Efficient FFTs on IRAM

Randi Thomas and Katherine Yelick
Computer Science Division
University of California, Berkeley *

Abstract: Computing Fast Fourier Transforms (FFTs) is notoriously difficult on conventional general-purpose architectures because FFTs require high memory bandwidth and strided memory accesses. Since FFTs are important in signal processing, several DSPs have hardware support for performing FFTs; moreover, some DSPs are designed solely for the purpose of computing FFTs and related transforms. In this paper, we show that the performance of the general-purpose VIRAM processor in computing floating-point FFTs is competitive with that of existing floating point DSPs and specialized fixed-point FFT chips. Furthermore, VIRAM is a complete "system on a chip," and therefore has power, area, and cost advantages over multi-chip systems based on DSPs. VIRAM's advantages result from a combination of hardware technology, instruction set design, and algorithmic techniques. In particular our FFT results are achieved by: an integrated processor-in-memory design that provides high memory bandwidth; vector processing, which provides simple and efficient utilization of the high memory bandwidth; a small amount of ISA support for in-register permutations; and an algorithm tuned to the VIRAM instruction set and implementation.

1 Introduction

Fast Fourier Transforms (FFTs) are critical for many signal processing problems as well as for the increasingly popular multimedia applications that involve images, speech, audio, graphics, or video. Several DSPs offer support to accelerate the computation of FFTs often including hardware to improve the performance of bit-reversals or transpose operations. Of these DSPs, the ones with the best performance are those that are specialized exclusively for computing FFTs and related transforms. The need for such specialization is primarily based on the observation that FFT algorithms have poor temporal and spatial locality, and therefore perform poorly on architectures that employ structures that rely on locality for performance (such as caches and stream-buffers). While the algorithm chosen to compute the FFT may be reorganized to somewhat improve data reuse [FJ98], FFT performance on conventional microprocessors is typically limited by the poor memory bandwidth and high memory latency on these machines.

To address these memory system issues, the IRAM project is exploring an unconventional microprocessor design based on combining logic with embedded DRAM ("Intelligent RAM") to construct a single-chip system designed for low power and high performance on multimedia applications. The Vector IRAM (VIRAM) system adds a vector processor to embedded DRAM in order to produce a low energy, high performance design suitable for the ever-growing market of portable devices [FPC+97]. Kozyrakis gives a more detailed overview of the VIRAM implementation and shows that performance on a set of media kernels exceeds that of high-end DSPs [Koz99]. However, the kernels examined in that paper do not include an FFT, and most of them use primarily unit-stride memory accesses. In this paper we show that the general-purpose VIRAM design is also well-suited to the memory access patterns of the FFT, and that its performance rivals the best performance of special-purpose DSPs for computing FFTs.

Section 2 gives an overview of the VIRAM architecture and discusses the key design features that make it suitable for multimedia processing on small portable devices. Section 3 describes a standard FFT algorithm and Section 4 shows a straightforward vectorization of that algorithm. Section 5 describes several optimizations. The performance, based on simulations, is reported in Section 6; the performance is shown to be comparable to existing DSPs for floating point-based algorithms. Section 7 draws some conclusions and gives an overview of our future plans for this research.

*This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract DABT63-96-C-0056 and the California State MICRO Program. The first author was supported in part by a National Science Foundation fellowship.

2 Overview of VIRAM

By combining a vector processor with embedded DRAM, one potentially exposes two orders of magnitude more memory bandwidth than is available in typical multi-chip systems that are limited by bus bandwidth and pin counts [PAC+97]. To take advantage of that on-chip bandwidth without excessive complexity, area, or power, the VIRAM architecture extends a RISC instruction set with vector processing instructions. VIRAM's general-purpose vector processor provides high performance on computations with sufficient fine-grained data-parallelism. VIRAM utilizes a delayed vector pipeline[1] [Asa98, Koz99] to hide memory latency; consequently there is no need for caches. Instead, VIRAM is built around a banked, pipelined, on-chip DRAM memory that is well-matched to the memory access patterns of a vector processor. Thus the VIRAM architecture conserves area while preserving the low-power benefits of a single chip because it avoids multiple accesses through a memory hierarchy, and because it does not require a high clock rate or the complexity of a superscalar processor. Since one vector instruction initiates a set of operations on an entire vector (64 single precision elements) VIRAM also has more compact instructions and greater code density than the VLIW architectures currently being used in DSPs like TI's TMS320C6000, Motorola/Lucent's StarCore 440, Siemens (Infineon) Carmel, and Analog Device's TigerSHARC. This reduction in code space and the corresponding reduction in instruction fetch bandwidth translate to power and performance advantages.

VIRAM is a complete "system on a chip," and therefore enjoys power, cost and area advantages over multi-chip systems [PAC+97]. In addition to the vector processor and embedded DRAM, VIRAM has a superscalar MIPS core, a memory crossbar, and an I/O interface for off-chip communication. The prototype implementation of VIRAM, currently under development, is designed to run both the vector and scalar processors at 200 MHz with 32 MB of DRAM organized into 16 banks, four 100 MB/s parallel I/O lines, a 1.2V power supply, and a power target of 2 watts [Koz99]. VIRAM has four 64-bit pipelines, called *lanes*, each of which has two integer functional units and one floating point functional unit; each of these functional units supports a multiply-add instruction that can complete in one cycle. To support narrower data widths, each of the 64-bit lanes can be subdivided into two or more *virtual lanes*. Specifically, a 64-bit lane can be divided into two 32-bit virtual lanes (which yields a total of 8 virtual lanes), or into four 16-bit virtual lanes (which yields a total of 16 virtual lanes). The narrower data widths are particularly useful for some DSP and multimedia computations. The number of virtual lanes and the number of functional units determines the maximum number of operations that can be executed in a single cycle in VIRAM. For example, for single precision data there are 8 virtual lanes, and each virtual lane has one floating point functional unit, so 8 floating point operations can execute in one cycle. For 32-bit integers, again there are 8 virtual lanes, but there are two integer functional units per virtual lane, so 16 integer operations can execute in one cycle. Since all the functional units support a multiply-add instruction, if all operations in the above examples were multiply-adds, then the number of operations that can execute in one cycle doubles for both cases.

Using multiply-adds, VIRAM's peak performance is 3.2 GFLOP/s[2] for single precision floating point and 6.4 GOP/s[3] for 32-bit integer operations. Since the VIRAM chip and compiler are still under development, the results in this paper are based on a near cycle-accurate simulator for VIRAM and use hand optimized vector assembly code for the FFT kernel.

Because multimedia applications have a high degree of fine-grained data parallelism (such as parallelism over all pixels in an image) a vector processor is well-suited to many of these applications. FFTs are also data-parallel, although the degree of parallelism depends on the size of the FFT and varies over the course of the algorithm. As we will show, high performance on short vectors (*i.e.* vectors whose element count is less than the maximum) is critical to the performance of FFTs. VIRAM contains several features that make short vector operations much more efficient than in the vector supercomputers of the past, such as Cray's C90 and T90. One such feature is a delayed pipeline organization that helps hide memory latency [Asa98, Koz99]. We will discuss additional VIRAM design support for short vectors as they become relevant to the problem of developing a high performance FFT algorithm.

[1] In such a pipeline the execution of all arithmetic operations is delayed for a fixed number of clock cycles after issue to match the latency of a worst-case memory access, thereby freeing the pipeline's issue stage. In this way the next instruction can be issued, and thus the pipeline does not stall for RAW hazards.

[2] 3.2 GFLOP/s = 8 virtual lanes * 1 floating point functional unit/virtual lane * 2 operations/cycle * 200 Mcycles/second

[3] 6.4 GOP/s = 8 virtual lanes * 2 integer functional units/virtual lane * 2 operations/cycle * 200 Mcycles/second

3 Computing the FFT

The Fourier Transform is a mathematical technique for converting a time-domain function into a frequency spectrum. Given an N-element vector x, its 1D Discrete Fourier Transform is another N-element vector y given by the formula:

$$\forall j \in \{0, 1 \ldots, N-1\} \quad y_j = \sum_{k=0}^{N-1} \omega_N^{jk} x_k,$$

$$\text{where } \omega_N^{jk} = e^{\frac{-2\pi i j k}{N}}$$

N is referred to as the number of *points*, and ω_N^{jk} is the jk^{th} *root of unity*.

The Fast Fourier Transform (FFT) [CT65] takes advantage of algebraic identities to compute the Fourier transform in $O(N \log N)$ steps. The computation is organized into $\log_2 N$ stages (for a *radix 2* FFT). In every stage each point is paired with another, the same computations are performed between the two, and the values are overwritten in the input vector. For example, in the first stage, x_0 and $x_{N/2}$ are paired and the computations are as follows:

$$\begin{aligned} x'_0 &= x_0 + \omega \cdot x_{N/2} \\ x'_{N/2} &= x_0 - \omega \cdot x_{N/2} \end{aligned}$$

where ω is one of the roots of unity. We will call this sequence of computations the *basic computation*. In a complex FFT, both the x_i's and the roots of unity are complex numbers; recall that one complex multiplication involves 4 multiplies, 1 add, and 1 subtract,[4] while a complex add/sub involves 2 adds, one for the real portion and one for the imaginary portion.[5] Consequently, the *basic computation* is comprised of a total of 10 arithmetic operations that are necessary to compute each *butterfly*, or new pair of points, which corresponds to 5 arithmetic operations per point.

4 Naive Vector Algorithm

Figure 1 illustrates the data flow pattern for the Cooley-Tukey FFT algorithm. In this algorithm, which we will call the *naive algorithm*, there are $\log_2$ N stages for an N-point FFT. The example in Figure 1 is for a 16-point FFT, so it shows all of the butterflies for each of the $\log_2 16 = 4$ stages. The points are labeled using their binary representation, and the *butterfly groups* are indicated using vr1 and vr2.

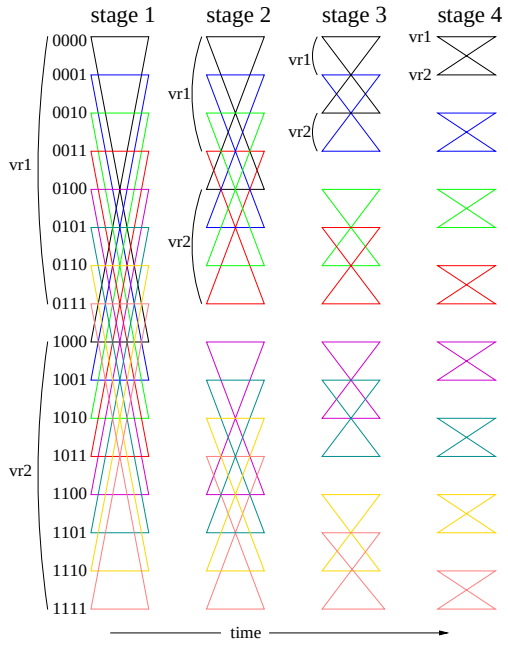

Figure 1: **Data dependencies in the Cooley-Tukey FFT algorithm. There are $\log_2$ N stages for an N-point FFT. In this figure N = 16 so it has $log_2 16$ = 4 stages. Note: For clarity, the figure only shows vector register 1 (vr1) and vector register 2 (vr2) which hold the real parts of the complex points. The imaginary parts of the complex points are assumed to be in vr3 and vr4.**

A natural vectorization of this naive FFT algorithm performs the basic computation on a set of butterflies as one vector operation. In Figure 1, for example, the first stage can be performed by loading the real and imaginary parts of elements 0-7 (0000-0111) into one pair of vector registers, vr1 for the reals and vr3 for the imaginaries, and elements 8-15 (1000-1111) into a second pair of vector registers, vr2 for the reals and vr4 for the imaginaries. Then the basic computation, *i.e.*, the 10 arithmetic operations, is performed using these 4 registers. In this first stage of this example, there are 8 elements in each of the vector registers, so the *vector length* (VL) is 8 and one vector instruction will cause the same operation to be performed on each of the 8 elements. The impact of instruction issue and memory access overheads will be minimized when the vectors are closer to the maximum vector length (MVL), which on VIRAM is 64 single precision elements. In the 16-point FFT depicted in Figure 1, notice that since the first stage has a vector length of 8, which is exactly N/2, there is only one *butterfly group* and therefore there is only one vectorized basic computation to perform. In each successive stage, the

[4] $(\omega_{\text{real}} + i \cdot \omega_{\text{imag}}) \cdot (x_{\text{real}} + i \cdot x_{\text{imag}}) = (\omega_{\text{real}} x_{\text{real}} - \omega_{\text{imag}} x_{\text{imag}})_{\text{real}} + (\omega_{\text{real}} x_{\text{imag}} - \omega_{\text{imag}} x_{\text{real}})_{\text{imag}} i$

[5] $(y_{\text{real}} + y_{\text{imag}} i) + (x_{\text{real}} + x_{\text{imag}} i) = (y_{\text{real}} + x_{\text{real}})_{\text{real}} + (y_{\text{imag}} + x_{\text{imag}})_{\text{imag}} i$

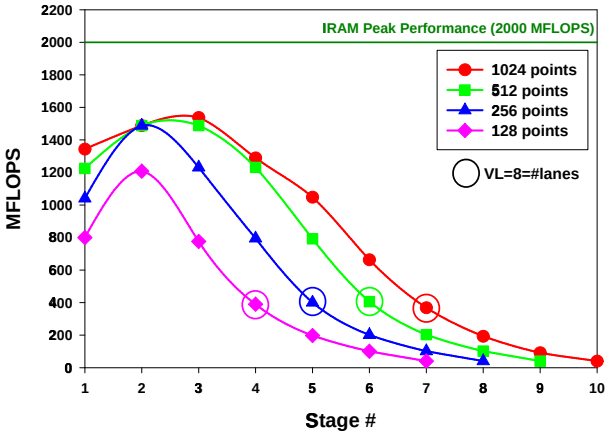

Figure 2: **Performance of each stage of an N-point FFT using the naive FFT algorithm on VIRAM for N = 128, 256, 512, and 1024. The circled points indicate the stage at which the VL = 8**

Number of FFT points	Percent of Total Time	Percent of Total Work
1024	94%	60%
512	95%	67%
256	96%	75%
128	97%	86%
64	100%	100%
32	100%	100%

Figure 3: **The percentages of total time and total work that the naive FFT algorithm spends in stages whose VL is less than MVL. Note: For all FFT sizes less than 128 both percentages are 100% since the VL starts out less than MVL.**

vector length drops by a factor of 2 and the number of butterfly groups doubles.[6]

After all the basic computations have been completed for the last stage, the order of the elements is *bit reversed*. This means, for instance that element 1 (0001) must be swapped with element 8 (1000). Similarly element 2 (0010) must be swapped with element 4 (0100). So the final step in this naive algorithm is to do the bit reversed swapping of all the elements so that the final array of elements in memory will be in the correct order.

Figure 2 shows the performance of this naive algorithm in MFLOP/s for each stage of FFTs of various sizes. The performance for a given stage depends primarily on the length of its vectors, so as VL halves from one stage to the next, the MFLOP rate dramatically decreases as well. The 2 GFLOP/s line on the plot shows the maximum performance for the radix-2 FFT computation that might be ideally attainable, taking into account only the arithmetic operations. As explained above, the VIRAM hardware peak of 3.2 GFLOP/s for single precision can only be obtained when multiply-add instructions are used; most other single floating point instructions have a hardware limit of 1.6 GFLOP/s.[7] Of the 10 arithmetic operations within a basic computation, 2 multiplies and 2 adds can be combined into 2 multiply-add instructions. Thus, the basic operation becomes 2 multiply-adds, 2 multiplies, and 4 adds/subs, or a total of 8 arithmetic operations, which results in the 2 GFLOP/s maximum for this mix of instructions.[8]

The overall performance of this algorithm is a disappointing 206 MFLOP/s for a 1024 point FFT. Looking at the performance for each stage in Figure 2, the reason becomes clear: the time is dominated by the later stages of the FFT, which have short vector lengths. The first stage is somewhat slower than the second because the program start-up overhead is included with the first stage only. The earlier stages (after the first) achieve a respectable rate of 1400-1800 MFLOP/s, but the rates for the later stages are much lower. The performance degradation is especially severe after the vector lengths fall below 8, because not all of the 8 single precision virtual lanes, and therefore not all of the 8 floating point functional units, are being fully utilized. Figure 3 gives the percentage of total time that the naive algorithm spends computing in all the stages with a VL less than MVL and the percentage of total work that the work in these stages represents. In particular, in a 1024 point FFT, 94% of the algorithm's total time is spent computing in the last 6 of the 10 stages, although the work in these stages constitutes only 60% of the total work.[9]

[6] In each stage, each butterfly group requires using a different root of unity for its basic computation.

[7] 1.6 GFLOP/s = 8 virtual lanes * 1 floating point functional unit/virtual lane * 1 FP operation/cycle * 200 Mcycles/second

[8] 2 GFLOP/s = 2 multiply-adds(MAs)/8 total * 3.2 GFLOP/s + 6 non-MAs/8 total * 1.6 GFLOP/s

[9] We assume that most applications will perform a series of FFTs, all of the same size, and we therefore precompute the roots of unity and some other values that are determined by the problem size.

Number of FFT points	No BRR No AI	AI	AI & BRR
1024	202	225	206
512	186	223	196
256	166	200	175
128	146	175	154
64	123	145	129
32	100	118	104
16	78	90	79
8	56	62	56
4	35	37	35

Figure 4: **The MFLOP rates for FFTs ranging in size fron to 1024 points are presented in this table. Each column cc tains results generated using different versions of the naive gorithm: the original one that did not do the bit reversal re rangement (BRR) of the final points, the same algorithm w.... auto-increment (AI) added, and the AI version with the (BRR) added.**

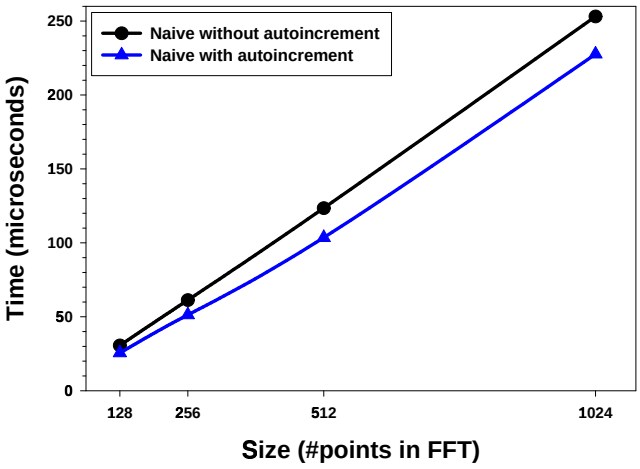

Figure 5: **VIRAM performance, measured in microseconds, of two versions of the naive FFT algorithm one utilizing auto-increment, the other not utilizing it.**

5 Optimizing the FFT

5.1 Auto-increment addressing

In order to optimize the naive FFT algorithm for VIRAM, the performance degradation observed in Figure 2 for the stages whose vector lengths were shorter than MVL had to be reversed. With this as the focus, our first optimization utilized an *auto-increment* feature for memory operations that automatically adds an increment value to the current address in order to obtain the next address. The auto-increment feature is useful, for example, when processing a sub-image of a larger image in order to jump to the appropriate pixel in the next row. In the FFT it can be used to jump between butterfly groups. Without auto-increment, scalar code is needed to calculate the next address to be accessed. The overhead cycles for this scalar address manipulation can be hidden only if the vector functional units are kept busy for an equal or greater number of cycles. Since the vector unit can complete 8 single precision operations per cycle, and since the scalar unit can complete only 2 per cycle, there must be 4 vector operations for every one scalar operation for the scalar operation to be hidden. Vector computations with short vector lengths contain fewer vector operations, and thus can hide fewer scalar operations. Thus by reducing the scalar code overhead for the stages whose vector lengths are short, the auto-increment feature helps to improve the performance of the naive algorithm because there is less scalar code to hide.

Unfortunately, the performance improvements realized by utilizing the auto-increment feature are not significant. For example, for a 1024 point FFT, the overall performance using the naive algorithm without the auto-increment feature was 202 MFLOP/s, while with auto-increment it was 225 MFLOP/s, a gain of only 23 MFLOP/s. As can be seen in Figure 4 and in Figure 5, the improvements in performance are even lower for the smaller size FFTs since a larger percent of their total work is being done in stages whose vector lengths are short.

As indicated above, the MFLOP rates graphed in Figure 2 and listed in the two left columns of Figure 4 are for the naive algorithm implemented with and without utilizing the auto-increment feature. However, neither of these versions of the algorithm do the necessary bit reversal rearrangement of the final data points. Bit reversal rearrangement of data can be quite expensive on some machines because, like the FFT itself, there is little data locality. Furthermore, there is a significant amount of address (pointer) arithmetic that is computed using scalar code. Adding more scalar code to the original naive algorithm would only exacerbate the problem that we were trying to ameliorate by utilizing the auto-increment feature, *i.e.*, to reduce the scalar code density since it cannot by hidden behind the vector computations when the vector lengths are short. The numbers in the right column of Figure 4 corroborate this claim; adding the bit rever-

sal rearrangement to the auto-increment version of the naive algorithm adds an average of approximately 12% overhead to the MFLOP rate of this version. Because the original naive algorithm is already very slow, and because adding the bit reversal rearrangement would only make it slower, we did not implement a version of the original naive algorithm with the bit reversal rearrangement.

5.2 Transpose-based algorithm

Due to the negligible performance gains from the use of the auto-increment feature in the naive algorithm, alternative approaches that might yield a more significant gain need to be considered if the naive algorithm is to be optimized. One such approach is to reorganize the data layout to maximize vector lengths in the later stages of an FFT. By viewing the 1D vector as a 2D matrix and performing a reorganization equivalent to a matrix transpose operation, one can increase the vector length used for the later stages in the naive algorithm. However, to keep full vector lengths, one may have to transpose several times, *e.g.*, 5 times in a 128 point FFT, which would clearly pose a performance problem, and would be even worse for vectors smaller than 128 elements.

In-register transpose

An alternative to the in-memory transpose is to transpose the elements within the vector registers themselves. This approach eliminates the need for intermediate memory accesses (which clearly is an optimization) and it keeps the vector lengths equal to MVL throughout the later stages of the naive algorithm (which eliminates our short vector length problem!). The naive algorithm could be used to vectorize all stages whose vector length is equal to or bigger than MVL.

The stage whose vector length equals MVL/2 will be the first stage for which the in-vector register transpose is utilized. For single precision data MVL $= 64$, so the in-vector register transposing would be performed for the last 6 stages where VL starts at 32, and is repeatedly halved for each successive stage, until the last stage, when it is equal to 1. Recall that the vector length determines how many elements there are in one butterfly group. When VL is 32 and MVL is 64, for instance, there are 32 elements in one butterfly group; so one vector register can hold all the elements for 2 butterfly groups. In this case, when the basic computation is performed on the elements in this register, two butterfly groups get computed with one vectorized basic computation. Similarly, for the next stage whose VL would be 16, one vector register could hold 4 butterfly groups, each having 16 elements, and the vectorized basic computation can be performed on all 4 butterfly groups. In this manner, each basic computation is performed on vector registers with VL = MVL, so the algorithm has been optimized for all of the short vector length stages.

Transpose example

For illustration purposes, assume MVL is 8 and $N = 16$. The first stage (and any previous stage in a larger FFT) would be performed by vectorizing across the butterflies as in the naive algorithm using a maximum vector length of 8 as pictured in Figure 1. Since stage 1 in this example has one butterfly group with a VL = MVL, one vectorized basic computation would be performed on all corresponding 8 element pairs, *i.e.*, elements 0-7 with elements 8-15. Therefore, at the beginning of stage 2, the two pairs of registers (*i.e.*, the real pair, vr1 and vr2, and the imaginary pair, vr3 and vr4) hold intermediate values for elements 0-7 and 8-15, respectively. As explained earlier, the new VL for stage 2 will be 4, which is half the VL of stage 1, and the new number of butterfly groups for stage 2 will be 2, which is twice the number in stage 1. Since the stage 2 VL, which is 4, is MVL/2, stage 2 is the stage in which the in-vector register transposes begin. The first stage 2 butterfly group will pair elements 0-3 with their corresponding elements 4-7, while its second butterfly group will pair elements 8-12 with their corresponding elements 13-15.

The first optimization is to rearrange the elements in the vector registers in order to eliminate the need to do the swap via memory accesses between each stage. The second optimization enables both butterfly groups to be done together using one vectorized basic computation with a VL of 8, no matter which stage is being computed. Consequently, after the rearrangement, the first set of vector registers (vr1 and vr3), which initially held the real and imaginary parts for elements 0-7, would end up holding the real and imaginary parts for elements 0-3 followed by elements 8-11. Likewise, after the rearrangement, the second set of real and imaginary vector registers (vr2 and vr4), which initially held elements 8-15 would end up holding elements 4-7 followed by elements 12-15. Notice that for the real values, the rearrangement essentially swaps elements 4-7 in vr1 with elements 8-12 in vr2. This element swapping within the vector registers from stage 1 to stage 2 is illustrated in Figure 6. Notice that with the elements swapped, the vectorized basic computation is being done once for two shorter butterfly groups instead of twice.

After the stage 2 computations have thus been per-

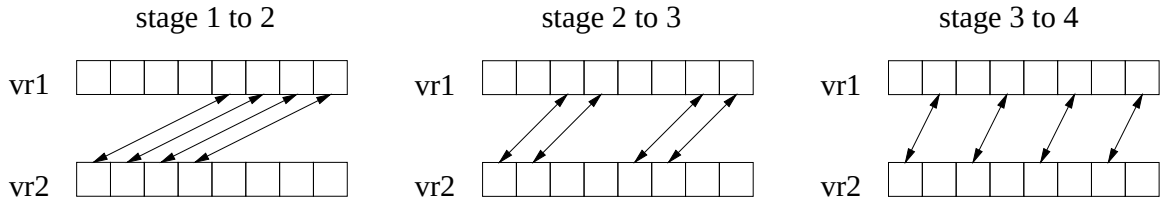

Figure 6: **In-register movements for the final 3 stages of a 16-point FFT, illustrated with 8 elements per register. Note that just the vector registers which hold the real portions of each element, *i.e.*, vr1 and vr2, are used in this figure. An identical pattern of swapping would be done between the vector registers which hold the imaginary portions of each element, *i.e.*, vr3 and vr4**

formed for these rearranged elements, the elements are once again rearranged in the vector registers in a similar fashion to allow for stages 3 and 4 to be done on those same set of 16 elements. These rearrangements are also illustrated in Figure 6. Notice that with each successive stage, the number of butterfly groups is still doubled and the number of elements in each butterfly group is still halved, but the vector lengths for the two registers are still equal to 8. If our example was a 32 or 64 point FFT instead of a 16 point FFT, but the MVL remained 8, a whole new set of 16 elements, *i.e.*, the next *outer iteration* butterfly group, would be loaded into the two registers and stages 2 to 4 would be performed for them in a similar fashion. This procedure would iterate until all the computations had been completed for all points in the FFT. It is important to notice that the number of elements in this outer iteration butterfly group is always equal to MVL * 2 since two vector registers must be filled with MVL elements (unless the number of points in the FFT is less than MVL * 2).

Implementation of in-register transpose

As illustrated above, the in-register FFTs require something akin to memory transpose operations but the data stays within the vector register file; the in-register data movement is much less expensive than doing memory accesses between each stage. Figure 6 shows the desired pattern of data movement between register pairs in the final stages of the FFT, with the arrows indicating elements that should be swapped. To help provide this functionality in the VIRAM ISA, two new instructions were added, `vhalfup` and `vhalfdn`. These instructions perform one-way moves that shift elements either up (`vhalfup`) or down (`vhalfdn`) between registers. A sequence consisting of one register-to-register copy followed by one `vhalfup` and one `vhalfdn` accomplishes the pattern of data movement required for the FFT.

An argument in a control register indicates the number of contiguous elements to be moved. With these new instructions, any FFT of size 128 or larger (for 32-bit values) can be performed with the maximum vector length throughout all stages of the computation. These instructions could also be used to *stack* a small number of shorter FFTs, for example executing four 32-point FFTs in parallel.

The implementation of `vhalfup` and `vhalfdn` in VIRAM was simplified due to the fact that these two new instructions can be seen as extensions of existing VIRAM ISA support for fast in-register reductions, *e.g.*, computing the sum of all elements in one vector register. With reductions, one repeatedly moves the top half of the vector register to the bottom half of a second register, and performs a vector addition using half the vector length of the previous addition. This process is repeated until the vector length is 1. This pattern of movement is similar to that induced by `vhalfdn`, with the exception that `vhalfdn` adds the ability to move non-contiguous blocks of elements; vhalfup generalizes that pattern to work in the other direction as well.

Although these new instructions were added to the VIRAM ISA to support the FFTs and other small transposes, the additional hardware support to do so was minimal. Given the recognized need for fast reductions in a variety of applications, the VIRAM design had already incorporated the inter-lane communication hardware necessary to support doing these reductions. This same hardware with a few additional control lines was required to implement the vhalf instructions [Koz99], so there was practically zero tradeoff and much to be gained in implementing them.

6 Performance Results

Figure 7 shows the running times in microseconds for various size FFTs for two versions of the naive algorithm and one version of the optimized *vhalf* algorithm (labeled “Vhalfup/dn”). Both naive versions utilize the auto-increment feature; one includes the final bit reversal, while the other does not. The vhalf ver-

sion utilizes the *new* `vhalfup` and `vhalfdn` instructions, the auto-increment feature, software pipelining, and code scheduling; the final output points are bit reverse rearranged. These results are all for single precision, floatigng point (32-bit), complex, radix-2 FFTs. Also included in this figure are single data points representing the FFT running times for various competitive CPU/DSPs for a single FFT size (full data on the CPU/DSP results is presented in Figure 8). Because the DSP results were obtained from the DSP manufacturers and are intended to showcase the performance of the DSPs, we assume that they represent the performance on highly-tuned DSP-specific FFT algorithms.

The most important fact to draw from Figure 7 is that the vhalf algorithm on VIRAM outperforms all versions of the naive algorithm by an order of magnitude. This gap testifies to the effectiveness of the vhalf algorithm in fully utilizing VIRAM's vector unit, and reveals the importance of the new vhalfup/dn instructions, which enabled the implementation of the vhalf algorithm. Furthermore, notice that VIRAM is competitive with the high-end, specialized DSPs. For example, it outperforms the TigerSHARC by a factor of 1.11. VIRAM is also within a factor of two of the performance of several other high-end DSPs: the Pulsar runs at 1.33 times the performance of VIRAM, the Wildstar is 1.48 times faster, and the Pathfinder-1 is 1.66 times faster.

We believe that VIRAM's performance could match or exceed the performance of these DSPs if the VIRAM architecture were implemented commercially; the chip that we have simulated here is an academic proof of concept implementation, and as such does not demonstrate the full potential of the architecture.

Finally, notice that the addition of the bit reversal rearrangement slows the naive algorithm down by up to a factor of 1.09 (by exposing the scalar overhead in the bit reversal, as described earlier). In contrast, the vhalf algorithm does not suffer a performance degradation when the bit reversal is added: although not plotted in Figure 7, the performance of the vhalf algorithm without the bit reversal rearrangement is nearly indistinguishable from that of the version with the bit reversal. This is because the vhalf version without bit-reversal uses strided stores after computing the final stage, whereas the version that performs a bit-reversal uses indexed stores. In the absence of memory bank conflicts (which were not noticeable for the FFT sizes we examined), VIRAM executes strided and indexed memory operations at the same speed.

7 Conclusions/Future Work

In this paper we have shown that, despite being primarily designed for the broad consumer market of portable multimedia devices, the general-purpose Vector IRAM processor is capable of performing FFTs at performance levels comparable to or exceeding those of high-end floating-point DSPs. It achieves this performance through a combination of a highly-tuned algorithm designed specifically for the VIRAM's model of vector processing, a set of simple yet powerful ISA extensions that underly that algorithm, and the efficient parallelism of a vector processor embedded in a high-bandwidth on-chip DRAM memory.

Furthermore, we believe that the performance of the VIRAM architecture on the FFT has the potential to improve significantly over the results presented here. First, we have only examined floating-point FFTs to date; VIRAM has the potential to run 32-bit fixed-point FFTs at up to twice the performance of its floating-point versions, because there are twice as many integer as floating-point functional units in the implementation. We intend to implement and study fixed-point FFTs on IRAM as future work. Second, as mentioned earlier, our simulation results are based on the current proof-of-concept VIRAM implementation, which has made compromises that trade off potential performance for ease of implementation in an academic setting.

Finally, we believe that VIRAM occupies an interesting space in the emerging market of hybrid CPU/DSPs such as the Infineon TriCore, the Hitachi SuperH-DSP, the Motorola/Lucent StarCore, and the Motorola PowerPC G4 (7400). Like these other chips, VIRAM includes both general-purpose CPU capability as well as significant DSP muscle, as demonstrated by its high performance on the FFT. In addition, VIRAM's vector plus embedded-DRAM design may prove to have further advantages in power, area, and performance over these more traditional processor designs.

Acknowledgments

I would like to express my extreme gratitude to my friend and fellow graduate student Aaron Brown, without whose support and technical expertise this paper would never have become a reality.

I also wish to thank the IRAM group, and in particular Richard Fromm, Christoforos Kozyrakis, and David Martin for building the tools that made the simulation results presented in this paper possible, for answering all of

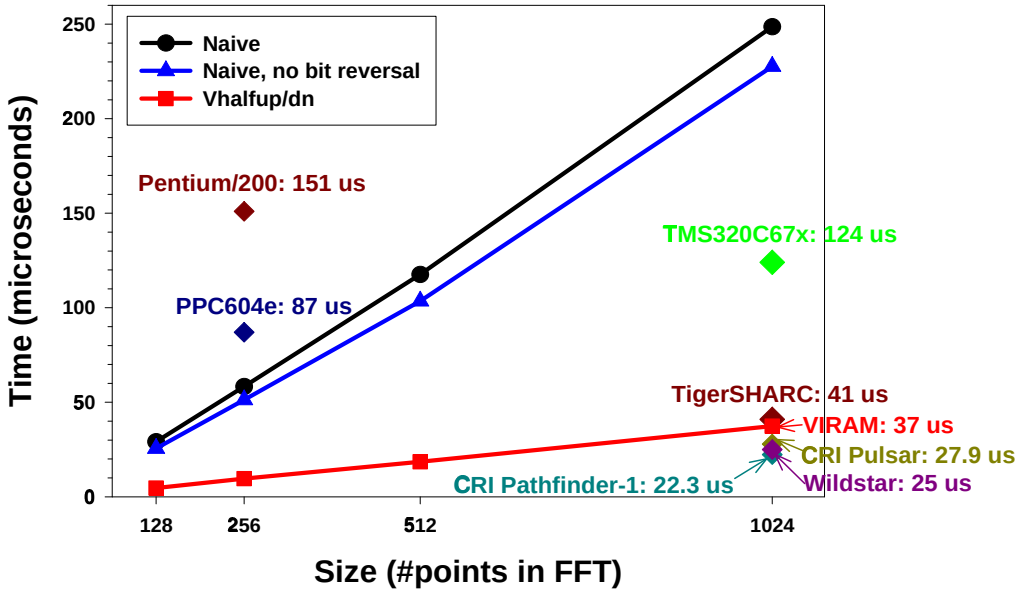

Figure 7: **Performance of three FFT algorithms on VIRAM: The naive algorithm with and without the final bit reversal, and the optimized algorithm that uses the Vhalfup/dn instructions.**

Processor	Reference	MHz	usec	Notes
1024 point FFT				
Pathfinder-1	[Inc99]	100	22.3	Block FP
Wildstar	[AMS]	n/a	25	FPGA
Pulsar	[Inc]	80	27.9	Block FP
VIRAM	this paper	200	37	vector
TigerSHARC	[Deva]	250	41	4-way VLIW
ADSP21160	[Devb]	100	92	Radix 4
TMS3206701	[Ins]	167	124.3	8-way VLIW
256 point FFT				
TigerSHARC	[Deva]	250	4.4	256 pt
VIRAM	this paper	200	9.525	vector
PowerPC604E	[Dub98]	200	87	256 pt
Pentium I	[Dub98]	200	151	256 pt

Figure 8: **Running times of a 1024 point and a 256 point floating point single precision complex FFT on various processors.**

my questions, and for agreeing with me that `vhalfup` and `vhalfdn` were an important addition to the VIRAM ISA.

Finally, I would like to thank Krste Asanovic and Cray Research, Inc. for introducing me to the world of vector computing.

References

[AMS] Inc. Annapolis Micro Systems. Annapolis micro systems, inc. homepage. `http://www.annapmicro.com/PR9126.html`.

[Asa98] Krste Asanovic. *Vector Microprocessors*. PhD thesis, University of California, Berkeley, 1998. UCB//CSD-98-1014.

[CT65] J.W. Cooley and J.W. Tukey. An algorithm for the machine computation of the complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[Deva] Analog Devices. Introducing tigersharc. `http://www.analog.com/techsupt/documents/tigersharc.html`.

[Devb] Analog Devices. pdf file:preliminary technical data report. `http://www.analog.com/pdf/ADSP_21160_p.pdf`.

[Dub98] Pradeep Dubey. Architectural and design implications of mediaprocessing, May 1998. `http://www.research.ibm.com/people/p/pradeep/media_tutorial/ppframe.htm%`.

[FJ98] M. Frigo and S.G. Johnson. FFTW: An adaptive software architecture for the FFT. In *ICASSP*, 1998.

[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.

[Inc] Catalina Research Inc. Cri web site: Fft tables. `http://www.cri-dsp.com/cri/newprod/fft.htm`.

[Inc99] Catalina Research Inc. Cri web site: Press releases, April 1999. `http://www.cri-dsp.com/cri/pressrls.htm#scorpio`.

[Ins] Texas Instrument. Tms320c6000 platform overview. `http://www.ti.com/sc/docs/products/dsp/c6000/67bench.htm`.

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