Scientific Kernels on VIRAM and Imagine Media Processors

Manikandan Narayanan[1], Leonid Oliker[2]
Adam Janin[1,3], Parry Husbands[2], and Xiaoye Li[2]

Abstract

Many high performance applications run well below the peak arithmetic performance of the underlying machine, with inefficiencies often attributed to a lack of memory bandwidth. In this work we examine two emerging media processors designed to address the well-known gap between processor and memory performance, in the context of scientific computing. The VIRAM architecture uses novel PIM technology to combine embedded DRAM with a vector co-processor for exploiting its large bandwidth potential. The Imagine architecture, on the other hand, provides a stream-aware memory hierarchy to support the tremendous processing potential of the SIMD controlled VLIW clusters. First we develop a scalable synthetic probe that allows us to parametize key performance attributes of VIRAM and Imagine while capturing the performance crossover point of these architectures. Next we present results for two important scientific kernels each with a unique set of computational characteristics and memory access patterns. Our experiments isolate the set of application characteristics best suited for each architecture and show a promising direction towards interfacing leading-edge media processor technology with high-end scientific computations.

1 Introduction

Traditionally, HPC technologies have been based on custom hardware designed specifically for that market. However, recent market forces have caused most modern supercomputing systems to rely on commodity based components. Since multi-media applications are becoming the dominant consumer of computing cycles [17], there is a correspondingly large effort to improve chip technology and ultimately create commodity components designed to efficiently process high-end media applications. Therefore it is important for the high-end scientific community to leverage the efforts of media processor development and investigate the overlap between the architectural requirements of both domains. From an applications perspective, both scientific and media processing fields share many of the same computational algorithms and can contain a high volume of data-parallelism: examples include linear algebra kernels as well as spectral transformations. In this work we examine two novel general-purpose media processors, each representing significantly different balances of architectural characteristics, in the context of scientific computing kernels.

Historically, embedded multimedia and signal processing chips have been manufactured as custom-designed ASICs; however, this is becoming impractical for many application fields due to the high cost and the relatively slow design cycle of custom fabrication. General purpose processors, on the other hand, remain unsuitable despite ever

[1] Computer Science Division, University of California, Berkeley CA 94720
[2] Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley CA 94720
[3] International Computer Science Institute, Berkeley CA 94704

increasing clock speeds and multimedia specific enhancements (such as Intel's MMX [23] extensions), due to their relatively poor performance and high power consumption. Media applications, unlike many classes of programs, exhibit poor temporal locality and receive little benefit from automatically managed caches of conventional microarchitectures. In addition, a significant fraction of media codes are characterized by predictable fine-grained data-parallelism that could be exploited at compile time with properly structured program semantics. However, most superscalar general-purpose processors are poor at dynamically exploiting this kind of parallelism, and are too expensive in terms of power consumption. Finally, many media programs require a bandwidth-oriented memory system; unlike conventional cache-based memory hierarchies that are entirely organized around reducing average latency time, and generally lack the raw bandwidth required for these applications. This paper presents two emerging media microprocessors, VIRAM and Imagine, and evaluates their potential efficacy for addressing the growing memory-gap of high-end numerical simulations.

First we develop a scalable synthetic probe called ***Sqmat*** that allows us to parametize key performance attributes of VIRAM and Imagine. ***Sqmat*** was specifically designed to reveal architectural characteristics of the two media processors in this study. By varying ***Sqmat's*** computational requirements, we explore the main architectural features of VIRAM and Imagine, and observe the crossover point where one technology becomes more suitable to the other. We then present two important scientific kernels, each requiring a different balance of microarchitectural resource to achieve high performance. The ***SPMV*** benchmark performs sparse matrix-vector multiplication, and is characterized by irregular data access and low computation per memory access. In contrast, our second scientific kernel ***QRD,*** performs the Householder QR factorization of complex matrices, and has a relatively high computational intensity for each data access. The purpose of this work is not to compare VIRAM and Imagine from a traditional benchmarking perspective. Instead, we use our scientific kernel codes to explore the salient features of these unique architectures, and define the program characteristics best suited for each of these radically different emerging technologies.

2 Architecture, Programming Paradigm, and Kernel Overview

In this section we provide a brief overview of the two media processors examined in this study, a summary of their programming paradigms and a description of the scientific kernels used in our experiments.

2.1 VIRAM The VIRAM processor [2] is a research architecture being developed at UC Berkeley. A floor plan of the VIRAM-1 prototype chip is presented in *Figure 1*. Its most novel feature is that is complete system on a chip, combining processing elements and 13 MB of standard DRAM into a single design. The processor-in-memory (PIM) technology allows the main RAM to be in close proximity to the processing elements, providing lower memory latency and a significantly wider memory interface than conventional microprocessors. The resulting memory bandwidth is an impressive 6.4 GB/s. VIRAM contains a conventional general purpose MIPS scalar processor on-chip, but to exploit its large bandwidth potential, it also has a vector co-processor consisting of 4 64-bit vector lanes. VIRAM has a peak performance of 1.6 GFlop/s for 32 bit data and is a low power chip, designed to consume only 2 Watts of energy.

The hardware resources devoted to functional units and registers may be subdivided to operate on 8, 16, 32, or 64-bit data. When the data width (known as the virtual processor width) is cut in half, the number of elements per register doubles, as does the peak arithmetic rate. The variable data widths in VIRAM are common to other SIMD media extensions such as Intel's SSE, but otherwise the architecture more closely matches vector supercomputers. In particular, the parallelism expressed in SIMD extensions are tied to the degree of parallelism in the hardware, whereas a floating-point instruction in VIRAM specifies 64-way parallelism while the hardware only executes 8-way. The advantages of specifying longer vectors include lower instruction bandwidth requirement, a higher degree of parallelism for memory latency masking, and the ability to change hardware resources across chip generations without requiring software changes.

2.2 Imagine A different approach for addressing the processor-memory gap is through stream processing. Imagine [12] is a programmable streaming microprocessor currently being developed at Stanford University. Stream processors are designed for computationally intensive applications characterized by high data parallelism and producer-consumer locality with little global data reuse. The general layout diagram of Imagine is presented in *Figure 2*. Imagine contains 48 arithmetic units, and a unique three level memory hierarchy designed to keep the functional units saturated during stream processing. The architecture is centered around a 128 KB stream register file (SRF), which reads data from off-chip DRAM through a memory system interface and sequentially feeds the 8 arithmetic clusters. The local storage of the SRF can effectively reuse intermediate results (producer-consumer locality), allowing for the amortization of off-chip memory accesses. In addition, the SRF can be used to overlap computations with memory traffic, by simultaneously reading from main-memory while writing to the arithmetic clusters (double-buffering). The Imagine architecture emphasizes raw processing power much more heavily then VIRAM with a peak performance of 20 GFlop/s for 32 bit data.

Each of Imagine's 8 arithmetic clusters consists of 6 functional units containing 3 adders, 2 multipliers, and a divide/square root. Imagine is a native 32-bit architecture; with support for performing operations on 16- and 8-bit data resulting in two and four times the peak performance respectively. This is analogous to VIRAM's virtual processor widths; however, unlike VIRAM there is no support for 64 bit operations. Thus we restrict our study to 32-bit data elements. A key difference between the two architectures is in the way instructions are issued. In Imagine, a single microcontroller broadcasts VLIW instructions in SIMD fashion to all of the arithmetic clusters. In contrast, VIRAM uses a more traditional single instruction per cycle issue, counting on parallelism within each vector instruction to achieve high performance.

Table 1 summarizes the high level differences between the VIRAM and Imagine architectures. Notice that Imagine has an order of magnitude higher peak performance, while VIRAM has twice the memory bandwidth and consumes half the power. Also observe that VIRAM has enough bandwidth to sustain one operation per memory access, while Imagine requires 30 operations to amortize one word of off-chip memory, and 2.5 operations for SRF references. In order to gain deeper insight into the two architectures, we constructed a scalable synthetic probe called ***Sqmat***. Using this simple benchmark, with abundant fine-grained data parallelism and no data dependencies, allows us to examine a spectrum of computational requirements while correlating performance to the underlying architectural features. Details are presented in *Section 3*.

2.3 Programming Paradigm and Software Environment

The vector programming paradigm [20] of VIRAM is well understood and can leverage off of years of algorithmic research as well as sophisticated compiler technologies. Logically, a vector instruction specifies the parallel operations to be performed on all elements of the vector register. However, at the hardware level each vector instruction splits into multiple element groups that then perform the operations. For example, when operating on 32-bit data in VIRAM, the logical vector length refers to 64 elements while the physical configuration contains only 8 lanes. Therefore each vector instruction results in the execution of 64/8=8 element groups, where each group uses the actual vector hardware to process 8 elements at a time.

Imagine supports the relatively new stream programming paradigm, designed to express the high degree of fine-grained parallelism necessary to effectively utilize the large number of functional units. The stream programming model organizes data as streams and expresses all computations as kernels [14]. A stream is an ordered set of records of arbitrary (but homogeneous) data-objects. For example, in a finite-element scientific simulation the computational stream could contain a set of records, where each record element represents various physical components of the experiment (such as pressure, velocity, position, etc.) Vectors, on the other hand, are restricted to operating on basic data types, and must decompose complex records into vectors of separate elements. Kernels perform computation on entire streams, by applying potentially complex functions to each stream record in order. However, kernels cannot make arbitrary memory reference and are limited to only accessing data from the SRF in a sequential fashion. The kernel memory reference restrictions allow the memory subsystem to effectively provide data to the large number of functional units. However, these memory access limitations increase programming complexity, especially for irregularly structured applications.

Both the vector and stream programming paradigms provide methods for expressing the fine-grained data parallelism of an application. Providing for explicit parallelism in the ISA, allows the underlying hardware to directly support vectors or streams, in an energy-efficient manner. The application performance, however, is highly correlated to the fraction of the application amenable to data parallelism. A key distinction between the two models is that the Imagine architecture supports streams of multi-word records directly in the ISA, as opposed to VIRAM whose ISA support is limited to vectors of basic data-types. Going back to our finite-element example, Imagine is able to access the multi-word data records of the simulation in a unit-stride fashion from main memory. Appropriate reordering is then performed in the on-chip memory subsystem, before passing the correctly structured data to the SRF. However, in vector architectures, strided accesses are required to load each basic data type of the underlying physical component causing potential memory overheads, detailed in *Section 3.2.1*. This permits Imagine to access off-chip main memory in a more efficient manner. Additionally, organizing streams as multi-word records also increases kernel locality, allowing for efficient VLIW processing by each of the functional units. Other advantages of multi-word parallelism include the potential of reduced programming complexity and low instruction bandwidth.

We end this section with a brief description of the software environment. In VIRAM, applications are coded in C using the vcc [16] vectorizing compiler. However, it is occasionally necessary to hand tune assembly instructions to overcome the deficiencies of the compiler environment. In Imagine, two languages are used to express a program: the StreamC language is used to coordinate the streaming of data while KernelC is used to define the computational kernels to be performed on each stream record. Separate stream and kernel compilers then map these two languages to the ISA of the stream controller and micro-controller respectively. The Imagine software environment allows for automatic code optimizations such as loop-unrolling and software pipelining, as well as visual tools for isolating performance bottlenecks. The results reported in this paper were gathered from the VIRAM and Imagine cycle-accurate simulators.

2.4 Scientific Kernels The first scientific kernel we examine is sparse matrix vector multiply ***SPMV***. This is one of the most heavily used algorithms in large-scale numerical simulations, and is a critical component in data mining, as well as signal and image processing applications. For example when solving large sparse linear systems or eigensystems, the running time is generally dominated by the ***SPMV*** kernel. The performance of sparse matrix operations tends to perform poorly on modern microprocessors due to the low ratio between arithmetic computation and memory accesses. Additionally, the irregular data access of this algorithm is inherently at odds with cache-based architectures. It is therefore important to evaluate the performance of VIRAM and Imagine in the context of ***SPMV***.

Our second scientific kernel is the QR decomposition of a complex floating-point matrix ***(QRD)***. ***QRD*** is a well-known linear algebra algorithm commonly used in scientific computing and signal processing applications. It is also a key component of a larger space-time adaptive processing application (STAP), which is used to cancel clutter and interference in airborne radar images [5]. Unlike the ***SPMV*** kernel, ***QRD*** is a dense matrix method with a high operation count for each word of data access. We therefore evaluate the performance behavior of VIRAM and Imagine for two scientific kernels, each with vastly different computational requirements and data access patterns.

3 Insights Into the Architectures

In order to gain insight into the architectural differences between VIRAM and Imagine, we constructed a scalable synthetic probe called ***Sqmat***. This specially designed microbenchmark has several tunable parameters used to isolate key characteristics of both systems, and capture the performance crossover point of these radically different technologies.

3.1 Sqmat Overview The computational task of ***Sqmat*** is to square a set of L matrices of size NxN repeatedly M times. By varying N and M, we can control the size of the computation kernel, as well as the number of arithmetic operations per memory access. In addition, by varying the number of matrices (L) we can correlate the vector/stream length with performance.

The squaring of each NxN matrix requires N^3 multiplications and $N^2\cdot(N\text{-}1)$ additions, while requiring $2N^2$ memory accesses (loading and storing 32 bit words). On VIRAM the minimum number of cycles (algorithmic peak) required to perform M repeated squarings of L matrices is $L\cdot M\cdot(2N^3\text{-}\ N^2)/8$, since each of the 8 vector lanes can perform one 32-bit flop per cycle. Additionally, the total number of operations per word of memory accessed in VIRAM is $M\cdot(2N^3\text{-}N^2)/2N^2 = M\cdot(2N\text{-}1)/2$. However, the analysis is somewhat different for Imagine since it contains multiple functional units per cluster and operates in VLIW fashion. To calculate Imagine's algorithmic peak performance, we can effectively ignore the cost of addition operations because Imagine can perform 3 adds and 2 multiplies per cycle, while the ***Sqmat*** benchmark requires fewer additions than multiplications. As a result Imagine's peak performance for ***Sqmat*** requires only $L\cdot M\cdot N^3/16$ cycles, since each of the 8 clusters can perform 2 multiplies per cycle. Additionally, the ratio between the number of multiplies performed per memory access is $M\cdot N^3/2N^2 = N\cdot M/2$. Thus for the ***Sqmat*** example, Imagine is required to sustain about twice the memory bandwidth of VIRAM to keep its functional units optimally saturated. Finally, note that due to limitations imposed by the number of VIRAM vector registers, N could be no larger then 3 for the repeated squaring (M>1) experiments.

3.2 Sqmat Performance We start by setting the ***Sqmat*** probe to the low end of the performance spectrum and work our way up to high efficiency, at each point highlighting the relevant architectural features. Our goal is not use ***Sqmat*** for benchmarking these systems, but rather as a tool for gaining insight into their key architectural features.

3.2.1 Low Operations per Memory Access In our first experiment, we examine 5 matrices (N=1..5), with a single matrix squaring (M=1) and short vector/stream length (L=16). Limiting this example to only a single squaring of the matrices causes relatively few operations per word of data access and results in high stress on the memory system. In addition, the short vector/stream lengths deleteriously affect the performance of both architectures. *Table 2* shows the percentage of theoretical peak achieved on VIRAM and Imagine. Notice that both architectures show poor performance for low N, achieving only 4.0% and 0.1% respectively. As N increases, so does the ratio of computation to memory access; thus improving performance. However, for N=5 Imagine's performance is still very poor achieving only 2.9% of peak. VIRAM, on the other hand, sustains 36.9%, a surprisingly large fraction of its peak performance considering the low volume of required computations and short vector length.

Figure 3 compares performance between VIRAM and Imagine for N=3 of a single matrix squaring (M=1), but here we examine the effects of increasing the vector/stream length varying L from 8 to 1024. Imagine's stream model requires large number of arithmetic operations per memory access to effectively use the underlying hardware. Therefore this benchmark example is not well suited for the Imagine architecture. The computational rate is too low to amortize off-chip memory bandwidth, and the SRF is not being used effectively since there is no producer-consumer locality in this example. Another requirement for good streaming performance is that the stream must be long enough to hide memory latency. *Figure 3* shows that as L is increased from 8 to 1024, peak performance gradually improves, but plateaus at only 7% of peak performance for N=3. For each kernel called, there are a number of overheads, including: sending the instructions from the host to the microcontroller, scheduling the SRF, and filling/draining the software pipeline. Thus performance is expected to improve with larger *L* since these costs are amortized. Additionally, increasing the stream size helps amortize expensive off-chip memory latency.

For VIRAM, on the other hand, performance starts low but quickly grows with *L* to a reasonable fraction of ***Sqmat's*** theoretical peak performance, achieving almost 40% when $L \geq 256$. The vector pipelines effectively hide memory access overheads by overlapping loads with arithmetic operations. In addition, the on-chip DRAM allows for high-bandwidth and low latency memory access. These examples demonstrate that the architectural balance of VIRAM is better suited for this difficult class of problems, characterized by low computational requirements and relatively short vector lengths.

A critical issue in determining performance on VIRAM, however, is the memory access patterns. *Table 3* shows performance for *N*=1, *M*=1 and *L*=256 for various strides, both with and without address generation. Best performance is achieved when using unit-stride and no address generation as seen in column two. The third column demonstrates the effects of using the same unit-stride memory access pattern, but with address generation turned on in the assembly instructions: resulting in a 46% degradation of ***Sqmat's*** performance. This is because VIRAM can only generate 4 addresses per cycle, independent of the data width. Although for 64-bit values there is sufficient address generation to load or store a value every cycle, but when working with the 8 32-bit lanes, the arithmetic units can more easily be starved for data. On the fourth and fifth columns performance significantly degrades due to both non-unit stride memory access patterns and the necessity of memory address generation. As we increase the memory stride, the DRAM bank structure can become apparent as multiple accesses to the same bank requires additional latency to charge the DRAM. The frequency of the bank-conflicts depends on the memory access pattern as well as number of banks and subbanks in the memory system. Stride effects are not as pronounced in Imagine, however, due to the on-chip streaming memory system, which allows multi-word records to be read in unit stride from main-memory and uses reorder buffers to properly arrange the data into the SRF, as discussed in *Section 2.3.*

3.2.2 High Operations per Memory Access *Table 4* presents performance results when the matrix is repeatedly squared (*M*=1,10,20), using the 3x3 matrix and a relatively large vector/stream length of 1024. As expected, both architectures achieve higher percentage of peak as *M* increases since there is more required computation for each word of data access. VIRAM achieves close to peak performance at 82% and 89% for *M*=10,20 respectively, an increase of more than a factor of 2 from the *M*=1 experiment. Imagine's performance also improves to 38% (*M*=10) and 49% (*M*=20) efficiency, a 7x improvement from the single squaring (*M*=1) experiment.

One reason for the impressive improvement in Imagine's performance is that the computational kernel is now significantly bigger. For small *K*, the number of arithmetic operations per kernel call is small, and the fixed overheads of each kernel call can dominate performance. These overheads include reading and writing from the SRF to the clusters, and filling/draining the kernel pipeline. For example, for the 3x3 matrix with *M*=1, the kernel's ideal execution time is 14 cycles, however each actual kernel execution requires 34 cycles, meaning that only 41% of the operations are going towards matrix multiplication. However, when the kernel size is increased to *N*=5 and *M*=10, the efficiency of each kernel call rises to 91%.

Although Imagine's performance has significantly improved from the *M*=1 experiment in *Section 3.2.1*, it still achieves less then 50% efficiency for the large stream size (*L*=1024). This is somewhat surprising since the ratio between multiplications and memory access is now 30, which should be enough operations to fully saturate the computational units and amortize off-chip memory references. These results demonstrate that for the Imagine stream architecture a very large number of ops per word are required to fully utilize the underlying hardware. However, Imagine's 50% efficiency in ***Sqmat*** translates to 8GFlop/s of performance, showing there are sufficient computational requirements in this parameter set to effectively utilize the large-scale processing power of the Imagine architecture.

A key aspect of the streaming paradigm is the concept of producer-consumer locality, where data is circulated between the SRF and arithmetic clusters, thereby avoiding expensive off-chip memory access. We explore this architectural feature by using two different approaches to perform ***Sqmat's*** repeated matrix squarings, while keeping the number of operations fixed (constant *M*). The first method of achieving a given *M* is to make the kernel more computationally intensive, as is performed in VIRAM. Imagine, however, allows a second approach where producer-consumer locality is utilized by using a less computationally intensive kernel and repeatedly passing the partial matrix product through the SRF until the calculation is complete. We therefore define *K* as the number of times the matrix is squared in the kernel, and *S* as the number of times that kernel is repeatedly called, where $M=K \cdot S$. By varying *K* and *S*, we can explore the effect of producer-consumer locality on kernel performance.

Figure 4 shows the percentage of peak performance for various combinations of *K* and *S* on Imagine. Stream length (*L*) here is set to very large optimal values to avoid any short-stream effects[4]. Notice that in order for the producer-consumer locality to work effectively, the kernel computations must be high enough to amortize the fixed kernel overheads. This can be seen by from the *K*=1 data set where performance is significantly below *K*=5,10 even though the total number of matrix squarings are the same. However, we see littler difference in the percentage of peak performance when *K* equals 5 and 10, showing that producer-consumer locality can be effectively used when the computational kernels are reasonably large. Finally, observe that even though up to 600 multiplications are performed for each word of data access, the percentage of theoretical peak plateaus at only 70%. This phenomenon is due to the SRF bandwidth effects, as well as fixed kernel overheads; demonstrating that producer-consumer computational intensity is not sufficient to fully saturate the arithmetic clusters, if the underlying kernels do not perform very large numbers of operations.

3.2.3 Peak Performance and Crossover *Table* 5 presents VIRAM and Imagine performance on ***Sqmat*** under ideal conditions. By running ***Sqmat*** using high computational requirements together with optimal vector/stream lengths, we can effectively mitigate memory access penalties and achieve over 90% of theoretical peak performance

[4] Stream length ranges from 4320 to 18160 depending on the optimal strip size as predicted by the software development environment.

on both architectures. Notice that for the Imagine case the matrix size had to be increase to *N*=5 to overcome fixed kernel overheads.

Finally, *Figure 5* presents the performance crossover point where Imagine outperforms VIRAM in terms of cycles (*L*≥256) and MFlop/s (*L*≥64). Here the raw processing power advantages of Imagine become apparent, achieving almost a 4x performance increase over VIRAM in terms of MFlop/s for *L*=1024. Codes characterized by this balance of computational intensity and memory requirements would greatly benefit from Imagine's streaming architecture.

4 Sparse Matrix Vector Multiplication

For the ***SPMV*** kernel we examined 3 different implementation strategies for Imagine and VIRAM, each of which highlights different aspects of the underlying architecture. We chose two matrices for this experiment, each with different characteristics that enable us to explore how architectural and programming differences affect performance. The first matrix *LSHAPE* is from Harwell-Boeing collection and represents a finite matrix problem. It is a 1008x1009 matrix with an average of 6.8 nonzeros and a maximum of 7 nonzeros per row. Our second matrix *LARGEDIS* is the same one used in previous IRAM experiments [9], and contains a pseudo-random pattern of non-zeros using a construction algorithm from the DIS specification [8], parameterized by the matrix dimension, and the number of nonzeros. The input matrix size is 10000x10000 with an average of 18 nonzeros and a maximum of 82 nonzeros per row.

4.1 VIRAM Implementation We consider 3 algorithms for the ***SPMV*** implementation on VIRAM [9], each reflecting a different optimization strategy for vector architectures. Compressed Row Storage (*CRS*) is the most common sparse matrix format, which stores an array of column indices and non-zero values for each row; ***SPMV*** is then performed as a series of sparse dot products. The second approach uses the *Ellpack* (or Itpack) format [15], which forces all rows to have the same length by padding them with zeros. It still has indexed memory operations, but increases available data parallelism through vectorization across rows. Finally, we experimented with the segmented sum (*Segsum*) algorithm, originally developed for the Cray PVP [4]. The data structure is an augmented form of the CRS format and the computational structure is similar to Ellpack, although there is an additional control complexity. Since VIRAM can only generate four addresses per cycle, the large stride memory access is slow, as discussed in *Section 3.2.1*. Therefore, we modified the original code to make it unit stride.

4.2 Imagine Implementation A key component of Imagine's streaming paradigm is that the computational clusters can only access data in a sequential fashion from the SRF. However ***SPMV*** requires irregular data access to properly index the source vector. Therefore, in all of the Imagine ***SPMV*** implementations, the data is properly reordered from main-memory into the SRF to avoid the need for any indirect addressing during computation. Additionally, the indexed source vector stream is expanded to as many elements as in the sparse matrix, since it is not possible to arbitrarily access the vector data.

Our first Imagine implementation uses the Compressed Row Storage (*CRS*) format and performs a series of sparse dot products. However, since the rows have different numbers of nonzeros, assigning each cluster to sum a unique row is inefficient since the lock-step execution of the SIMD architecture would limit performance to the longest computation. Our algorithm therefore uses all eight arithmetic clusters to process one row at a time, using intercluster communication to perform the summation reduction. Note that in this algorithm, conditional input and output streams are used to selectively fetch the correct number of row elements and properly output the result only when the row computation is complete. Since the kernel language has extremely restricted conditional syntax, conditional streaming [13] allows a number of important operations, such as handling data-dependent control constructs, merging or appending streams, and limited load balancing.

Our next implementation strategy (*Streams*) leverages the stream concept of producer-consumer locality. Here, in addition to the matrix and indexed vector, the computational kernel receives a third (sentinel) stream indicating which nonzeros entries are at the end of a row. Based on this information, the arithmetic clusters selectively sum two elements if they are determined to be on the same row. The partial sum is repeatedly passed through the computational kernel until the dot product summation is complete. The third version is an Imagine implementation of the VIRAM *Ellpack* algorithm. In this routine we fill the rows of the sparse matrix such that each has the same number of non-zeros. Each cluster then performs all of the required multiply-adds on a given row and outputs the corresponding entry of the result. This results in a very simple kernel, but its performance is dependent on the length of the row.

4.3 Performance Results *Table 6* presents the performance results of ***SPMV*** on VIRAM and Imagine using the *LSHAPE* and *LARGEDIS* matrices. Note that algorithmic peak performance of ***SPMV*** on VIRAM is 8 operations per cycle (one for each vector lane), while on Imagine arithmetic peak performance is 32 operations per cycle (2 multiplies and 2 adds for each of 8 clusters). Comparing SPMV performance using the original matrix patterns (*CRS*, *Segsum,* and *Streams)* performance is rather low due to the lack of parallelism, since the average rows of *LSHAPE* and *LARGEDIS* contain only 8 and 18 nonzeros respectively. However, notice that VIRAM achieves a significantly higher fraction of peak performance than Imagine (8.4% vs. 1.5% on *LARGEDIS*). Imagine's ***SPMV*** *Streams* implementation sustained particularly poor performance of less then 1% efficiency. We believe that this was partially due to the unpredictable length of the output streams after each kernel cycle, which caused the stream scheduler to function inefficiently.

Results are even more dramatic when comparing the *Ellpack* (filled) implementations. Here VIRAM attains 31% and 32% of peak respectively compared with 1.2% and 6.3% in the Imagine versions, and requires fewer total cycles. However, Imagine still achieves higher MFlop/s for certain algorithm comparisons due to its large computational potential. Note, that since the Ellpack matrices are artificially padded with zeros to create symmetric row lengths, the fraction of useful operations can be arbitrarily poor depending on the matrix structure. However, this fraction of useful computations would penalize the effective performance of both architecture equally. These experiments demonstrate VIRAM's ability to effectively handle codes with low operations per data access and validates our conclusions of *Section 3.2.1.*

5 Complex QR Decomposition

In the QR decomposition, a matrix *A* is decomposed as *A=QR*, where *Q* is a orthogonal and *R* is a upper triangular matrix. A standard way of performing this decomposition is to use Householder transformations: orthogonal transformations that annihilate the lower part of each column (i.e., the part of the column below its diagonal element) of the matrix *A*, thus producing *R*. If performed in a column-by-column manner, (computing a Householder transformation for each column and updating the subsequent columns of *A* using that transformation) this process is rich in level-2 BLAS [18] operations of matrix-vector multiplication and outer product updates [10].

5.1 VIRAM and Imagine Implementations In order to increase the computation to memory ratio of the Householder QR, we use block variants of the algorithm, that are rich in level-3 BLAS operations. These block methods consider a block of columns and factorize them (using the Householder QR) to obtain an upper triangular (a diagonal block of *R*) matrix, as well as the transformation used to decompose this block. The transformations are stored in a suitable matrix representation and then applied to the subsequent columns of the matrix, and the computation begins anew with a new column block. One representation of the blocked Householder is the so-called *compact-WY* representation [3,24], which involves matrices *Y*, and *T*, that obey the identity $A=(I-YTY^H)R$, where $I-YTY^H=Q$ (Y^H is the conjugate transpose of *Y*). The reader is referred to [10] for a complete description of the blocked Householder QR. Both VIRAM and Imagine implementations use this blocked algorithm, to decompose a matrix A of complex elements. The use of complex elements enhances the computational intensity (ops/word) and the locality of the algorithm, since each complex multiplication expands to six arithmetic operations.

VIRAM implementation is a port of the *CLAPACK* [1] routine *CGEQRF* and its associated *BLAS* [18] routines. In the VIRAM implementation, columns are considered in blocks of 32 and the whole implementation is composed of calls to *BLAS* routines. The optimization process was straightforward and involved insertion of vectorization directives [20] in the source code of *BLAS* routines. For certain *BLAS* routines, loops were interchanged, converting large stride accesses to smaller ones to avoid the overheads described in *Section 3.2.1*. For instance, *SAXPY* version of matrix-vector multiply would do considerably better than the *dot-product* version [16], for matrices stored in column-major order (as in *CLAPACK*[1]). This is because the latter implementation requires strided accesses, in addition to the expensive reductions for computing the sum.

The Imagine implementation described in [19] also uses a blocked algorithm. Blocks of 8 columns are fed into kernels that compute the R matrix for that block. The Householder transformation is also computed at this point. This transformation is then applied to the subsequent column blocks of the matrix and the process iterates. Some complicated indexing of the matrix stream need to be performed as each iteration of the process requires smaller and smaller matrices.

5.2 Performance results The performance of QR on a 192-by-96 (*m-by-n*) complex matrix *A*, taken from the Mitre RT_STAP benchmark suite [5], is shown in *Table 7*. Note that this algorithm requires $8mn^2$ operations. VIRAM sustains only 34.1% of its theoretical hardware peak on this computationally intensive kernel, chiefly due to memory accesses with large stride, and achieves 546 MFlop/s. Imagine, on the other hand, performs at over 65% efficiency and shows an impressive speed of over 13 GFlop/s [14,19][5], an improvement of almost 24x in raw processing power over VIRAM . These results demonstrate the considerable performance can be obtained on Imagine on classes of computations with high operations per memory access, as described in *Section 3.2.2*.

6 Conclusions and Future Work

In this work we successfully demonstrated the overlap between emerging high-end media processors and scientific computations. We were able to gain insight into the salient features of VIRAM and Imagine in the context of numerical kernels, and quantify the computational space best suited for each processing paradigm. First, we developed a scalable synthetic probe, ***Sqmat***, which allowed us to parameterize key features of the architectures. By varying a small set of parameters we explored performance in the context of: computational intensity, vector/stream length, memory access patterns, kernel overheads, producer-consumer locality, and hierarchical memory structure. Two important scientific kernels each with distinct program behavior were then presented. We showed that the ***SPMV*** kernel, characterized by low operations per data access and irregular memory access, mapped to the low end of ***Sqmat's*** performance spectrum; Whereas the high end of ***Sqmat's*** spectrum was correlated to ***QRD,*** a dense algorithm requiring high computational intensity.

We also discussed the complex interactions between programming paradigms, architectural support at the ISA level and the underlying micoarchitecture of these two systems. The ***Sqmat*** and ***QRD*** benchmarks were able to leverage the multi-word record support of the streaming architecture. Although VIRAM's compiler was able to vectorize these multi-word codes, it was restricted to using the native vector instructions which only operate on basic-data types. As a result, VIRAM was forced to incur the overhead of strided memory accesses. However, program development was more challenging in Imagine than in the well-known vectorization paradigm, because the programmer is exposed to the memory hierarchy and cluster organization of the Imagine architecture. Improvement in the quality of the compiler and software development tools, and abstracting lower level details of the hardware will be essential in bringing the stream programming model to the wider scientific community. Brook [7] and StreamIt [25] are two examples of recently proposed high-level streaming languages that attempt to increase programmer productivity while achieving high performance.

Future plans include validating our results on real hardware as it becomes available, as well as examining a broader scope of scientific codes. We plan to evaluate more complex data-parallel systems such as the Streaming Supercomputer [7] and the Diva [11]. Our long-term goal is to evaluate these technologies as building blocks for future high-performance multiprocessor systems.

[5] As of this time we have been unable to reproduce these results. We are currently working with the Stanford team to resolve any inconsistencies.

References

1. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorenson. *LAPACK Users' Guide.* Third Edition. Society for Industrial and Applied Mathematics. 2000.
2. The Berkeley Intelligent RAM (IRAM) Project, Univ. of California, Berkeley, at http://iram.cs.berkeley.edu.
3. C. Bischof and C. Van Loan. The WY representation for products of Householder matrices. *SIAM J. Scientific and Statistical Computing.*, 8(1):s2-s13, 1987.
4. G. Blelloch, M. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Tech. Rep. CMU-CS-93-173, Carnegie Mellon Univ., Pittsburgh, 1993.
5. K. Cain, J. Torres, and R. Williams. RT_STAP: Real-time space-time adaptive processing benchmark. MITRE Tech. Rep. MTR96B021, February 1997.
6. J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *University of Tennessee at Knoxville*, CS-94-246, September 1994.
7. W. Dally, P. Hanrahan, and R. Fedkiw. A Streaming Supercomputer. *Whitepaper*, September 18, 2001.
8. DIS Stressmark Suite, v 1.0. Titan Systems Corp., 2000, at http://www.aaec.com/projectweb/dis/
9. B. Gaeke, P. Husbands, X. Li, L.Oliker, K. Yelick, and R. Biswas. Memory Intensive Benchmarks: IRAM vs. Cache-Based Machines. *Proc. 2002 International Parallel and Distributed Processing Symposium* 2002.
10. G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins Univ Press. December 1996.
11. M. Hall, P. Kogge, J. Koller, P. Diniz , J. Chame, J. Draper, J. LaCoss, J. Granacki, A. Srivastava, W. Athas, J. Brockman, V. Freeh, J. Park, J. Shin. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. *Proc. of SC99, 1999.*
12. The Imagine project, Stanford University, at http://cva.stanford.edu/imagine/.
13. U. Kapasi, W. Dally, S. Rixner, P. Mattson, J. Owens, and B. Khailany. Efficient Conditional Operations for Data-parallel Architectures *Proceedings of the 33rd Annual International Symposium on Microarchitecture,* Dec. 10-13, 2000.
14. 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*, Mar/April 2001.
15. D. Kincaid, T. Oppe, and D. Young. ITPACKV 2D user's guide. Tech. Rep. CAN-232, Univ. of Texas, Austin, 1989.
16. C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, and K. Yelick. Hardware/compiler co-development for an embedded media processor. *Proceedings of the IEEE,* 2001.
17. C. Kozyrakis. A media-enhanced vector architecture for embedded memory systems. Tech. Rep. UCB-CSD-99-1059, Univ. of California, Berkeley, 1999.

18. C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Soft., 5*, 1979.
19. P. Mattson. Programming System for the Imagine Media Processor. Ph.D. Thesis, Stanford University, 2002.
20. Maximizing CRAY T90/J90 Applications Performance - vectorization of C code. *Scientific Computing at NPACI (SCAN),* Volume 3 Issue 15: July 21, 1999.
21. J. Owens, S. Rixner, U. Kapasi, P. Mattson, B. Towles, B. Serebrin, and W. Dally. Media Processing Applications on the Imagine Stream Processor. *Proc. 2002 International Conference on Computer Design*. 2002.
22. D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, R. Thomas, C. Kozyrakis, and K. Yelick. Intelligent RAM (IRAM): Chips that remember and compute. *Proc. Intl. Solid-State Circuits Conf.*, 1997.
23. A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM,* 40(1):24-38, January 1997.
24. R. Schreiber and C. Van Loan. A storage efficient WY representation for products of Householder transformations. *SIAM J. Scientific and Statistical Computing,* 10(1):53-57, 1989.
25. W. Thies, M. Karczmarek and S. Amarasinghe. StreamIt: A Language for Streaming Applications. *Computational Complexity, pg. 179-196, 2002.*

		Imagine Memory	Imagine SRF
Bandwidth GB/sec	6.4	2.7	32
Peak Flops GFlop/s (32 bit)		20	
Peak Flop/Word	1	30	2.5
Clock Speed MHz	200	500	
Chip Area	15x18mm (270 mm^2)	12x12mm (144 mm$^{2)}$	
Data widths supported		32/16/8 bit	
Transistors	130 Million	21 Million	
Power consumption	2 Watts	4 Watts	

Table 1:***Highlights of VIRAM and Imagine architecture***

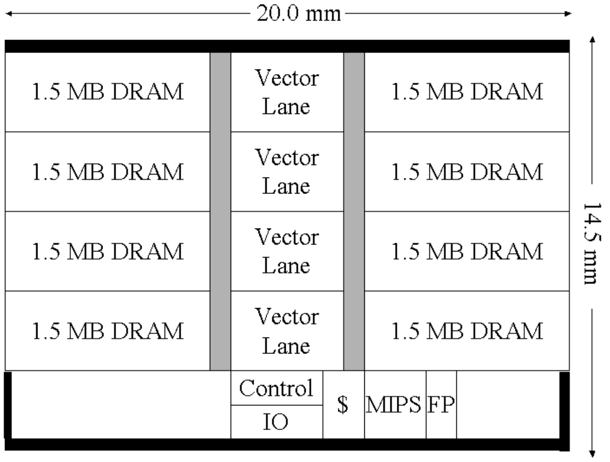

Figure 1: ***Block diagram of the VIRAM architecture***

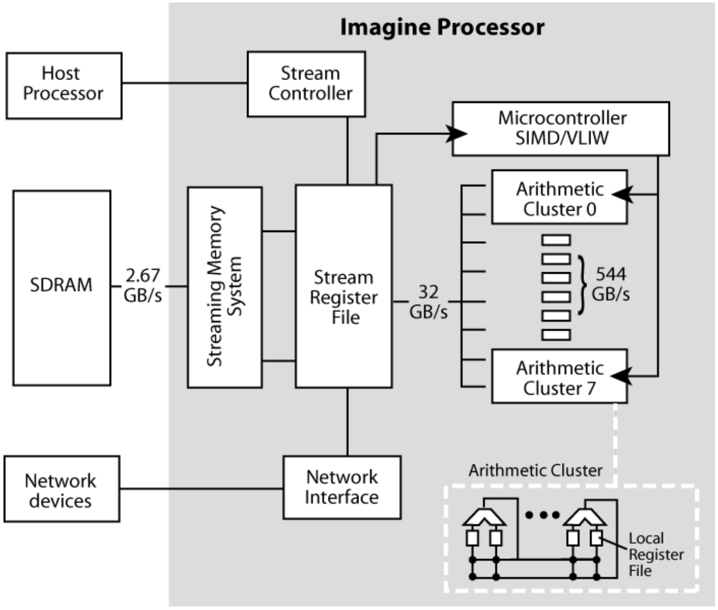

Figure 2: ***Overview of the Imagine architecture***

N	1	2	3	4	5
VIRAM		19.0%		25.3%	
IMAGINE		0.7%		2.3%	

Table 2: ***Percentage of algorithmic peak performance of VIRAM and Imagine with M =1 and L =16***

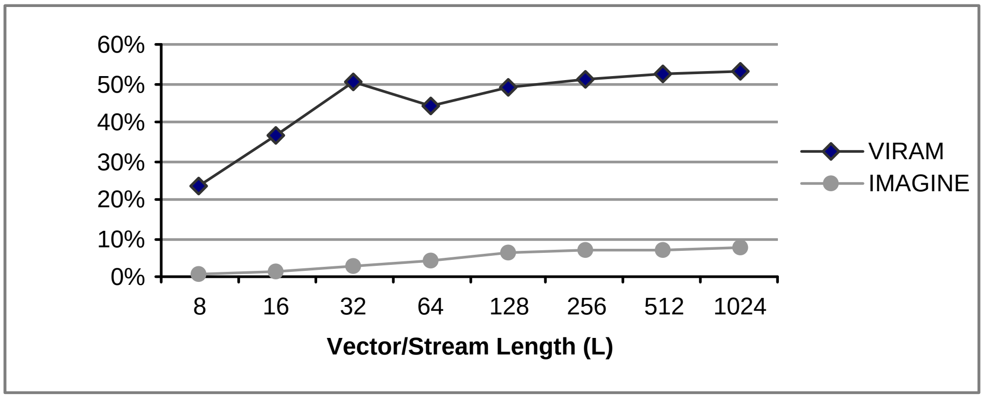

Figure 3: ***Percentage of algorithmic peak performance of VIRAM and Imagine for varying vector/stream lengths when N =3 and M = 1***

Stride				
Address Generation	No	Yes	Yes	Yes
Cycles	118	173	186	298

Table 3: Number ***of VIRAM cycles using various stride patterns, with and without address generation, for N =1, M =1, and L =256***

M	VIRAM	IMAGINE
1	**39%**	**7%**
10	**82%**	**38%**
20	**89%**	**49%**

Table 4*: Percentage of algorithmic peak performance of VIRAM and Imagine for N =3, M = 1,10,20 and L =1024*

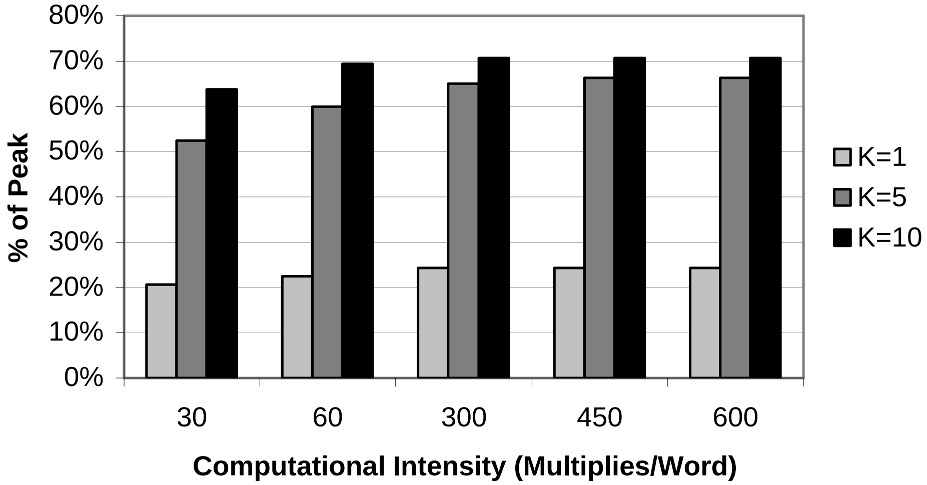

Figure 4: *Percentage of algorithmic peak performance of Imagine with N =3 and K=1,5,10 using long streams and varying computational intensity*

	VIRAM (N=3)				Imagine (N=5)			
Ops/Word	50	90	120	150	100	200	300	400
% Peak	82%	88%	89%	91%	86%	89%	90%	91%

Table 5: *Achieving high efficiency for VIRAM and Imagine using long streams and high computational intensity*

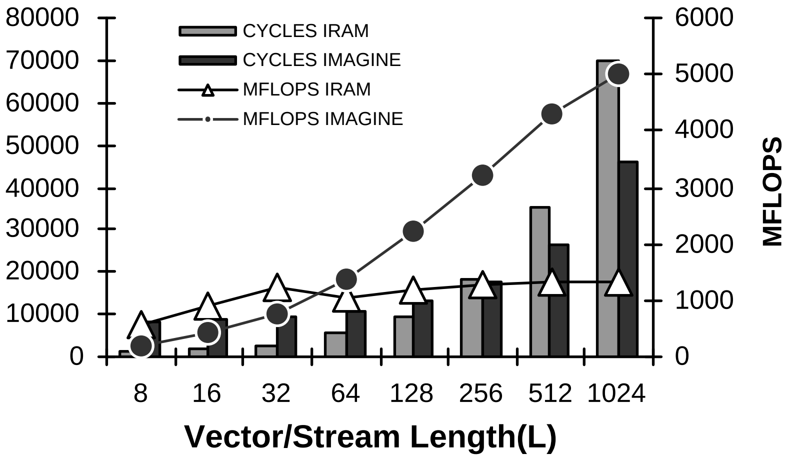

Figure 5: *Performance crossover between VIRAM and Imagine for N =3 and M =10*

Matrix Rows (Nonzeros)		CRS		Ellpack	CRS		Ellpack
LSHAPE 1008 (6958)	% of Peak	2.8%	7.4%	31%	1.1%	0.8%	1.2%
	Total cycle	66823	23802	5666	40300	48190	37930
	MFlop/s	44	118	496	170	142	186
LARGEDIS 10000 (177820)	% of Peak	3.2%	8.4%	32.0%	1.5%	0.6%	6.3%
		802070		641512	742310		753540
	MFlop/s	91	135	511	240	97	1088

Table 6: ***Performance of SPMV on VIRAM and Imagine for the LSHAPE and LARGEDIS matrices using various algorithms***

Matrix	Performance		
MITRE RT_STAP 192-by-96 complex matrix	% of Peak	34.1%	65.5%
	Total Cycles	5188817	712770
	MFlop/s	546	13,100

Table 7***: Performance of QRD on VIRAM and Imagine for the 192-by-96 MITRE RT_STAP matrix***