Efficient FFTs on IRAM

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

Abstract: Computing Fast Fourier Transforms (FFTs) are 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 doing FFTs, some of which are designed solely for the purpose of computing FFTs and related transforms. In this paper, we show that the general-purpose VIRAM processor's performance exceeds that of existing DSPs for computing floating point FFTs and is competitive with the performance of the specialized fixed-point FFT chips. VIRAM is a complete "system on a chip," and therefore has power, area, and cost advantages over multi-chip systems based on DSPs. The key to achieving these results are: an integrated processor-in-memory design for VIRAM that provides high memory bandwith; 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, e.g. hardware to improve the performance of bit-reversals or transpose operations. Of these, the ones with the best performance are those which 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 which 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 improve data re-use to some extent [FJ98], FFT performance on conventional microprocessors is typically limited by the poor memory bandwidth and high memory latency on these machines.

The IRAM project is exploring the use of embedded logic in DRAM ("Intelligent RAM") for building a single chip system designed for low power and high performance on multimedia applications. The Vector IRAM (VIRAM) system adds a vector processor to the embedded logic which produces a low energy, high level design suitable for the ever-growing market of portable and hand-held 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 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 rivals the best performance

*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.

of the special-purpose DSPs for computing FFTs.

Section 2 gives an overview of the VIRAM architecture and a discussion of the key design features that make it suitable for multimedia processing on small portable devices such as PDAs. Section 3 describes a standard FFT algorithm and section 4 shows a straightforward vectorization of that algorithm. Section 5 shows several optimizations. The performance, based on simulations, is reported in section 6; the performance is shown to be superior to all existing DSPs for single precision floating point and competitive with many fixed-point implementations. Section 7 draws some conclusions and gives an overview of our future plans for this research.

2 Overview of VIRAM

By placing a processor inside DRAM memory, one potentially exposes 100x more memory bandwidth than 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. The general-purpose vector processor provides high performance on computations with sufficient fine-grained data-parallelism. Since a delayed vector pipeline is utilized [Asa98, Koz99], no cache is necessary to hide memory latency, therefore the VIRAM architecture conserves area, while preserving the low-power benefits of a single chip 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, e.g., 64 elements (for single precision), VIRAM also has more compact instructions than the VLIW architectures currently being used in DSPs like TI's TMS320C6000, Motorola's StarCore 440, Siemens (Infineon) Carmel, and Analog Device's TigerSHARC, which translates to power and performance advantages.

VIRAM is a complete "system on a chip," and therefore enjoys power, cost and area advantages over multichip systems [PAC+97]. In addition to the vector processor and DRAM memory, VIRAM has a simple MIPS core, a memory crossbar, and an I/O interface for off-chip communication. The implementation of VIRAM, which is currently under development, is designed to run both the vector and scalar processors at 200 MHz, with four 64-bit vector pipes, 16 MB of DRAM organized into 8 banks, four 100 MB/s parallel I/O lines, a 1.2V power supply, and a power target of 2 Watts [Koz99]. Each of the four 64-bit vector pipelines, called *lanes*, may also be used as two 32-bit lanes (which makes a total of 8 lanes) or 4 16-bit lanes (which makes a total of 16 lanes;) the narrower data widths are particularly useful for some DSP and multimedia computations. There are two integer units and a single floating point unit per lane; all of these support a multiply-add instruction. Using multiply-adds, the peak performance is 3.2 GFlops for single precision floating point and 6.4 GOPS for 32-bit integer operations. Since the VIRAM chip is not yet available, 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 the speed of certain memory access patterns. 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. IRAM 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 details of the VIRAM design as they become relevant to the problem of developing a high performance FFT algorithm.

3 Vector Algorithms for computing FFTs

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:

$$\forall j \in [0, N-1] \quad y_j = \sum_{k=0}^{N-1} \omega_N^{jk} x_k, \quad \text{where} \quad \omega_N^{jk} = e^{\frac{-2\pi i jk}{N}}$$

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

The Fast Fourier Transform [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) and in every stage each point is paired with another, the same basic computation is performed between the two, and the values overwritten in the input vector. For example, in the first stage, x_0 and $x_{N/2}$ are paired and the basic computation is as follows:

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

where ω is one of the roots of unity. In a complex FFT, both the x_i's and the roots of unity are complex numbers, therefore the complex multiplies actually involve 4 multiplies, 1 add, and 1 subtract, while a complex add involves 2 adds. Consequently, the basic computation has a total of 10 arithmetic operations to compute each new pair, which is called a "butterfly."

4 Naive Vector Algorithm

Figure 1 shows the data flow pattern for the 4 stages in a 16-point FFT.

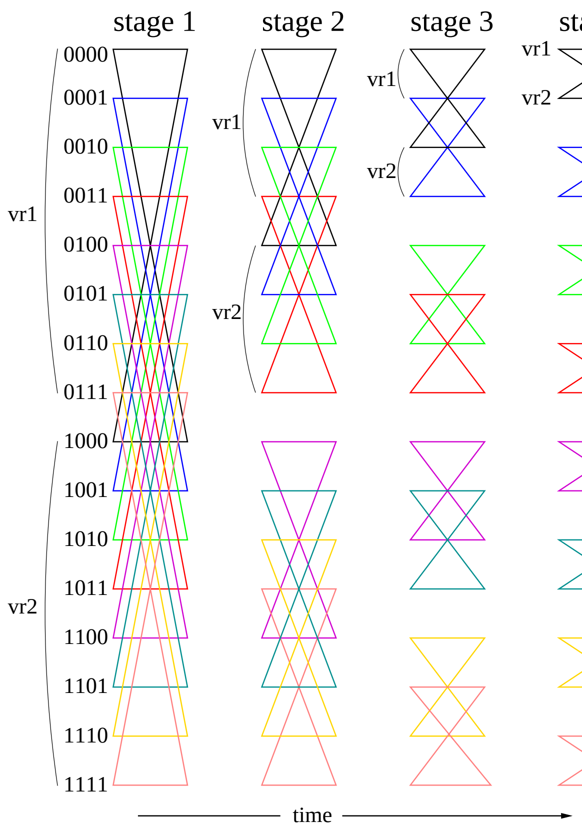

Figure 1: Data dependencies in the Cooley-Tukey FFT algorithm.

A natural vectorization of the 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 into one pair of vector registers, vr1 and vr3, and elements 8-15 into a second pair of vector registers, vr2 and vr4. Then the 10 arithmetic operations are performed using these 4 registers. In this case there are 8 elements in each of the vector registers, so the *vector length* is 8. The impact of instruction issue and memory access overheads will be minimized when the vectors are closer to the maximum, which on VIRAM is 64 single precision elements. In Figure 1, the first stage has a vector length of 8 and there is one vectorized basic computation to perform (i.e. one group). In each successive stage, the vector length drops by a factor of 2 and the number of vectorized basic computations (i.e. the number of groups) doubles.

Figure 2 shows the performance in MFlops of each stage in this algorithm. As can be seen, the performance for a given stage is dependent primarily on the length of its vectors. For each line in figure 2 which doubles the number of points of the line appearing below it, the performance curve is shifted to the right by 1 stage, since the added stage is one with longer vectors. In other words, if all of the lines began in canon and ended at stage 10 instead of all beginning at stage 1 and ending in canon, then all the curves would be on top of one another.[1] (For

[1]The naive algorithm used for this figure does not perform bit-reversal that is necessary in most FFT applications, although our optimized algorithm in a later figure does. For the final version of the paper we will add bit-reversals to the naive algorithm and report

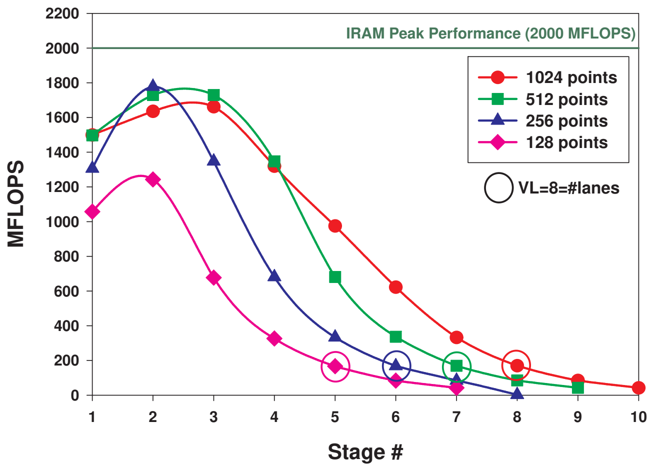

Figure 2: Performance of each stage of a naive FFT algorithm on IRAM.

each line in figure 2, the first stage is somewhat slower than the second because the program start-up overhead is included with the first stage only.) The 2 GFlops line on the plot shows the maximum performance for the radix-2 FFT computation that one might attain, taking into account only the arithmetic operations. The VIRAM hardware peak of 3.2 GFlops 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 GFlops. Of the 10 arithmetic operations within a basic computation, 2 multiplies and 2 adds can be combined into 2 multiply-add instructions, which results in the 2 Gflop maximum for this mix of instructions.

The performance of this algorithm is a disappointing 202 MFlops for a 1024 point FFT. The reason is clear from looking at the performance in each stage in Figure 2: the time is dominated by the latter stages of the FFT which have short vector lengths. The earlier stages (after the first) achieve a respectable rate of 1400-1800 MFlops, but the latter stages are much lower. The performance especially degrades after the length falls below 8, because not all of the 8 single-precision lanes are being fully utilized. In particular, in a 1024 point FFT, 94% of the total time is spent in the last 6 of the 10 stages, although these stages constitute only 60% of the total work. [2]

that time as well.

[2]We assume that most applications will perform a series of FFTs, all with the same size, and we therefore precompute the roots of unity and some other values that are determined by the problem size. Some things that could be precomputed are not, but will be in the final paper, possibly resulting in better performance on all of the algorithms in the paper.

5 Optimizing the VIRAM FFT

The first optimization we implemented to improve performance on short vectors is the use of an *auto-increment* feature. The auto-increment mode is used on memory operations to automatically add an increment value to the previous address to obtain the next one. The auto-increment feature is useful, for example, when processing a sub-image of a larger image in order to jump to the next row. In the FFT it can be used to jump between groups of butterflies. Since the scalar address manipulation overhead cannot be as easily hidden by the computations when short vectors as opposed to long vectors are used, the auto-increment feature helps performance by reducing this overhead. Although auto-increment improves the performance, the overall result is only 225 MFlops for a 1024 point FFT and even lower performance for the smaller sizes.

In order to attain close to peak performance, one approach is to reorganize the data layout to maximize vector lengths in the latter stages of an FFT. By viewing the 1D vector as a 2D array and performing a reorganization equivalent to a transpose operation, one can increase the vector length. 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.

An alternative to the in-memory transpose is to complete all of the last 6 stages whose vector lengths are less than 64 elements for one group of elements before moving on to the next group. This is equivalent to performing an in-register FFT. For illustration purposes, assume the maximum vector length is 8 elements per vector register and that $N = 16$ as in Figure 1. The first stage (and any previous stage in a larger FFT) would be performed by vectorizing across the butterflies as in the naive algorithm and would use the maximum vector length. At the beginning of the second stage whose vector length is 4 elements, the two pairs of registers hold elements 0-7 and 8-15, respectively. In order to do the desired butterflies on elements 0-3 and 4-7 and on elements 8-11 and 12-15, the elements must be rearranged in the vector register. The rearranged order is as follows: elements 0-3 and 8-11 are in one register and elements 4-7 and 12-15 are in the other register. Notice that now the vectorized basic operation is being done once for two shorter groups. After the stage 2 computations have thus been performed for these elements, the elements are once again rearranged in the vector register in a similar fashion to allow for stages 3 and 4 to be done on those same set of 16 elements. Notice that with each successive stage, the number of groups is doubled and the number of elements in each group is halved, but the vector lengths for the two registers is still equal to 8. If there were more points left, a whole new set of 16 elements would be loaded into the two registers and stages 2-4 would be performed for them in a similar fashion.

The in-register FFTs require something akin to transpose operations within the vector register file, but this is much less expensive than going out to memory between each stage. To provide the necessary functionality in the

VIRAM ISA, extensions were made to operations already in the ISA for performing 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. Figure 3 shows the desired movements between register pairs in the final stages of the FFT, with the lines indicating elements that should be swapped. In the VIRAM ISA, these data movements are performed with a series of register-to-register copies and one-way moves that shift elements either up (`vhalfup`) or down (`vhalfdn`) between registers. 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 at full vector length throughout the computation. These instructions could also be used to “stack” a small number of shorter FFTs such as four 32-point FFTs that execute in parallel.

Although some new instructions were added to the VIRAM ISA to support the FFTs and other small transposes, the additional hardware support was minimal. Given the recognized need for fast reductions in a variety of applications, the VIRAM design already needed to support inter-lane communication, which is the main feature required to implement the vhalf instructions [Koz99].

6 Performance Results

Figure 4 shows the running time for various size FFTs using the naive algorithm, the same algorithm optimized with auto-increment, and the optimized algorithm that uses the vhalf instructions. These results are all for single precision floating point (32-bit), complex, radix-2 FFTs. The numbers for the two naive algorithms do not include bit-reversal time, since it had not been implemented when those simulations were run; for the final version of the paper we will include the bit-reversal time. The version optimized with vhalf instructions *does* include bit-reversal; an interesting note is that the addition of bit-reversal code had negligible impact on the running time, because the vhalf version without bit-reversal uses strided stores in the final stage, whereas the version that performs a bit-reversal uses indexed stores. VIRAM executes strided and indexed memory operations at the same speed, except when there are memory bank conflicts, and our experience indicates that bank conflicts were not noticeable for the given FFT

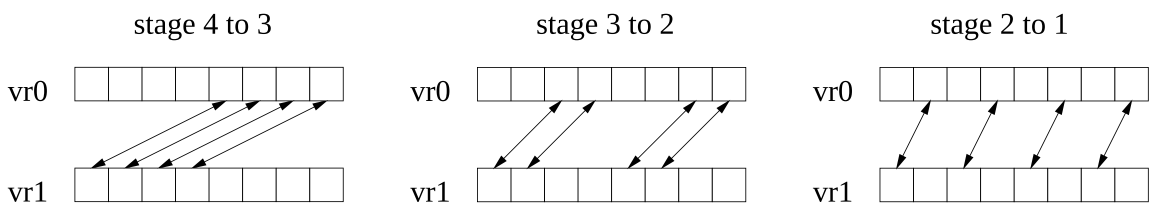

Figure 3: Register movements for the final 3 stages of an in-register FFT, illustrated with 8 elements per register.

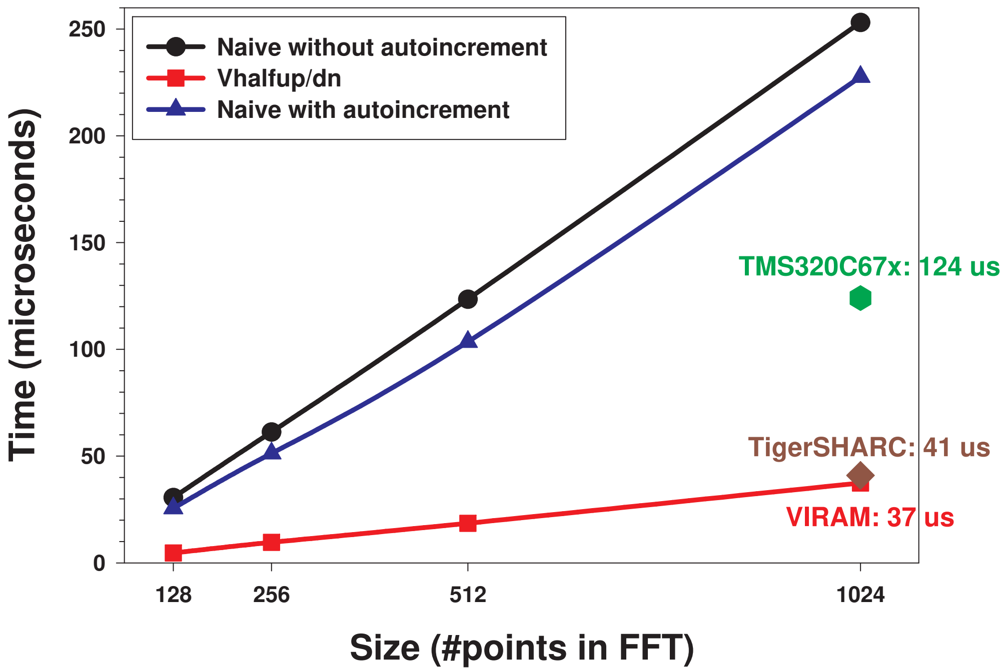

Figure 4: Performance of 3 FFT algorithms on VIRAM.

sizes.

7 Conclusions and Future Work

The data in Figure 4 shows that VIRAM is faster than the best floating-point DSPs. A 1024 point FFT executes in 37 usec on VIRAM, compared to 41 usec on the TigerSharc design and 128 usec on TI's TMS320C6000. Although the comparison is somewhat unfair to VIRAM, we can also compare its single precision performance to 24-bit fixed point computations. We believe the fastest of these is CRI's Pathfinder-1 vector processor, which performs a 1024 point complex FFT in 28 usec. Motorola's DSP56002 (an older design) performs the same operation in 908 usec. Although beyond the scope of this paper, VIRAM has the potential to run 32-bit fixed point FFTs up to 2x its floating point versions, because there are twice as many integer as floating point units in the implementation.

Another interesting comparison is the power consumption of VIRAM verses the DSPs. Precise comparisons are difficult, because the DSPs often require multiple chips to perform a comparable task. The Pathfinder-1 chip, for example, consumes 2 Watts, which is equal to the power budget for VIRAM; however, Pathfinder-1 needs off-chip

SRAM to execute the FFTs and more power will be consumed for its external memory and memory bus.

In VIRAM, the combination of high memory bandwidth from embedded DRAM, which is much lower in cost, area, and power than off-chip SRAM, and vector processing is well-suited to the memory access patterns that arise in FFTs. Our results show that although VIRAM is a processor designed for a broad consumer market of hand-held and portable devices that can efficiently run multimedia applications, when VIRAM is running a well-chosen algorithm it is faster than the fastest DSPs for computing floating point FFTs and competitive in power and performance when compared to the fastest fixed-point DSPs.

References

[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.

[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.

[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.