Exploiting On-Chip Memory Bandwidth in the VIRAM Compiler

David Judd, Katherine Yelick, Christoforos Kozyrakis, David Martin, and David Patterson

Computer Science Division
University of California *
Berkeley, CA 94720, USA
{dajudd,yelick,kozyraki,dmartin,pattrsn}@cs.berkeley.edu,
WWW Page: http://iram.cs.berkeley.edu/

Abstract. Many architectural ideas that appear to be useful from a hardware standpoint fail to achieve wide acceptance due to lack of compiler support. In this paper we explore the design of the VIRAM architecture from the perspective of compiler writers, describing some of the code generation problems that arise in VIRAM and their solutions in the VIRAM compiler. VIRAM is a single chip system designed primarily for multimedia. It combines vector processing with mixed logic and DRAM to achieve high performance with relatively low energy, area, and design complexity. The paper focuses on two aspects of the VIRAM compiler and architecture. The first problem is to take advantage of the on-chip bandwidth for memory-intensive applications, including those with non-contiguous or unpredictable memory access patterns. The second problem is to support that kinds of narrow data types that arise in media processing, including processing of 8 and 16-bit data.

1 Introduction

Embedded processing in DRAM offers enormous potential for high memory bandwidth without high energy consumption by avoiding the memory bus bottlenecks of conventional multi-chip systems [FPC+97]. To exploit the memory bandwidth without expensive control and issue logic, the IRAM project at U.C. Berkeley is exploring the use of vector processing capabilities in a single-chip system called VIRAM, designed for multimedia applications [PAC+97]. Studies of hand-coded VIRAM benchmarks show that performance on a set of multimedia kernels exceeds that of high-end DSPs and microprocessors with media-extensions [Koz99,Tho00]. In this paper, we demonstrate that a vectorizing compiler is also capable of exploiting the vector instructions and memory bandwidth in VIRAM.

* This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract DABT63-96-C-0056, by the California State MICRO Program, and by the Department of Energy. The VIRAM compiler was built using software provided by Cray, Inc.

The technology for automatic vectorization is well understood within the realm of scientific applications and supercomputers. The VIRAM project leverages that compiler technology with some key differences that stem from differences in the application space of interest and in the hardware design: the VIRAM processor is designed for multimedia benchmarks and with a primary focus on power and energy rather than high performance. There is some overlap between the algorithms used in multimedia and scientific computing, e.g., matrix multiplication and FFTs are important in both domains. However, multimedia applications tend to have shorter vectors, and a more limited dynamic range for the numerical values, which permits the use of single-precision floating-point as well as integer and fixed-point operations on narrow data types, such as 8, 16, or 32 bit values. VIRAM has features to support both narrow data types and short vectors.

2 Overview of the VIRAM Architecture

2.1 The Instruction Set

The VIRAM instruction set architecture (ISA) [Mar99] extends the MIPS ISA with vector instructions. It includes integer and floating-point arithmetic operations, as well as memory operations for sequential, strided, and indexed (scatter/gather) access patterns. The ISA specifies 32 vector registers, each containing multiple vector elements. Each vector instruction defines a set of operand vectors stored in the vector register file, a vector length, and an operation to be applied element-wise to the vector operands. Logically, the operation described by an instruction may be performed on all the vector elements in parallel. Therefore, we use the abstract notion of a *virtual processor* in which there is one simple processor per vector element that executes the operation specified by each vector instruction.

The maximum number of elements per vector register is determined by two factors: the total number of bits in a register and the width of the elements on which operations are being performed. For example, in the VIRAM processor a vector register holds 2K bits, which corresponds to 32 64-bit elements, 64 32-bit elements, or 128 16-bit elements. The VIRAM ISA supports arithmetic and memory operations on these three data widths. The bit width of the elements is known as the *virtual processor width* (VPW) and may be set by the application software and changed as different data types are used in the application.

Apart from narrow data types, multimedia applications frequently use fixed-point and saturated arithmetic. Fixed-point arithmetic allows decimal calculations within narrow integer formats, while saturation reduces the error introduced by overflow in signal processing algorithms. The VIRAM architecture supports both features with a set of vector fixed-point add, multiply, and fused multiply-add instructions. Programmable scaling of the multiplication result and four rounding modes are used to support arbitrary fixed-point number formats. The width of the input and output data for multiply-add are the same for these operations, hence all operands for this instruction can be stored in regular vector

registers. There is no need for extended precision registers or accumulators, and this simplifies the use of these instructions. The maximum precision of calculations can be set by selecting the proper virtual processor width.

To enable efficient vectorization of conditional statements, the ISA includes a vector flag register file with 32 registers. Each register consists of a bit vector with one bit per vector element, which may be applied as a mask to the majority of vector operations. The same flag registers are used to support arithmetic exceptions, as well as software-controlled speculation of both load and arithmetic operations.

2.2 The VIRAM Processor

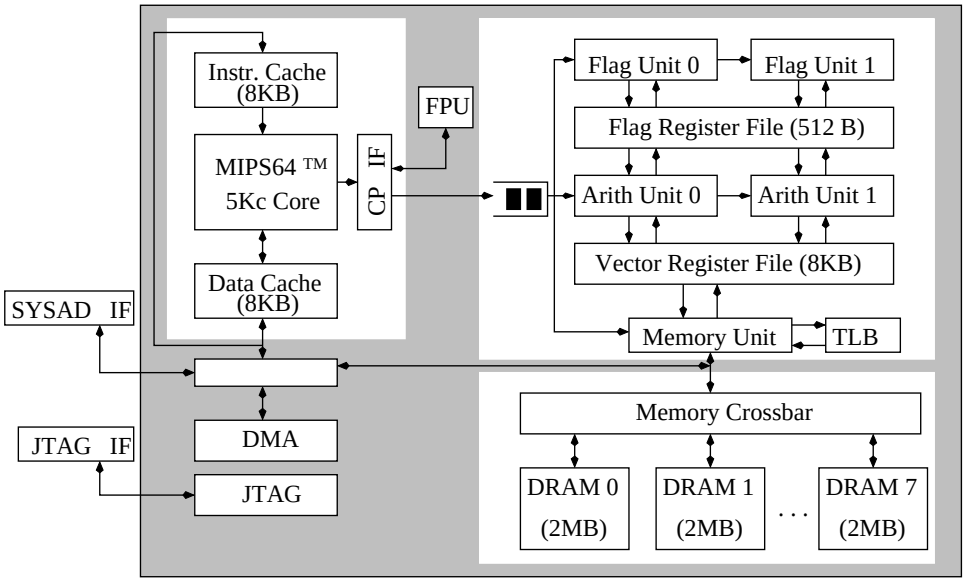

Fig. 1. The Block Diagram of the VIRAM Processor.

The VIRAM processor chip is an implementation of the VIRAM architecture designed at U.C. Berkeley [KGM+00]. Its block diagram is presented in Figure 1. It includes a simple in-order (scalar) MIPS processor with first level caches and floating-point unit, a DMA engine for off-chip access, an embedded DRAM memory system, and a vector unit which is managed as a co-processor. Both the vector and scalar processors are designed to run at 200 MHz using a 1.2V power supply, with a power target of 2 Watts [Koz99].

The vector unit of the VIRAM processor includes one floating-point and two integer arithmetic units. Each arithmetic unit contains a 256-bit datapath, which can be used to execute 4 64-bit operations, 8 32-bit operations, or 16 16-bit

operations simultaneously. Thus, the virtual processors that one may imagine acting on an entire vector register are implemented in practise by a smaller set of vector *lanes*, which execute a vector operation by computing a subset of the elements in parallel in each clock cycle. With 256 bits datapath width, the vector unit has 4 64-bit lanes, which can be viewed as 8 32-bit lanes or (for integer operations) 16 16-bit lanes.The ISA allows for 32 and 64-bit floating-point as well as 8, 16, 32, and 64-bit integer operations. The VIRAM processor supports only 32-bit floating-point and 16-, 32-, and 64-bit integer operations.[1] VIRAM's peak performance is 1.6 GFLOPS for 32-bit floating-point, 3.2 GOPS for 32-bit integer operations, and 6.4 GOPS for 16-bit integer operations.

The vector unit uses a simple, single-issue, in-order pipeline structure for predictable performance. The combination of pipelined instruction startup and chaining, the vector equivalent of operand forwarding, enables high performance even with short vectors. To avoid large load-use delays due to the latency of DRAM memory, the worst-case latency of a on-chip DRAM access is included in the pipeline and the execution of arithmetic operations is delayed by a few pipeline stages. This hides the latency of accessing DRAM memory for most common code cases.

There are 16 MBytes of on-chip DRAM in the VIRAM processor. They are organized in 8 independent banks, each with a 256-bit synchronous interface. A DRAM bank can service one sequential access every 6.5ns or one random access every 25ns. The on-chip memory is directly accessible from both the scalar and vector instructions using a crossbar interconnect structure with peak bandwidth of 12 GBytes/s. Instruction and data caches are used for scalar accesses, while references from the vector unit are served directly by the DRAM macros. The memory unit in the vector coprocessor can exchange 256 bits per cycle with the DRAM macros for sequential accesses, or up to four vector elements per cycle for strided and indexed accesses. It also includes a multi-ported TLB for virtual memory support.

3 Compiler Overview

The VIRAM compiler is based on the Cray vectorizing compiler, which has C, C++ and Fortran front-ends and is used on Cray's supercomputers. In addition to vectorization, the compiler performs standard optimizations including constant propagation and folding, common subexpression elimination, in-lining, and a large variety of loop transformations such as loop fusion and interchange. It also performs outer loop vectorization, which is especially useful for extracting large degrees of parallelism across images in certain multimedia applications. Pointer analysis in C and C++ are beyond the scope of this paper, so we use the Cray compiler strategy of requiring "restrict" pointers on array arguments to indicate they are unaliased.

[1] The fused multiply-add instructions in the VIRAM ISA are implemented in the VIRAM processor for fixed-point operations but not floating-point. These instructions are not targeted by the compiler and will therefore not be considered here.

The strategy in this project was to re-use as much compiler technology as possible, innovating only where necessary to support the VIRAM ISA. The Cray compiler has multiple machines targets, including vector machines like the Cray C90 and parallel machines like the T3E, but it did not support the MIPS architecture, so our first step was to build a MIPS back-end for scalar code and then a VIRAM code generator for vectorized code. Differences in VIRAM and Cray vector architectures lead to more interesting differences:

1. VIRAM supports narrow data types (8, 16, and 32 bits as well as 64 bits). Cray vector machines support 32 and 64 bit types, but, historically, the hardware ran 32-bit operations at the same speed as 64-bit, so there was no motivation to optimize for narrower types. In VIRAM using narrow data types leads to higher peak performance, as each 64-bit datapath can execute multiple narrower operations in parallel.
2. In both Cray and VIRAM architectures, conditional execution is supported by allowing a vector instruction to be computed with a set of flags (or masks) that effectively turns off the virtual processors at those positions. VIRAM treats flags as 1-bit vector registers, while the Cray machines treat them as 64 or 128-bit scalar values.
3. VIRAM has special support for fixed-point arithmetic, saturation and various rounding modes for integer operations, which are not provided on Cray machines.
4. VIRAM allows for speculative execution of vector arithmetic and load instructions, which is particularly important for vectorizing loops with conditions for breaking out of the loop, e.g., when a value has been found.

This list highlights the differences in the architectures as they affect compilation. In this paper we focus mainly on the first of these issues, i.e., generation of code for narrow data widths, which is critical to the use of vector processing for media processing. The last two features, fixed-point computations and speculative execution, have no special support in our compiler, and while the flag model resulted in somewhat simpler code generation for VIRAM, the difference is not fundamental. In VIRAM the vector length register implicitly controls the set of active flag values, whereas the Cray compiler must correctly handle trailing flag bits that are beyond the current vector length.

4 On-Chip Memory Bandwidth

The conventional wisdom is that vector processing is only appropriate for scientific computing applications on high-end machines that are expensive in part because of their SRAM-based, high bandwidth, memory systems. By placing DRAM and logic on a single chip, VIRAM demonstrates that vector processing can also be used for application domains that demand lower cost hardware. In this section, we use a set of different implementations of the dense matrix-vector multiplication (MVM) benchmark to explore the question of how well VIRAM

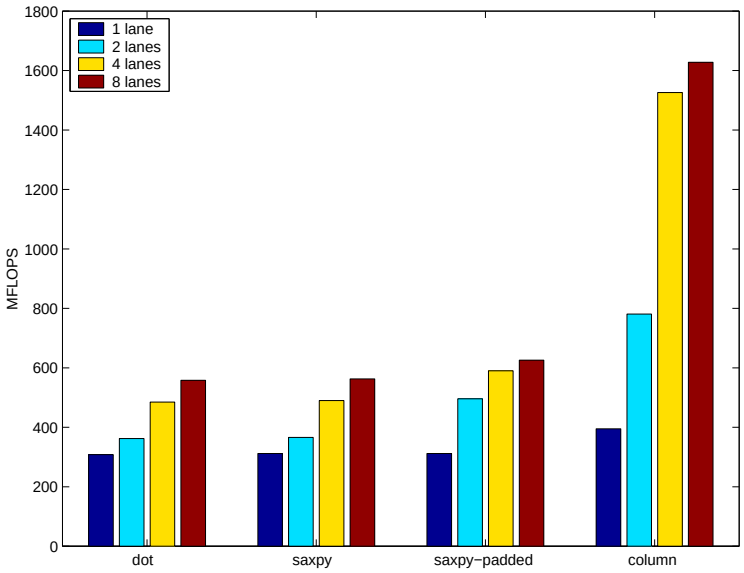

Fig. 2. Performance of 64×64 Single-precision Floating-point Matrix-Vector Multiplication on the VIRAM Processor.

supports memory-intensive applications. Although MVM is a simple compilation problem, it does not perform well on conventional microprocessors due to the high memory bandwidth requirements. There are only two floating-point operations per matrix element.

When the matrix uses a row-major layout, matrix vector multiplication can either be organized as a set of dot products on the rows of the matrix or a set of saxpy's on the columns of the matrix. The dot products have an advantage in using unit stride access (sequential) to the matrix, while saxpy's require strided accesses. Strided and indexed vector accesses are slower than unit stride, because they can only fetch four elements per cycle, regardless of the virtual processing width. A unit stride load can always fetch 256 bits per cycle, which is eight 32-bit elements for example. In addition, strided and indexed accesses may cause DRAM bank conflicts which stall the vector unit pipeline. On the other hand, the dot product involves a reduction that is less efficient, because it involves operations on short vectors near the end of the reduction. For very long vectors, the final stages of the reduction are amortized over more efficient full vector operations, so the impact on efficiency is negligible.

Figure 2 shows the performance in MFLOPS of single-precision floating-point matrix vector multiplication routine on a 64×64 matrix. The numbers are taken for the VIRAM processor implementation by varying the number of 64-bit lanes from 1 to 8. Our performance results are produced using a simulator that gives a cycle-accurate model of the VIRAM vector unit. The first of these groups is a dot-product implementation using unit stride. Not surprisingly, with only 64

elements per vector, efficiency is somewhat low due the percentage of time spent on reductions of short vectors. The second version shows a saxpy implementation, where the accesses to the matrix are the stride of the matrix dimension (64). If one pads the array with an extra element, giving it a prime stride of 65, that reduces memory bank conflicts, resulting in the third group of numbers. Finally, if we use a column-major layout for the matrix, a saxpy implementation can be used with unit stride, which produces the best performance. We note that while is may be attractive to consider only using the column-based layout, another common kernels used in multimedia application is vector-matrix multiplication, which has exactly the opposite design constraints, so a row layout is preferred there.

As indicated by this data, an 8-lane vector unit does not perform well with only 8 banks. Even with 4 lanes there are significant advantages to having more banks for the saxpy implementations with the row-major matrix layout. More DRAM banks reduce the likelihood of bank conflicts for strided memory accesses. A similar effect can be achieved by organizing the existing banks in a hierarchical fashion with sub-banks, which would allow overlapping of random accesses to a single bank [Y+97]. Unfortunately, hierarchical DRAM bank technology was not available to us for the VIRAM processor, but we expect it to be available in the next generation of embedded DRAM technology.

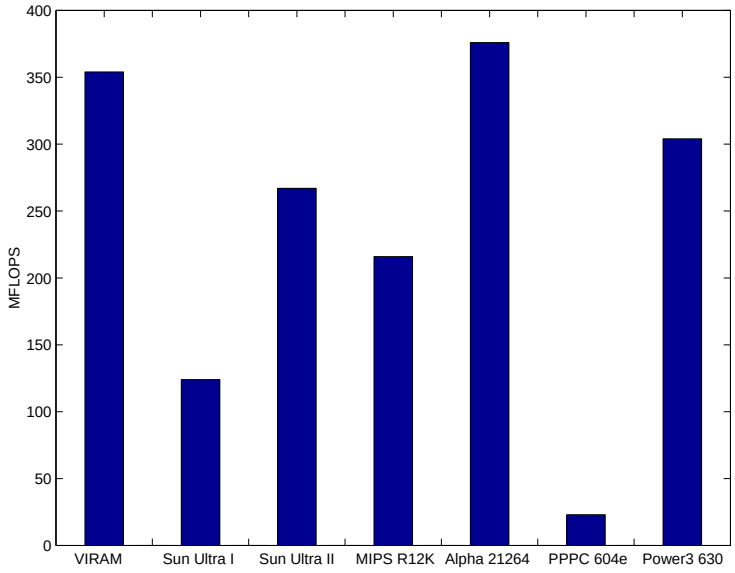

Fig. 3. Performance of 100×100 Double-precision Floating-point Matrix-Vector Multiplication.

Nevertheless, the performance is impressive even for the 4-lane, 8 bank VIRAM design when compared to conventional microprocessors with higher clock rates, higher energy requirements, more total area per microprocessor, and many more chips in the system for caches and main memory. Figure 3 presents the performance of the hand-coded vendor implementation of 100×100 double-precision MVM as reported in the LAPACK manual for a set of high performance server and workstation processors [ABB+99]. A column-major layout is used with all of them. The VIRAM processor does not support double-precision operations on its floating-point datapaths because they are of no practical use for multimedia applications. Still, we were able to calculate the performance of a four lane VIRAM processor with support for double-precision using a simulator.

From the six other processors, only the hand-coded Alpha 21264 outperforms compiled code on VIRAM by merely 6%. The rest of the processors perform 1.3 to 15 times worse, despite their sophisticated pipelines and SRAM based cache memory systems. Note that for larger matrices VIRAM outperforms all these processors by larger factors. The performance of server and workstation processors is reduced for larger matrices as they no longer fit in their first level caches. On the other hand, the vector unit of VIRAM accesses DRAM directly and incurs no slow-down due to caching effects, regardless of the size of the matrix. In addition, multiplication of larger matrices leads to operations on longer vectors, which amortize better the performance cost of the final stages for reduction operations in the saxpy implementation. Hence, the performance of the VIRAM processor actually increases with the size of the matrix.

5 Narrow Data Types

One of the novel aspects of the VIRAM ISA is the notion of variable width data which the compiler controls by setting the virtual processor width (VPW). The compiler analyzes each vectorizable loop nest to determine the widest data width needed by vector operations in the loop and then sets the VPW to that width. The compiler uses static type information from the variable declaration rather than a more aggressive, but also more fragile, variable width analysis [SBA00].

The process of computing the VPW is done with multiple passes over a loop nest. On the first pass, the VPW is assumed to be 64 bits, and the loop nest is analyzed to determine whether it is vectorizable. At the same time, the widest data type used within the loop is tracked. If the maximum data width is determined to be narrower than 64 bits, vectorization is re-run on the loop nest with the given estimate of VPW (32 or 16 bits). The reason for the second pass is that narrowing the data width increases the maximum available vector length, and therefore allows more loops to be executed as a single sequence of vector instructions without the strip-mining loop overhead.

The above strategy works well for 32-bit data types (either integer or floating-point) but does not work well for 16 bits in standard C. The reason is that the C language semantics require that, even if variables are declared as 8-bit characters or 16-bit shorts, most operations must be performed as if they were in 32 bits.

To enforce these semantics, the compiler front-end introduces type casts rather aggressively, making it difficult to recover information about narrower types in code generation. Our initial experiments found that only the most trivial of loops, such as an array initialize, would run at a width narrower than 32 bits. As a result, we introduced a compiler flag to force the width to 16 bits; the compiler will detect certain obvious errors such as the use of floating-point in 16 bit mode, but trusts the programmer for the bit-width of integer operations.

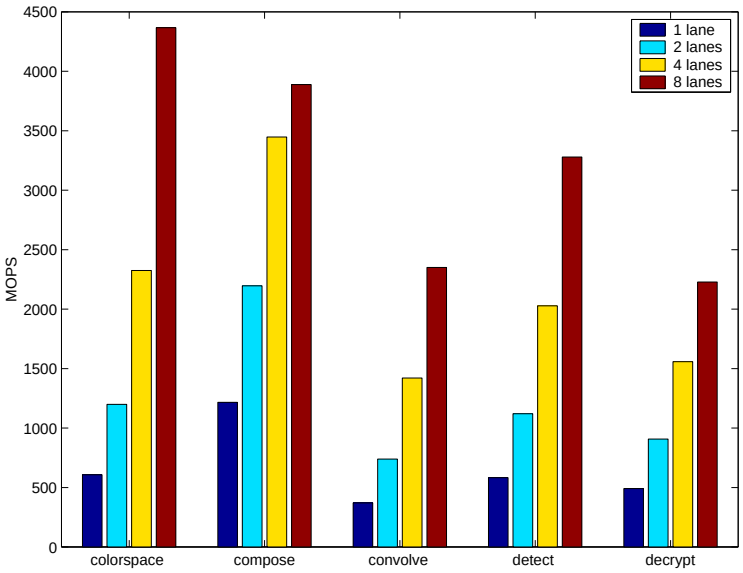

Fig. 4. Performance of Integer Media Kernels on VIRAM.

Figures 4 and 5 shows the performance of several media processing kernels on a VIRAM configuration with 16 memory banks and varying numbers of 64-bit lanes from 1 to 8. Figure 4 presents integer benchmarks taken from the UTDSP benchmark suite [LS98], while Figure 5 includes single-precision floating-point kernels.

- `colors` (colorspace conversion) is a simple image processing kernels that converts from the RGB to the YUV format. It reads and writes 8-bit data and operates internally on it as 16-bit data. Strided memory operations are convenient for selecting pixels of the same color that are interleaved in RGB format in the image. Loads of 8-bit data while the VPW is set to 16 bits, fetches each 8-bit value into a single 16-bit vector element, without any shuffling or expanding of data needed.
- `compose` (image composition) is similar to `colors` in the data types, but can be performed using unit stride memory accesses only.

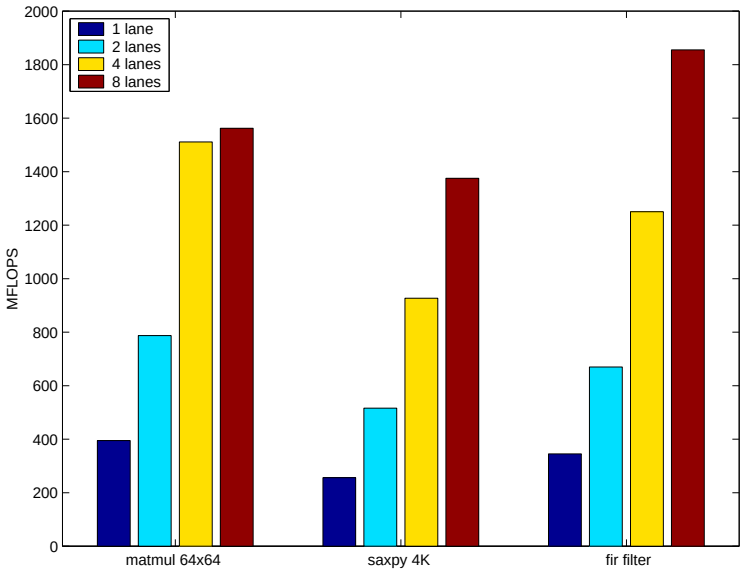

Fig. 5. Performance of Single-Precision Floating-point Media Kernels on VIRAM.

- `convlv` is an image convolution, which like `colorspace` performs a 16-bit integer computation on images stored in RGB format using strided memory accesses. Each output pixel is computed by multiplying and summing the pixels in a 3x3 region. The source code contains 4 nested loops, and it is critical that the compiler vectorize an outer loop to obtain reasonable vector lengths. By default, the compiler merges the middle two loops and vectorizes them while unrolling the innermost one. The performance shown here improves on that default strategy by unrolling the two inner loops in source code, while the compiler vectorizes the 2nd loop that goes over rows of the image.
- `detect` is an edge detection algorithm for images and `decrypt` performs IDEA decryption. Both of these use a mixture of 16-bit and 32-bit integer operations.
- `FIR` is an FIR filter, `saxpy1` and `saxpy2` are, respectively, 64 element and 1024 element saxpy's. `Matmul` is a 64x64 matrix multiplication. All of these use 32-bit floating-point computations.

Although hand-optimized VIRAM code is not available for all of these kernels in this form, two floating-point kernels are available and are quite close in performance. The hand-coded performance numbers for a 4-lane configuration with 8 memory banks are: 720 MFLOPS for `saxpy2` and 1,580 MFLOPS for `matmul`. Note that the performance in both cases depends on the size of the input data. The two versions of saxpy are included to show how performance improves with longer application level arrays, because they better amortize the

time it takes to fill and drain the long vector pipeline at the beginning and at the end of the kernel. This shows again an advantage VIRAM for computations that operate on large images or matrices. VIRAM does not depend on caches, hence it performance often increases rather than decreases with increased data size.

The performance of kernels with 16-bit operations is limited primarily by the number of elements that can be fetched per cycle for non-unit stride accesses. The VIRAM processor fetches one element per lane for indexed and strided accesses, while a 64-bit lane can execute 4 16-bit arithmetic operation simultaneously. This mismatch exists due to the high area, power, and complexity cost of generating, translating, and resolving for bank conflicts more than one address per lane per cycle for strided and indexed accesses. Unit stride accesses, due to their sequential nature, can fetch multiple elements with a single address for all lanes. Bank conflicts are also responsible for some performance degradation.

6 Related Work

The most closely related compiler effort to ours is the vectorizing compiler project at the University of Toronto, although it does not target a mixed logic and DRAM design [LS98]. Compilers for the SIMD microprocessor extensions are also related. For example, Larsen and Amarsinghe present a compiler that uses *Superword Level Parallelism* (SLP) for these machines; SLP is identical to vectorization for many loops, although it may also discover vectorization across statements, which typically occurs if a loop has been manually unrolled.

Several aspects of the VIRAM ISA design make code generation simpler than with the SIMD extensions [PW96,Phi98]. First, the VIRAM instructions are independent of the datapath width, since the compiler generates instructions for the full vector length, and the hardware is responsible for breaking this into datapath size chunks. This simplifies the compilation model and avoids the need to recompile if the number of lanes changes between generations. Indeed, all of the performance numbers below use a single executable when varying the number of lanes. Second, VIRAM has powerful addressing modes such as strided and indexed loads and stores that eliminate the need for packing and unpacking. For example, if an image is stored using 8-bits per pixel per color, and the colors are interleaved in the image, then a simple array expression like `a[3*i+j]` in the source code will result in a strided vector load instruction from the compiler.

7 Conclusions

This paper demonstrates that the performance advantages of VIRAM with vector processing do not require hand-optimized code, but are obtainable by extending the vector compilation model to multiple data widths. Although some programmer-control over narrow data types was required, the programming model is still easy to understand and use. VIRAM performs well and demonstrates scaling across varying numbers of lanes, which is useful for obtaining

designs with lower power and area needs or for high performance designs appropriate for a future generations of chip technology. The compiler demonstrates good performance overall, and is often competitive with hand-coded benchmarks for floating-pointer kernels. The availability of high memory bandwidth on-chip makes a 2 Watt VIRAM chip competitive with modern microprocessors for bandwidth-intensive computations. Moreover, the VIRAM instruction set offers an elegant compilation target, while the VIRAM implementation allows for scaling of computation and memory bandwidth across generations through the addition of vector lanes and memory banks.

Acknowledgments

The authors would like to thank the other members of the VIRAM group and Corinna Lee's group at the University of Toronto who provided the UTDSP benchmarks. We are also very grateful for the support provided by the Cray, Inc. compiler group in helping us use and modify their compiler. IBM Microelectronics provided the manufacturing technology for the VIRAM processor, while MIPS Technologies designed the MIPS processor it includes. Avanti Corp. provided the CAD tools used for the design.

References

[ABB+99] 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*. SIAM, 1999.

[FPC+97] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick. The energy efficiency of IRAM architectures. In *the 24th Annual International Symposium on Computer Architecture*, pages 327–337, Denver, CO, June 1997.

[KGM+00] C.E. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and D. Yelick. VIRAM: A Media-oriented Vector Processor with Embedded DRAM. In *the Conference Record of the 12th Hot Chips Symposium*, Palo Alto, CA, August 2000.

[Koz99] Christoforos Kozyrakis. A media-enhanced vector architecture for embedded memory systems. Technical Report UCB//CSD-99-1059, University of California, Berkeley, July 1999.

[LS98] C.G. Lee and M.G. Stoodley. Simple vector microprocessors for multimedia applications. In *31st Annual International Symposium on Microarchitecture*, December 1998.

[Mar99] D. Martin. *Vector Extensions to the MIPS-IV Instruction Set Architecture*. Computer Science Division, University of California at Berkeley, January 1999.

[PAC+97] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for Intelligent DRAM: IRAM. *IEEE Micro*, 17(2):34–44, April 1997.

[Phi98] M. Phillip. A second generation SIMD microprocessor architecture. In *Proceedings of the Hot Chips X Symposium*, August 1998.

[PW96] A. Peleg and U. Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[SBA00] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, Vancouver, British Columbia, June 2000.

[Tho00] Randi Thomas. An architectural performance study of the fast fourier transform on VIRAM. Technical Report UCB//CSD-99-1106, University of California, Berkeley, June 2000.

[Y+97] T. Yamauchi et al. The hierarchical multi-bank DRAM: a high-performance architecture for memory integrated with processors. In *the Proceedings of the 17th Conf. on Advanced Research in VLSI*, Ann Arbor, MI, Sep 1997.