Performance Evaluation of Two Emerging Media Processors: VIRAM and Imagine

Sourav Chatterji, Manikandan Narayanan
Computer Science Division
University of California
Berkeley CA 94720
{souravc,nmani}@cs.berkeley.edu

Jason Duell, Leonid Oliker
Computer Science Research Division
Lawrence Berkeley National Laboratory
Berkeley CA 94720
{jcduell,loliker}@lbl.gov

Abstract

This work presents two emerging media microprocessors, VIRAM and Imagine, and compares the implementation strategies and performance results of these unique architectures. VIRAM is a complete system on a chip which uses PIM technology to combine vector processing with embedded DRAM. Imagine is a programmable streaming architecture with a specialized memory hierarchy designed for computationally intensive data-parallel codes. First, we present a simple and effective approach for understanding and optimizing vector/stream applications. Performance results are then presented from a number of multimedia benchmarks and a computationally intensive scientific kernel. We explore the complex interactions between programming paradigms, the architectural support at the ISA level and the underlying microarchitecture of these two systems. Our long term goal is to evaluate leading media microprocessors as possible building blocks for future high performance systems.

1. Introduction

Multimedia applications are quickly becoming the dominant consumer of computing cycles [12], and there is a correspondingly large research effort to create commodity components designed to efficiently process high-end media applications. Since many scientific computing and multimedia algorithms share the same computational requirements, it is important for the scientific community to leverage the research efforts of media processor development. 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 increasing clock speeds and multimedia specific enhancements due to their relatively poor performance and high power consumption. This paper investigates two emerging media processors, VIRAM and Imagine, each representing significantly different balances of architectural characteristics, in the context of multimedia applications and scientific computing kernels.

Media applications and certain classes of scientific computations exhibit poor temporal locality and receive little benefit from automatically managed caches of conventional microarchitectures. In addition, a significant fraction of these codes are characterized by predictable fine-grained data-parallelism that can 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 and scientific 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 work examines two general-purpose media processors designed to address the well-known gap between processor and memory performance. 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 present a simple and effective approach for understanding and optimizing vector/stream applications. Performance results are then presented from a number of multimedia benchmarks and a computationally intensive scientific kernel. We explore the complex interactions between programming paradigms, the architectural

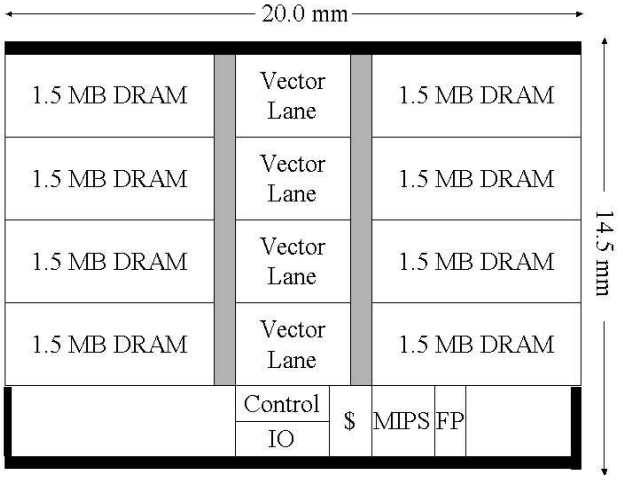

Figure 1. Overview of VIRAM architecture

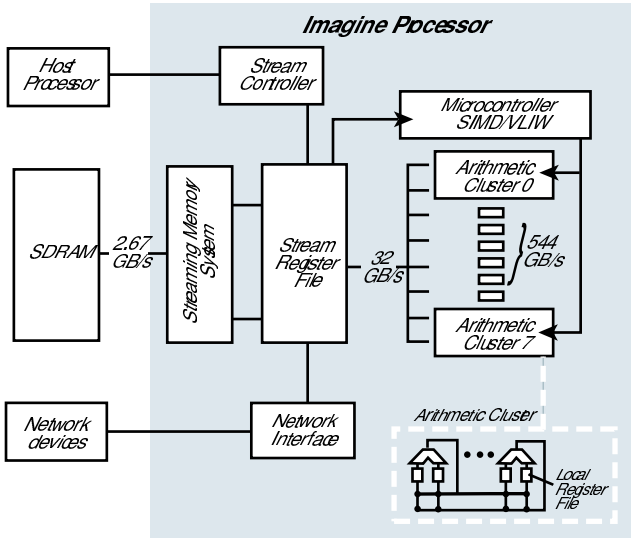

Figure 2. Overview of Imagine architecture

support at the ISA level and the underlying microarchitecture of these two systems. Our long term goal is to evaluate leading media microprocessors as possible building blocks for future high performance systems.

2. VIRAM Architecture

The VIRAM [1] chip, a research architecture being developed at UC Berkeley, is presented in Figure 1. This novel processor is a 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, 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.

3. Imagine Architecture

The Imagine [3] architecture, shown in Figure 2, 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. 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. 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 proces-

	VIRAM	Imagine
Clock Speed	200 MHz	500 MHz
Memory Bandwidth GB/s	6.4 on-chip	2.7 off-chip
Peak GFLOP/s (32 bit)	1.6	20
Peak Flop/Word	1	30
Chip Area	15x20mm	12x12mm
Data Width Support (bits)	64/32/16	32/16/8
Transistors	130 Million	21 Million
Power Consumption	2 Watts	4 Watts

Table 1. Highlights of VIRAM and Imagine

sor widths. 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 (the ratio is 2.5 for SRF references).

4. Programming Environment

The vector programming paradigm [4] of VIRAM is well understood and can leverage off 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.

The Imagine streaming programming paradigm is designed to express the high degree of fine-grained parallelism necessary to effectively utilize the large number of functional units. The relatively new stream programming model, organizes data as streams and expresses all computations as kernels [11]. A stream is an ordered set of records of arbitrary (but homogeneous) data-objects. 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.

For both programming models application performance is highly correlated to the fraction of the application amenable to data parallelism. However, a key distinction between the two platforms 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. Organizing streams as multi-word records increases kernel locality and allows 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.

In VIRAM, applications are coded in C using the vcc 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 underlying architecture (stream controller, micro-controller, etc). 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 work were gathered from the VIRAM and Imagine cycle-accurate simulators.

5. Program Characterization

In this section we describe a methodology for program characterization and performance optimization on the VIRAM and Imagine architectures. A key insight into program classification is to determine if application performance is limited by computational resources or memory bandwidth overhead. Once the bottleneck source is isolated, various optimization schemes can be applied to improve performance.

In a typical data-parallel execution of a vector or stream application data elements are first loaded from the memory in groups (called chunks), processed by the arithmetic units, and written back out to memory. The optimal chunk granularity is generally determined by the underlying hardware system and software infrastructure. This kernel-memory

	VIRAM		Imagine	
	T_k	T_m	T_k	T_m
Unoptimized (cycles)	114	95	2153	1167
Optimized (cycles)	108	17	1147	1165
Chunk Size (elements)	64		1024	

Table 2. RGB→YIQ kernel (T_k) and memory (T_m) cycles before and after optimization

pattern repeats itself, one chunk at a time, until all data elements have been appropriately processed. We define T_k as the time taken by the functional units to operate on one chunk, and T_m as the load/store memory time associated with each chunk. Note that the total time required to fully execute one chunk may be more than $max(T_m, T_k)$, due to idle cycles resulting from an incomplete overlap between computation and memory operations.

Using T_k and T_m we can now classify applications into two categories. Programs with $T_k > T_m$ are characterized as *computation-bottleneck* applications. Here the computational kernel overhead dominates causing the memory subsystem to wait, and thus not fully utilizing the memory bandwidth potential. Conversely, *memory-bottleneck* applications, characterized by $T_m > T_k$, cause the arithmetic units to stall waiting for memory transfers.

Once the bottleneck source has been determined, various optimization strategies can be used to increase the performance of the slower execution component. For example, the kernel computation may be performing at low efficiency due to poor scheduling of the ALUs. In this case, restructuring the computations using techniques such as loop unrolling or software pipelining can significantly improve performance. Similarly, the memory system may not be achieving peak bandwidth due to non-unit data access patterns. For certain applications, it may be possible to reorder the data structures and/or computational patterns to increase unit-stride access, thereby improving memory performance.

5.1. RGB→YIQ Optimization

To demonstrate our characterization approach and ensuing optimization strategies, we investigate the RGB→YIQ color-conversion application from the EEMBC [2] benchmark suite. Table 2 presents the kernel and memory cycle counts (T_k, T_m) of RGB→YIQ using the *csmall.ppm* data set. For the VIRAM implementation Table 2 indicates that computation is the bottleneck, however a more detailed analysis reveals that the low computational performance is caused by the ALUs inefficient interactions with the memory system.

The poor VIRAM memory performance for the unoptimized RGB→YIQ (approximately 1/8 of peak bandwidth) is due to two main reasons. The first inefficiency is attributed to loading bytes into words. The VIRAM memory system can deliver a peak of 32 bytes per cycle; however there are only 8 vector lanes for word-mode operations (vpw=32 bits). This discrepancy (32 vs. 8) is responsible for reducing memory performance by a factor of 4. The second source of memory degradation is due to the strided data access pattern. VIRAM can only generate 4 addresses per cycle, independent of the data width, so for 64-bit values there is sufficient address generation to load or store a value each cycle. However when operating in word-mode using 8 (32-bit) lanes, a factor of two is lost in performance since only 4 of the 8 outstanding lane addresses can be generated each cycles.

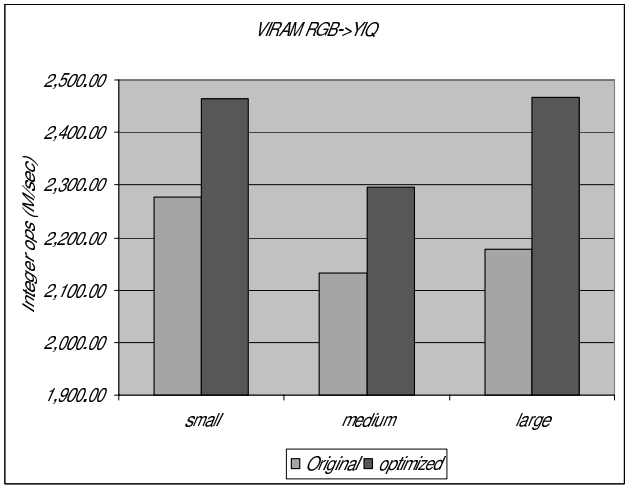

Figure 3. VIRAM RGB→YIQ conversion

The RGB→YIQ kernel was optimized in VIRAM by replacing strided memory access with unit strides, causing T_m to improve by more than a factor of 5 (Table 2). This was done by loading the input vector (in unit stride) into a temporary register and performing an in-register shuffle to generate the required format. As a result, even though the volume of computation increased, improving the memory access pattern caused the computation to be scheduled in a more efficient manner, thus improving ALU utilization and overall kernel performance as seen in Figure 3 (using *csmall.ppm, cmedium.ppm* and *clarge.ppm* date sets).

The T_k and T_m characteristics of the Imagine RGB→YIQ implementation are presented in Table 2, indicating that computation is cause of the performance bottleneck. A more detailed analysis shows that the computational limitation is due to poor ALU utilization; and not functional unit saturation. This can be seen by examining the VLIW schedule, which reveals many unused cycles between useful instructions in the unoptimized code. Applying software pipelining to the loop causes the VLIW instructions to be packed more densely, thus reducing the total number of computational cycles (T_k) by almost a fac-

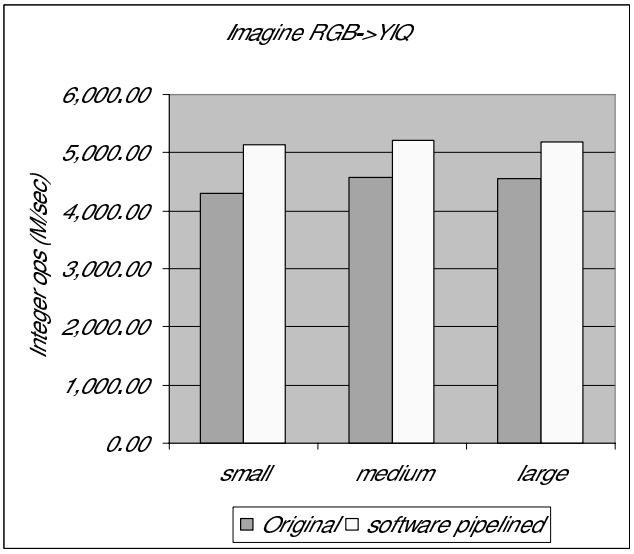

Figure 4. Imagine RGB→YIQ conversion

tor of two as seen in Table 2. However, Figure 4 shows that only a limited improvement is achieved in the overall performance. This is due to the saturation of the off-chip memory system caused by the dense VLIW schedule of the optimized RGB→YIQ implementation. These examples demonstrate that although performance is dependent on the complex interaction between the memory system and ALU utilization, understanding and optimizing stream/vector applications can be approached through our simple methodology.

6. Benchmarks

Table 3 presents the benchmarks used in our study. Vector-add is a commonly used microbenchmark characterized by low computational rate relative to memory requirements. The next four kernels are taken from EEMBC [2] suite, which represents a collection of benchmarks from different application areas for embedded processors. Finally we present QRD, a scientific kernel which performs the Householder QR factorization of complex matrices. All of the chosen benchmarks are amenable to some degree of vectorization or streaming, for a reasonable comparison between the two architectures.

6.1. Media Benchmark Results

Figure 5 presents performance results for both the computation rate (GOPS) and bandwidth (GBps) of our media benchmark set. These codes were developed from scratch on Imagine, whereas on VIRAM we optimized previous implementations [12]. Notice that as expected relative benchmark performance in GOPS is correlated to the attainable

Benchmark	Datawidth (VIRAM/Imagine)	Remarks
Vector-Add	32/32 bits	$c[i] = a[i] + b[i]$
RGB→YIQ	32/32 bits	Color-conversion
RGB→CMYK	16/8 bits	Color-conversion
GreyFilter	16/32 bits	3x3 Convolution
Autocorrelation	16/32 bits	Dot-products
QRD	32/32 bits	Complex QR

Table 3. Benchmark Overview

memory bandwidth. Our first microbenchmark, vector-add (using 64K elements), has only one computation per element: thus performance is limited by memory speed. This explains why memory bandwidth is close to peak, while the computation rate is rather low. The next two color conversion benchmarks, RGB→YIQ and RGB→CMYK (both using *cmedium.ppm*), show Imagine with a significant performance advantage and demonstrate the difference in the attainable main memory bandwidth of the two systems. Imagine effectively uses twice the memory bandwidth of VIRAM due its relatively large processing power. Additionally, the VIRAM performance of RGB→CMYK is also limited since it cannot perform the 8-bit operations required for this benchmark, and is therefore restricted to using 16-bit data. Note that the VIRAM ISA does support 8-bit data types, however this feature is not included in the first hardware implementation. Imagine, on the other hand, can process 8-bit data streams with explicit programmed data subdivision.

The Grey-filter benchmark was the most challenging to implement using Imagine's streaming programming paradigm. The Grey-filter algorithm involves sliding a 3x3 window across the entire image and is difficult to express in a one-dimensional streaming representation. However, we were able to achieve high performance on the VIRAM version using an optimized unit-stride vector implementation. We expect to improve Imagine's performance through sophisticated stream programming.

Finally, Figure 5 shows the performance of an autocorrelation benchmarks using three different data sets. This code, unlike our previous benchmarks, is characterized by short vector/stream lengths. Notice that performance is particularly poor for the "pulse" dataset whose vector/stream lengths are the shortest of the three sets. In this example, Imagine's performance is limited by relatively expensive host transfer latency of stream instructions, which cannot be amortized during short stream computations. Therefore VIRAM has an advantage in this case due to its ability to efficiently process short vector operations.

It is important to note that the media benchmarks presented are not fully representative of streaming applications. Additionally, they only represents a small kernel of larger real-world applications currently being run on media de-

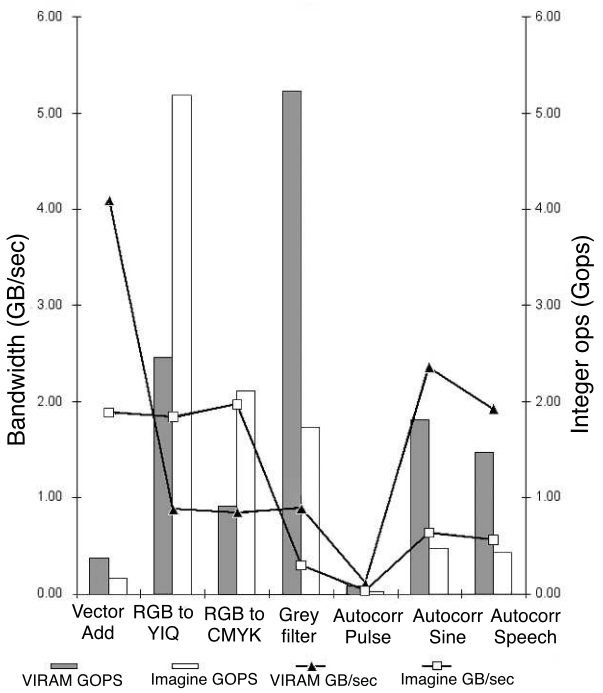

Figure 5. Media Benchmark Performance on VIRAM and Imagine

vices. Furthermore, this benchmark set does not stress the memory systems in complex strided or indexed patterns. However, EEMBC is a well-known benchmark set for embedded processors and a good starting point for our architecture evaluation.

6.2. Complex QR Decomposition

QRD is an important scientific kernel with a significantly higher computational intensity than our set of media benchmarks. 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 [14] operations of matrix-vector multiplication and outer product updates.

6.2.1 VIRAM and Imagine Implementation

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 [6], 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 [9] 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.

Matrix	Performance	VIRAM	Imagine
MITRE 192x96 complex	% of Peak	34.1%	65.5%
	Cycles	5,188,817	712,770
	MFlop/s	546	13,100

Table 4. QRD on VIRAM and Imagine

The VIRAM implementation is a port of the CLAPACK [5] routine CGEQRF and its associated BLAS [14] 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 [4] 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 5.1. For instance, SAXPY version of matrix-vector multiply would do considerably better than the dot-product version [13], for matrices stored in column-major order. This is because the latter implementation requires strided accesses, in addition to the expensive reductions necessary for computing the sum.

The Imagine implementation described in [15] 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 needs to be performed as each iteration of the process requires smaller and smaller matrices.

6.2.2 QRD 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 [7], is shown in Table 4. 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 [15], an improvement of almost 24x in raw processing power over VIRAM . These results demonstrate the considerable performance can be obtained on Imagine on streaming computations with high operations per memory access.

7. Conclusions

In this work we examined two emerging media processors designed to address the well-known gap between processor and memory performance. First, we presented a simple and effective approach for understanding and optimizing the performance of vector/stream applications. A number of multimedia benchmarks were then presented and performance was compared across the two architectures. We found that VIRAM needs more processing power in order to fully exploit its large on-chip bandwidth. Imagine, on the other hand, has extremely powerful functional units but is constrained by the 2.7 GB/s off-chip memory bandwidth. VIRAM is therefore better suited for applications with a relatively low ratio of operations per memory access, while Imagine excels for computationally intensive problems. This was demonstrated by the QRD scientific kernel, where Imagine achieved an impressive 13GFlop/s, due to the high volume of operations per memory access.

A key difference between the two technologies is Imagine's ISA support for multi-word records; while VIRAM is restricted to native vector instructions which only operate on basic-data types. 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 computing community. Brook [8] and SteamIt [16] are two examples of recently proposed high-level streaming languages that attempt to increase programmer productivity while achieving high performance.

Future plans include examining a broader scope of application codes, as well as validating our results on real hardware as it becomes available. We also plan to evaluate more complex data-parallel systems such as the Streaming Supercomputer [8] and the DIVA architecture [10]. Our long-term goal is to evaluate these technologies as building blocks for future high-performance multiprocessor systems.

Acknowledgments

The authors would like to thank Parry Husbands for his insight into the QRD algorithm, and Arin Fishkin for her artistic contribution. This work was supported in part by the Laboratory Directed Research and Development Program of the Lawrence Berkeley National Laboratory (supported by the U.S. Department of Energy under contract number DEAC03-76SF00098).

References

[1] *The Berkeley Intelligent RAM (IRAM) Project*. Univ. of California, Berkeley, http://iram.cs.berkeley.edu.

[2] *EEMBC (EDN Embedded Microprocessor Benchmark Consortium) Benchmark Suite*. http://www.eembc.org.

[3] *The Imagine Project*. Stanford University, http://-cva.stanford.edu/imagine/.

[4] *Maximizing CRAY T90/J90 Application Performance*. Scientific Computing at NPACI (SCAN), Volume 3 Issue 15: July 21, 1999.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, D. J. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorenson. *LAPACK Users Guide. Third Edition*. Society for Industrial and Applied Mathematics, 2000.

[6] C. Bischof and C. Van Loan. Basic linear algebra subprograms for FORTRAN usage. *SIAM J. Scientific and Statistical Computing.*, 8(1):2–13, 1987.

[7] K. Cain, J. Torres, and R. Williams. Real-time space-time adaptive processing benchmark. *MITRE Tech. Rep.*, (MTR96B021), Feb 1997.

[8] W. Dally, P. Hanrahan, and R. Fedkiw. A streaming supercomputer. *Whitepaper*, Sep 2001.

[9] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins Univ Press., 1996.

[10] 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, and J. Shin. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. *Proc. of SC99*, 1999.

[11] 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.

[12] C. Kozyrakis. *Scalable Vector Media-Processors for Embedded Systems*. PhD Thesis, Univ. of California, Berkeley, 2002.

[13] 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.

[14] C. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Soft*, 5, 1979.

[15] P. Mattson. *Programming System for the Imagine Media Processor*. Ph.D. Thesis, Stanford University, 2002.

[16] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. *Computational Complexity Journal*, pages 179–196, 2002.