 summarizes key parameters of each processor. Note that the PowerPC is a highly optimized chip in performance implemented with custom logic. However, other processors are research chips implemented using standard cells and very small design teams. Thus, if the same level of design effort were applied to these research architectures, we would expect much higher clock rates and density to be achieved.

In Table 3, a summary of the implementation results is shown. Figure 8 shows the speedup in terms of cycles and Figure 9 shows the speedup in terms of execution time. Note that Figure 8 and Figure 9 are both using a log scale on the vertical axis.

**Table 2. Processor Parameters**

| | PPC G4 | VIRAM | Imagine | Raw |
|---|---|---|---|---|
| Clock (MHz) | 1000 | 200 | 300 | 300 |
| # of ALUs | 4 | 16 | 48 | 16 |
| Peak GFLOPS | 5 | 3.2 | 14.4 | 4.64 |

**Table 3. Experimental results** (cycles in $10^3$)

| | Corner Turn | CSLC | Beam Steering |
|---|---|---|---|
| PPC | 34,250 | 29,013 | 730 |
| Altivec | 29,288 | 4,931 | 364 |
| VIRAM | 554 | 424 | 35 |
| Imagine | 1,439 | 196 | 87 |
| Raw | 146 | 357 | 19 |

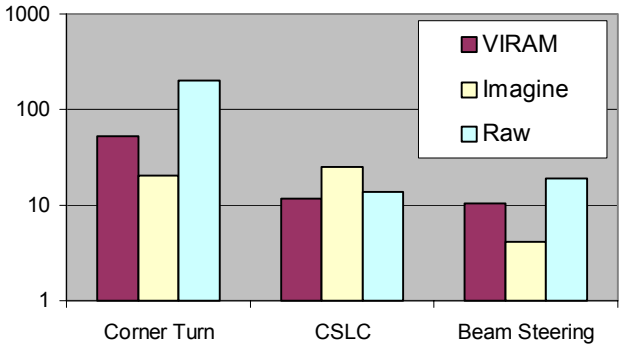


**Figure 8. Speedup compared with PPC with AltiVec (Cycles)**

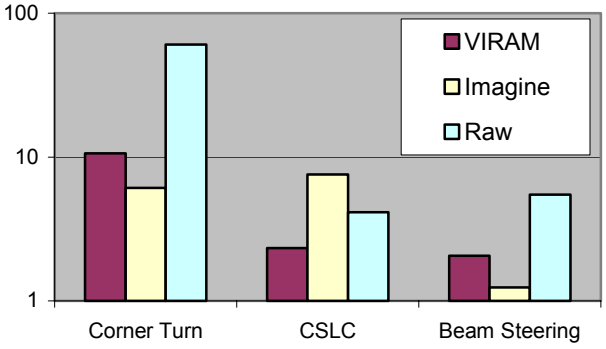


**Figure 9. Speedup compared with PPC with AltiVec (execution times when PPC=1 GHz, VIRAM=200 MHz, Imagine=300 MHz, and Raw=300 MHz)**

## 4.2 Corner turn

Table 4 summarizes the expected execution time using the performance model shown in Section 3. All three architectures provided speedups of more than 20 compared with a PowerPC system in terms of number of cycles. Corner turn performance is mostly a measure of memory bandwidth, which is not a direct property of an architecture, but rather a function of the number of pins in the package. However, the corner turn does demonstrate an architecture's ability to leverage memory bandwidth that does exist. Since VIRAM has on-chip DRAM, the kernel measures on-chip bandwidth. On the Imagine and Raw architectures, we're stressing off-chip memory.

The performance of corner turn on VIRAM is about half of what would have been expected from peak memory bandwidth. About 21% of the total cycles are overhead due to DRAM pre-charge cycles (which would be mostly hidden with sequential accesses) and TLB misses, and 24% are due to a limitation in strided load performance imposed by the number of address generators.

On Imagine, we assume the memory clock is the same frequency as the processor clock. Imagine has two address generators that provide two words per clock cycle. Note that the number of address generators is a processor implementation choice and is not a limitation of the stream architecture. Since the goal of the Imagine project was to demonstrate how memory traffic could be reduced, the Imagine team chose not to implement a high-bandwidth memory interface.

If network port were used to transfer data between SRF and an external memory connected to network port for corner turn, the performance would be the same since the network port has peak performance of two words per cycle.

87% of the cycles in the Imagine corner turn are due to memory transfers. The remaining 13% of the execution cycles are due to unoverlapped cluster instructions. Conceptually, the kernel instructions should be fully overlapped with memory accesses, but a limitation induced by the stream descriptor registers prevented full software pipelining in our implementation.

The Raw chip implementation actually provides enough main memory bandwidth that it is not the performance limiter for our corner turn implementation. Load/store issue rates and local memory bandwidth limit performance. 16 instructions per cycle are executed on the Raw tiles, and the static network and DRAM ports are not a bottleneck. The performance we achieved is nearly identical to the maximum performance predicted by the instruction issue rate. Memory latency is fully hidden (except for negligible start-up costs).

## 4.3 CSLC

CSLC mainly consists of FFTs and matrix-vector multiplication. Since the FFT length is 128, the working set fits into local memory, the performance of the CSLC depends primarily on ALU performance for Imagine and Raw.

Our IRAM CLSC analysis takes about 3.6 times longer than what is predicted by peak performance. The first factor reducing performance is overhead instructions. Instructions are needed to perform the FFT shuffles and increase the number of cycles by a factor of 1.67. The second factor that reduces FFT performance is ALU utilization. Since the second vector arithmetic unit in VIRAM cannot execute vector floating point instructions, performance on the FFT is reduced by a factor of 1.52. Finally, memory latency and vector startup costs increase performance by a factor of 1.41.

Imagine has the best performance of the three architectures on CSLC. This is because it is a computation-intensive kernel for which the working sets fit in the stream register files. Although the data access patterns for FFT are challenging for any architecture, the streaming execution model of Imagine is able to reduce memory operations and Imagine functions as intended on this kernel. Overall, performance achieved on CSLC on Imagine is about 20% of what is predicted by peak performance. While this is much lower than has been achieved for many media benchmark kernels, it still allows Imagine to perform about 10 useful operations per cycle; much better than can be achieved on today's superscalar architectures. Performance is reduced by 30% because inter-cluster communication is used to perform parallel FFTs. An alternative implementation, which was not completed for this study, would execute independent FFTs in parallel to eliminate inter-cluster communication overhead.

For the FFT kernel, ALU utilization (as measured by minimum FFT computations / total ALU cycles available) is 25.5%. If we exclude the divider, which is not useful for the FFT, then the utilization is 30.6%. Note that the utilization for the 128-point FFT is a little lower than the more than 40% obtained in other processing intensive applications [6]. The reason for the relatively low utilization is that the small size of the FFT reduces the amount of software pipelining and increases start-up overheads.

On Raw, we implemented a data parallel version of CSLC. The local memory on Raw successfully caches the working sets, and less than 10% of the execution time is spent on memory stalls. Note that most of this

stalling could have been eliminated by implementing a streaming DMA transfer to the local memory that is overlapped with the computation.

The CSLC on Raw uses radix-2 FFT to avoid register spilling encountered in the radix-4 FFT. The number of operations (including loads and stores) in the radix-2 FFT is about 1.5 the number in the radix-4 FFT. So care should be given when the performance of the Raw on CSLC is compared with CSLC performance on other architectures.

One problem with our data parallel implementation of CSLC on Raw was load balancing. The CSLC is easily parallelized for 16 tiles. However, since the number of data sets is 73, which is not a multiple of the number of tiles, some tiles processed five sets while others processed four sets. About 8% of CPU cycles are idle due to load balancing. However, the number of sets in a real environment is not fixed at 73. In a real implementation, the input data sets would arrive continuously. Therefore, it is reasonable to assume that Raw could have perfect load balancing in a real implementation. Thus, we report the performance numbers for CSLC on Raw based on an extrapolation that assumes perfect load balancing.

Raw achieves about 31.4% of the peak performance on CSLC. In addition to the cache stall time previously discussed, about 26% of the cycles on Raw are consumed by load and store instructions. The remaining cycles are consumed by address and index calculations and loop overhead instructions.

If FFT is implemented using the stream interface that uses static network, it hides the cache miss stalls, and load and store operations are not needed. A primitive implementation result suggests about 70% of FFT performance improvement.

### 4.4 Beam steering

Beam steering has small numbers of memory accesses (2 reads and 1 write) and computations (5 additions and 1 shift) per output data.

On VIRAM, the lower bound of the computation time is 56% of the simulation time. The difference between the expected time and simulation cycles (15,412) comes from waiting for the results from previous vector operations and the cycles needed to initialize the vector operations.

On Imagine, the computations and memory accesses for beam steering are overlapped. The performance is limited by memory bandwidth due to the relatively low number of computation per memory access. The load and store operations take 89% of the simulation time. The remaining 11% of execution time is due to the software pipeline prologue.

In an actual signal processing pipeline the beam steering kernel would stream its inputs from the proceeding kernel in the application (e.g., a poly-phase filter bank) and stream its outputs to the following kernel (e.g., per-beam equalization). In such a pipeline the performance of beam steering will not be limited by memory bandwidth, as in the case of this isolated kernel, but rather will be limited by arithmetic performance. On such a streaming application Imagine is expected to achieve a high fraction of its peak performance. If table values were read from the stream register file rather than memory on our kernel, performance would be increased by a factor of about two. The performance of a beam steering algorithm with more computation per data (which is a realistic assumption) could be much higher.

On Raw, we used the static network to stream data from memory while hiding memory latency. In this implementation, loads and stores are not necessary and ALU utilization is very high. The Raw beam steering implementation has the best performance of the three architectures because of the combination of memory bandwidth and high ALU utilization.

### 4.5 Altivec mapping

The PowerPC G4 provides a vector instruction set extension, which was used manually to achieve the G4 results shown in Section 4.1. The Altivec instruction set allows four 32-bit floating-point operations to be specified and executed in a single instruction. Using the AltiVec architecture gains a performance factor of about six for the CSLC and about two for beam steering and does not significantly improve performance for the corner turn, which is limited by main memory bandwidth.

### 4.6 Architecture comparison

VIRAM's primary advantage comes from the high bandwidth between the vector units and DRAM without paying the cost (in terms of pins and power) that are required to achieve high bandwidth between chips. VIRAM is especially suitable for vectorizable applications that can utilize the high bandwidth interface and that are small enough to fit in the on-chip memory. VIRAM outperformed the G4 Altivec by more than a factor of 10 on all three of our kernels and showed especially good performance on the kernels that emphasize memory bandwidth. For embedded applications with reasonably sized data sets, the VIRAM can be used as a one-chip system. If the application size is larger than the on-chip DRAM, the data needs to come

from off-chip memory and VIRAM would lose much of its advantage.

Imagine's high peak performance can be utilized in streaming applications where main memory accesses can be avoided or minimized. The CSLC kernel demonstrates that even when the Imagine ALUs are not fully utilized, performance can be quite high, especially when compared to a commercial microprocessor like the G4 Altivec. Imagine's stream-based architecture is designed for scalability and power efficiency and the Imagine architecture has the highest peak performance of the architectures in this study.

Raw also performs best on streaming applications since load and store operations can be eliminated and the static networks provide tremendous on-chip bandwidth. The kernels used in this study do not fully exploit this mode of execution. But we have shown that the tile structure of Raw can be used to utilize the memory bandwidth available from the external ports of Raw. The tile structure also provides flexible support for MIMD and ILP applications.

## 5. CONCLUSION

The authors have presented simulated performance results for three data-intensive radar processing kernels: the corner turn, coherent side-lobe canceller, and beam steering on systems based on three recent research processors (VIRAM, Imagine, and Raw). The results show that all three of these architectures have strengths and provide significant performance potential compared to the current generation of superscalar processors with vector extensions.

These emerging architectures demonstrate that they can be programmed quickly in high level languages and existing compilers to obtain adequate performance, while with hand optimization or future compilers, they can achieve performance that far outstrips existing architectures. Furthermore, all three of these architectures will scale as technology shrinks far better than today's superscalar processors.

## 6. ACKNOWLEDGMENTS

The authors gratefully acknowledge the extraordinary support of the UC Berkeley IRAM team, the Stanford Imagine team, and the MIT Raw team for the use of their compilers, simulators, and computational kernels and their generous help. This study obviously would not have been possible without their generous support.

The authors also appreciate comments, suggestions, and help from Krste Asanovic, Christos Kozyrakis, Bill Dally, Anant Agarwal, Brian Patrick Towles, Jung Ho Ahn, Abhishek Das, Brucek Khailany, Ujval J. Kapasi, John Owens, Michael.B.Taylor, Hank Hoffmann, Dong-In Kang, and Lavanya Swethranyan.

Effort sponsored by Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory, USAF, under agreement number F30602-99-1-0521 and F30602-01-C-0171. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government.

## 7. References

[1] Apple, http://www.apple.com/powermac/, 2002.

[2] M. Gordon, W Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A Stream Compiler for Communication-Exposed Architectures, MIT Tech. Memo TM-627, Cambridge, MA, March, 2002.

[3] A. Gupta, J. L. Hennessy, K. Gharachorloo, T. Mowry, and W. D. Weber, "Computative Evaluation of Latency Reducing and Tolerating Techniques," Proc. 18th Annual International Symposium on Computer Architecture, Toronto, May 1991.

[4] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang., "Imagine: Media Processing with Streams," IEEE Micro, March/April 2001, pp. 35-46.

[5] C. Kozyrakis, "Scalable Vector Media-processors for Embedded Systems," Ph. D. dissertation, UC Berkeley, May 2002.

[6] U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The Imagine Stream Processor," International Conference on Computer Design, Freiburg, Germany, September 2002.

[7] J. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishers, Inc., 1996.

[8] Mitsubishi Microcomputers, M32000D4BFP-80 Data Book, http://www.mitsubishichips.com/data/datasheets /mcus/ mcupdf/ds/e32r80.pdf.

[9] J. D. Owens, S. Rixner, U. J. Kapasi, P. Mattson, B. Towles, B. Serebrin, and W. J. Dally, "Media Processing Applications on the Imagine," Stream Processor Proceedings of International Conference on Computer Design, Freiburg, Germany, September 2002.

[10] S. A. Przybylski, Cache and Memory Hierarchy Design: A Performance-Directed Approach, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[11] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, "A Bandwidth-Efficient Architecture for Media Processing," 31st Annual International Symposium on Microarchitecture, Dallas, Texas, November 1998.

[12] A. J. Smith, "Cache Memories," Computing Surveys, Vol. 14, No. 3, pp. 473-530, 1982.

[13] J. Suh and S.P. Crago, "PIM- and Stream Processor-based Processing for Radar Signal Applications," MSP 02, Austine, TX, 2002.

[14] J. Suh, S. P. Crago, C. Li, and R. Parker, "A PIM-based Multiprocessor System," International Parallel and Distributed Processing Symposium, San Francisco, CA, 2000.

[15] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, W. Lee, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, and A. Agarwal, "A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network," *Proceedings of the IEEE International Solid-State Circuits Conference*, February 2003.

[16] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, "Scalar Operand Networks: On-chip Interconnect for ILP in Partitioned Architectures," International Symposium on High Performance Computer Architecture, February 2003.

[17] C. Kozyrakis, D. Patterson, "Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks," *35th International Symposium on Microarchitecture*, Instabul, Turkey, November 2002.