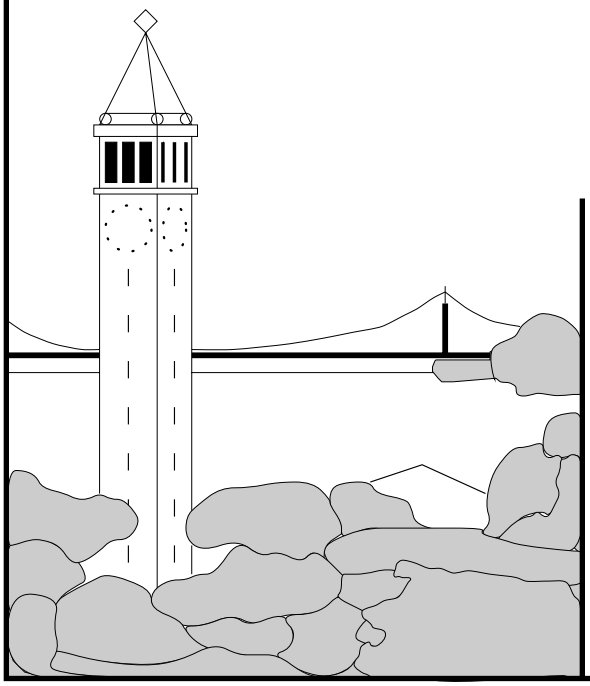
An Architectural Performance Study of the Fast Fourier Transform on Vector IRAM

Randi Thomas
Computer Science Division
University of California, Berkeley



Report No. UCB/CSD-00-1106

June 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

An Architectural Performance Study of the Fast Fourier Transform on Vector IRAM

by Randi Thomas

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

Committee:

Professor Katherine Yelick
Research Advisor

(Date)

* * * * * *

Professor David A. Patterson
Second Reader

(Date)

This thesis is dedicated

To the memory of my parents,
Eleanor and James Haritos,

and

To my wonderful and supportive son,
Tyson James Thomas

An Architectural Performance Study of the Fast Fourier Transform on Vector IRAM

Randi Thomas *

Master of Science Report

Abstract

In this paper we develop an optimized algorithm for performing the Fast Fourier Transform (FFT) on the Vector IRAM (VIRAM) architecture in both the fixed- and floating-point domains. We discuss the impact of various optimizations on the performance of the FFT algorithm on VIRAM, including both an analysis of the usefulness of various VIRAM ISA features as well as a consideration of the performance and accuracy consequences of performing the FFT computations in the fixed-point domain rather than the traditional floating-point domain.

We compare the performance of our most-optimized FFT algorithm on a simulated version of VIRAM to that of eleven high-end fixed- and floating-point Digital Signal Processors (DSPs) and DSP-like architectures, and find that VIRAM outperforms all of the fixed-point DSPs and all but two of the special-purpose floating-point FFT DSPs. On 1024-point FFTs, VIRAM achieves 1.3 GFLOP/s in floating-point mode, and 1.9 GOP/s in fixed-point mode.

Despite its high performance relative to the DSPs, however, we find that the VIRAM architecture is being underutilized by as much as two thirds while running the FFT algorithm. We thus embark on an architectural analysis to determine the underlying cause of this underutilization, and discover that it results from bottlenecks in VIRAM's memory functional units and memory access conflicts in VIRAM's memory system. For larger FFTs, the memory system impact becomes more severe, and we find that the number of memory banks and subbanks plays a crucial role in the scalability of our algorithm's performance to large FFT sizes.

*This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract DABT63-96-C-0056, the National Science Foundation Infrastructure under grant no. CDA-9401156, the California State MICRO Program, and by a grant from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. The author was supported in part by a National Science Foundation fellowship.

Acknowledgments

I would like to express extreme gratitude to my friend and fellow graduate student Aaron Brown, without whose support and technical expertise this report 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 my questions, and for agreeing with me that **`vhalfup`** and **`vhalfdn`** were an important addition to the VIRAM ISA.

In addition, I would like to thank Krste Asanovic, Jim Kohn, and Cray Research, Inc. for introducing me to the world of vector computing, my advisor Professor Kathy Yelick for insightful comments, technical advice, and encouragement to study the FFTs on VIRAM, and Professor David A. Patterson for his enthusiastic interest, encouragement, and support and for willingly taking on the task of reading this report and giving me his comments.

I wish to acknowledge my very dear friend, Gail Cotton High, for all her love, support, encouragement, meals, time, friendship, and for always being there for me when I need her. In addition I thank my **AOPi** sister and friend, Ayris Hatton for cheering me on and keeping my perspective clear. Finally I would like to thank my family, the Van Stolks, the Striebels, the Arenas, the Puzzos, as well as my son, Tyson. I truly appreciate the love and support they have all given to me, especially during my graduate study years. I dedicate this thesis to the memory of my parents, Eleanor and James Haritos, because both of them, but especially my father who barely finished eighth grade, would have been extremely proud to have a daughter with a Masters of Science degree.

Contents

List of Figures

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). Although the algorithm chosen to compute the FFT may be reorganized to improve data re-use [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 paying careful attention to those architectural details that impact the performance of the FFT. It also discusses the key design features that make VIRAM suitable for multimedia processing on small portable devices. Section 3 describes a standard FFT algorithm. Section 4 first discusses a straightforward vectorization of that standard algorithm; it then compares several optimizations to this algorithm and finishes with an analysis of the performance of the optimized algorithm. Section 5 discusses the adaptations made to the optimized floating-point version to create an optimized fixed-point version; it then describes the performance of both the fixed-point and floating-point versions, based on simulations, and finishes with an error analysis of the fixed-point results. A comprehensive architectural analysis of the performance results is given in Section 6. VIRAM's performance results are then compared to both fixed-point and floating-point DSP performance in Section 7; the performance is shown to be comparable to existing DSPs for both floating-point-based and fixed-point-based algorithms. Section 8 draws some conclusions and then makes some suggestions for future implementations of the VIRAM architecture.

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

[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.

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 32-bit elements or 128 16-bit elements), VIRAM also has more compact instructions and greater code density than the VLIW architectures currently being used in DSPs. TI's TMS320C6201 and TMS320C6701, Motorola/Lucent's StarCore 440, Siemens (Infineon) Carmel, and Analog Device's TigerSHARC are examples of such VLIW DSPs. 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 multichip 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 is designed to run both the vector and scalar processors at 200 MHz. It has 16 MB of DRAM organized into 8 banks with no subbanks,[2] four 100 MB/s parallel I/O lines, a 1.2V power supply, and a power target of 2 watts [Koz99]. Since several of our experiments were performed before the final VIRAM design decisions had been finalized, these earlier experiments assumed a memory configuration of 32MB of DRAM with 16 banks and no subbanks.[3]

2.1 VIRAM Pipelines

VIRAM has four 64-bit pipelines, called *lanes*, each of which has two integer functional units; one of the integer functional units also serves as a floating-point functional unit. Each of these functional units supports a multiply-add instruction that can complete in one cycle. [4] 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).

To fully utilize all the available computational ability of VIRAM, a vector operand register must supply each virtual lane with one element per cycle. This subset of a vector register's elements is known as an *element group* and there are always eight *element groups* in a vector register that is filled to capacity. For example, for 32-bit data, the maximum number of elements one vector register can hold is 64. Since there are 8 virtual lanes, there are 8 elements in one *element group*, and eight *element groups* in one vector register.

2.2 VIRAM Memory Accesses

In the VIRAM architecture there are three different ways to access memory: *unit-stride* loads/stores, *strided* loads/stores, and *indexed* loads/stores. Specifically, *unit-stride* loads/stores access consecutive elements of memory, while *strided* loads/stores access memory using a constant jump between addresses such as every other element or every fourth element. In both the unit-stride and the strided loads/stores, memory is accessed in a uniform pattern. The *indexed* loads/stores, however, access memory non-uniformly by accessing arbitrary non-consecutive elements of memory. This non-consecutive memory access pattern is also called a *gather/scatter* and is accomplished in a vector architecture by using an array of indices that have been pre-loaded into a separate vector register to compute the memory addresses.

The VIRAM prototype implementation can generate four such memory addresses per cycle. For a unit-stride load or store, only one address need be generated per element group. Since there are eight element groups in a fully loaded vector register, all eight addresses can thus be generated in 2 cycles. So unit-stride loads and stores can execute with no pipeline stalls caused by the address generator.[5]

[2] The memory configuration has 2 semetrical wings, 4 banks/wing, 1 subbank/bank, 8192 rows/subbank, and 2048 bits/row.

[3] Each figure is notated with the memory configuration that was assumed for the experiment being illustrated.

[4] After the experiments in this paper were performed, it was decided that the floating-point functional units in the VIRAM prototype chip would not support the floating-point multiply-add instruction, although this instruction is still defined in the VIRAM ISA and we assume in our simulations that it is implemented. Note that the integer functional units in the VIRAM prototype chip do still support the multiply-add.

[5] If the element groups do not start on a correct memory

For indexed or strided memory accesses, however, an address for each element being loaded or stored must be generated.[6] This means that only 4 elements can be accessed per cycle. Recall that the number of virtual lanes determines the number of elements in an element group and that one element group can be computed upon in one cycle by VIRAM. If there are 4 elements in an element group, as is the case for 64-bit data, then unit-stride, strided, and indexed memory operations will all be able to access 4 elements per cycle so all types of memory accesses using 64-bit data will go at the same speed and utilize all the available computational ability of VIRAM.

However, when there are more than 4 elements in an element group, as is the case for 32-bit and 16-bit data, the unit-stride memory operations can still access an entire element group in one cycle, since only one address need be computed for the entire element group. However, the strided and indexed memory operations for these narrower data widths can only access 4 elements per cycle, since a separate address must be generated per element accessed and since the VIRAM hardware is limited by having only 4 address generators.

So generating the addresses for indexed and strided loads and stores for the 32-bit and 16-bit data widths stalls the pipeline. The narrower the data width, the more the situation is exacerbated. For example, for 32-bit data, loading an entire vector register of MVL$= 64$ elements using unit-stride would take 8 cycles, one cycle per element group.[7] Doing a strided or indexed load for the same vector register would take 16 cycles, one cycle per four elements. For 16-bit data a unit-stride load of an entire vector register containing MVL$= 128$ elements would still take 8 cycles, one cycle per element group.[8] However, a strided or indexed load for the same vector register would take 32 cycles, one cycle per four elements. Thus for 32-bit data the strided and indexed memory accesses take double the number of cycles of the unit-stride to do the access, but for 16-bit data, the strided and indexed memory accesses take quadruple the number of cycles of the unit-stride to do the access.

At the time the experiments in this paper were performed, the VIRAM prototype implementation and simulator did not decouple the memory functional unit from the arithmetic functional unit pipeline. Therefore any stalls in the memory functional unit, which are caused by the slower indexed or strided accesses or by memory bank conflicts, impact all the vector functional unit pipelines. Specifically, during an indexed or strided memory access, the arithmetic functional unit pipeline is operating at half its full capability for 32-bit data widths and at one fourth its full capability for 16-bit data widths. For 32-bit data, this means that one element group is processed by the arithmetic functional unit in two cycles instead of one, so for both of these cycles half of the available computational functional units are idle. In the case of 16-bit data the situation is exacerbated since one element group is processed by the arithmetic functional unit in four cycles instead of one, so for each of these four cycles three fourths of the available computational functional units are idle.

To address these deficiencies (which were revealed through our and other simulations), the final VIRAM design was altered to include several performance-enhancing improvements. One such improvement is the decoupling of the arithmetic functional unit pipeline from the memory functional unit pipeline. With this new feature incorporated into the chip, memory stalls for one single load or one single store will no longer stall the arithmetic functional unit pipeline. Although the VIRAM prototype will have a buffer to hold but one outstanding load or store, the buffer size could be increased in a future implementation.

The effect of this improvement on our FFT implementation is not reported in this paper since the VIRAM performance simulator has not yet incorporated these features, and consequently we cannot get such results at this time. However, notwithstanding this improvement, indexed and strided memory accesses should be used with care in the VIRAM ar-

boundary alignment, then it is possible that two addresses will have to be generated for one element group for a unit-stride access. In this case, the maximum number of addresses that would need to be generated to load or store a full vector register using a unit-stride would be exactly nine.

[6] The address for each element is computed by adding its corresponding index register element's value to a base address.

[7] This assumes all element groups start on the correct memory boundary alignment. If this is not the case, then the entire load would take 9 cycles instead of 8.

[8] Same assumption as above, *i.e.* correct memory boundary alignment.

chitecture.

In the VIRAM architecture, any instruction that operates on elements, such as the loads and stores just described, must know how many bytes there are per element. This is accomplished in the VIRAM ISA by setting a vector control register, the **vpw** ***control register***, to indicate the number of bytes per element, also known as the data width. Each instruction that operates on elements does so under the control of this **vpw** control register, and thus such instructions are able to access the correct number of bytes per element. Consequently changing the contents of the **vpw** register causes the hardware to easily switch from one data width to another.

2.3 VIRAM Peak Performance

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 floating-point 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[9] for single precision floating-point, 6.4 GOP/s[10] for 32-bit integer operations, and 12.8 GOP/s for 16-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 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.

[9] 3.2 GFLOP/s = 8 virtual lanes * 1 floating-point functional unit/virtual lane * 2 operations/cycle * 200 Mcycles/second

[10] 6.4 GOPS = 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:

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

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

N is referred to as the number of *points*, and ω_N^{jk} is the N-point jk^{th} *root of unity*. Thus the Fourier Transform takes O(N^2) steps to compute.

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.* Note that for complex data, the *basic computation* would involve doing 1 complex multiply, 1 complex add, and 1 complex subtract.

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,[11] while a complex add/sub involves 2 adds/subs, one for the real portion and one for the imaginary portion.[12]

Consequently, for a complex FFT, 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 Floating-Point FFT Vector Implementation

In this section, we describe a vector implementation of the FFT algorithm described in Section 3. We start with a straightforward, or "naïve", version, in section 4.1. We then continue in sections 4.2 and 4.3 by describing two optimizations to the *naïve* algorithm that allow it to take better advantage of the VIRAM architecture and thereby obtain higher performance. Throughout this section we develop the algorithm for a radix-2, single precision (32-bit), complex FFT.

4.1 Naïve Vector Algorithm

Figure 1 illustrates the data flow pattern for the Cooley-Tukey radix-2 FFT algorithm. In this algorithm, which we will call the *naïve* 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.

4.1.1 Naïve Vectorization

A natural vectorization of this *naïve* 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 — the 10 arithmetic operations — is performed using these 4 registers and the results in the four vector registers are then stored to memory. In the 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 VL is closer to the maximum vector length (MVL), which on VIRAM is 64 for 32-bit 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*

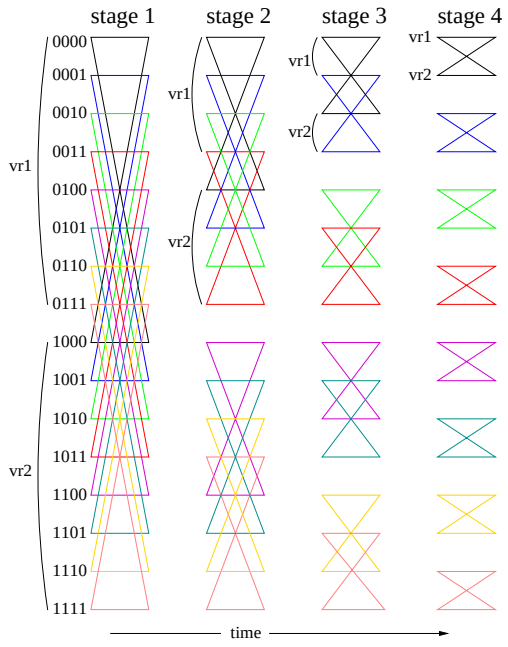


Figure 1: **Data dependencies in the Cooley-Tukey FFT algorithm. There are $\log_2$ N stages for an N-point FFT. In this figure N = 16 so it has $\log_2$16 = 4 stages. For clarity, the figure only shows vector register 1 (vr1) and vector register 2 (vr2) which hold the real parts of the complex points.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

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

[12] $(y_{\text{real}} + i \cdot y_{\text{imag}}) + (x_{\text{real}} + i \cdot x_{\text{imag}}) = (y_{\text{real}} + x_{\text{real}})_{\text{real}} + i \cdot (y_{\text{imag}} + x_{\text{imag}})_{\text{imag}}$
where x_{real} represents the real part of x and x_{imag} represents the imaginary part of x.

group and therefore there is only one vectorized basic computation to perform. In each successive stage, the vector length is halved and the number of butterfly groups doubles. For every stage, each butterfly group requires using a different root of unity for its basic computation. So for this example, stage 1 uses only one root of unity, stage 2 uses two roots of unity, stage 3 uses four roots of unity, and stage 4 uses eight roots of unity.

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). Therefore the final step in this *naïve* 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. This bit reversal can be accomplished in our implementation of the *naïve* algorithm by simply storing the results sitting in the four vector registers using an indexed store. To do the indexed store, an array of the appropriate offsets is first loaded into a vector register and then these offsets are used to compute the appropriate bit reversed address for each result element to be stored.

4.1.2 Alternatives to the Naïve Algorithm

This section clarifies why the *naïve* algorithm described above was the appropriate one upon which to build. Observe that our implementation of the *naïve* algorithm uses unit stride loads and stores for all stages but the last one. After the last stage an indexed store is used in order to place the final results in memory at a different location from the input points in bit reversed order.[13]

There are two alternatives to doing the bit reversed swapping — the indexed store — after the last stage. The first is to load the points in bit reversed order *before* the first stage. This requires a slight change in the algorithm. Specifically the first stage would have N/2 butterfly groups, each with a VL$=1$. In this case the number of butterfly groups would halve and the VL would double for each successive stage, until the last stage would have one butterfly group with a VL=N/2. This alternative is the mirror image of the one explained above which we chose to implement. Both versions must access memory using a random access pattern. As mentioned above, our implementation uses an indexed store after the last stage and unit-stride loads and stores for all the remaining stages. The alternative algorithm just described uses indexed loads before its first stage and unit-stride loads and stores for all of its remaining stages. Therefore the two algorithms are exact mirrors of each other, so our choice to use one over the other will have no impact on the performance data we collected.

The second alternative to doing the bit reversed swapping after the last stage is to repeatedly rearrange the order of the elements between the stages. This can be accomplished by either an indexed store of the intermediate results after completing each stage, or an indexed load of these intermediate values before doing each of the following stages. By doing this intermediate element rearranging between stages, the results after the last stage will then be in the correct bit reversed order. This second alternative would use a unit stride load before the first stage and a unit stride store after the last stage. Each intervening stage, however, would either use an indexed load followed by a unit-stride store, or a unit-stride load followed by an indexed store. Besides being much more complicated to code, this second algorithm alternative uses more indexed operations than either the first alternative or our *naïve* algorithm uses, so we did not use it as the starting point for our research.

4.1.3 Performance of the Naïve Algorithm

Throughout this section all intermediate performance figures that appear assume 32-bit, floating-point, single precision, complex arithmetic and give performance numbers for FFT sizes that are assumed to be powers of 2 and that range between 4 and 8192. For all the figures in this section it was also assumed that there were 32MB of memory divided into 16 banks with no subbanks.[14]

[13] The bit reversal uses twice the memory space as an in-place algorithm, which stores the results back into the original input array.

[14] 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. Thus these computations are not included in our performance results.

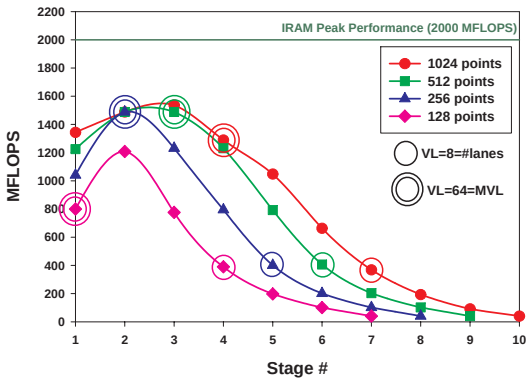


Figure 2: **Performance in MFLOP/s of each stage of a floating-point, single precision, 32-bit, N-point, radix-2 FFT using the *naïve* FFT algorithm on VIRAM for N = 128, 256, 512, and 1024. The circled points indicate the stage at which the VL=8=the number of virtual lanes. The double circles indicate the stage at which VL=64=MVL. The 2 GFLOP/s line shows the maximum performance for the 32-bit, floating-point FFT computation that might be ideally attainable on VIRAM, taking into account only the arithmetic operations. Memory: 32MB, 16 Banks, No Subbanks.**

Figure 2 shows the performance of the *naïve* 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 the vector length is halved from one stage to the next, the MFLOP/s rate dramatically decreases as well.

The 2 GFLOP/s line in Figure 2 shows the maximum performance for the radix-2 complex, 32-bit, floating-point FFT computation that might be ideally attainable on VIRAM, taking into account only the arithmetic operations. As explained above, the VIRAM hardware peak of 3.2 GFLOP/s for single precision floating-point can only be obtained when multiply-add instructions are used; most other single precision floating-point instructions have a hardware limit of 1.6 GFLOP/s.[15] 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 floating-point arithmetic operations, which results in the 2 GFLOP/s maximum for this mix of instructions.[16]

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 floating-point 32-bit complex *naïve* 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.**

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. For all the FFT sizes 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. Since the vector length is greater or equal to MVL = 64 for the earlier stages, these earlier stages (after the first) achieve a respectable rate of 1400-1800 MFLOP/s, but the rates for the later stages where the VL drops below MVL 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 *naïve* algorithm spends computing in all the stages that have 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.

[15] 1.6 GFLOP/s = 8 virtual lanes * 1 floating-point functional unit/virtual lane * 1 FP operation/cycle * 200 Mcycles/second

[16] 2 GFLOP/s = 2 multiply-adds(MAs)/8 total * 3.2 GFLOP/s +6 non-MAs/8 total ∗1.6 GFLOP/s

4.2 Optimization #1: Auto-increment addressing

In order to optimize the *naïve* FFT algorithm for VIRAM, the performance degradation observed in Figure 2 for the stages whose vector lengths were shorter than MVL has to be reversed. With this as the focus, our first optimization utilizes 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 the auto-increment feature, scalar code is needed to calculate the next address to be accessed. The overhead 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, floating-point element operations per cycle, and since the scalar unit can complete only 2 per cycle, there must be 4 vector element operations for every one scalar operation for the scalar operation to be hidden. Vector computations with short vector lengths contain fewer vector element 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 *naïve* algorithm because there is less scalar code to hide.

4.2.1 Performance of the Auto-increment Optimization

In Figure 4 the MFLOP/s rates for 32-bit, floating-point, complex FFTs ranging in size from 4- to 8192-points are presented in table form. The second, third, and fourth columns contain the MFLOP/s rate achieved by different versions of the *naïve* algorithm; from left to right these columns hold results generated by: 1) the original *naïve* algorithm implemented without the bit reversal rearrangement of the final points and without the auto-incrementing (**No BR, No AI**); 2) the same algorithm as (1) with auto-incrementing added (**No BR, AI**); 3) the same algorithm as (2) with the bit reversal added, (**BR, AI**). The rightmost two columns give the percentage of MFLOP/s gained by adding: 1) only the auto-increment feature to the original *naïve* algorithm, and 2) both the auto-increment feature and the bit reversing to the original *naïve* algorithm.

Unfortunately, there is only a 6% to 20% performance improvement realized by utilizing the auto-increment feature. For example, for a 1024-point FFT, the overall performance using the *naïve* 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, the peak improvement of 20% is reached for FFTs of size 512, 256, and 128. Reducing the number of scalar operations for FFTs of these sizes tips the ratio of scalar to vector operations in just the right direction to realize this 20% benefit. However, when this ratio either grows or shrinks auto-incrementing ceases to have a significant positive effect on performance.

Inspecting Figure 4 we observe that the improvement in performance is lower for FFT sizes less than 128. Since 100% of the work done by these smaller FFTs is being done in stages whose vector lengths are shorter than MVL, the ratio of scalar to vector operations is quite large. This is because the loops containing shorter vector lengths do fewer vector operations but still require the same number of loop-controlling scalar operations as the loops containing longer vector lengths and more vector operations. The auto-incrementing does reduce the number of scalar operations in each of these stages, but for the smaller FFTs this small reduction in the number of scalar operations doesn't really significantly change the ratio of scalar to vector operations enough to show a decent improvement in the overall performance. In other words, the overall performance is still limited by the comparitively large number of scalar operations that remain even after employing the auto-increment feature.

A good analogy of what is happening is that of dropping some coloring agent into a large bucket of water. If we take two such buckets that have an equal amount of water (scalar operations) in them and if we drop the same amount of coloring agent (vector operations) into both buckets, the water in both buckets will be the exact same color. If we start again but remove a few teaspoons of water (the scalar operations

# FFT points	No BR No AI	No BR AI	BR & AI	% AI	% BR
8192	247	264	253	7%	2%
1024	202	225	206	11%	2%
512	186	223	196	20%	5%
256	166	200	175	20%	5%
128	146	175	154	20%	5%
64	123	145	129	15%	5%
32	100	118	104	18%	4%
16	78	90	79	15%	1%
8	56	62	56	11%	0%
4	35	37	35	6%	0%

Figure 4: **This table reports the MFLOP/s rates for 32-bit floating-point, complex FFTs ranging in size from 4 to 8192 points. The second, third, and fourth columns contain the MFLOP/s rate achieved by different versions of the *naïve* algorithm. From left to right these columns hold results generated by: 1) the original *naïve* algorithm implemented without the bit reversal rearrangement of the final points and without the auto-incrementing (No BR, No AI); 2) the same algorithm as (1) with auto-incrementing added (No BR, AI); 3) the same algorithm as (2) with the bit reversal added, (BR, AI). The rightmost two columns give the percentage of MFLOP/s gained by adding: 1) only the auto-increment feature to the original *naïve* algorithm, and 2) both the auto-increment feature and the bit reversing to the original *naïve* algorithm.**

eliminated by using the auto-increment feature) from one of the buckets, the color in the water will still appear to be the exact same color although we know that one of them is slightly less diluted and therefore darker in color (better performance) than the other.

As we see from Figure 4 the improvement in performance from auto-incrementing is also lower for FFT sizes higher than 512 because these FFTs have many stages that operate with VL=MVL. Consequently during these stages that have VL=MVL, these FFTs generate many vector operations, which hide most of the scalar operations. Therefore as the FFT size increases, the vector operations hide all the scalar operations that are possible to hide. Thus the benefit to be gained from auto-incrementing, whose primary purpose is to reduce the number of scalar operations, disappears.

This behavior can be visually seen in Figure 5, which shows the MFLOP/s performance for each FFT size for the *naïve* algorithm implemented in three ways. At this point, we will only concern ourselves with the top and bottom curves in the figure: the upper curve represents the performance of the *naïve* algorithm with auto-incrementing and the bottom curve represents the performance of the *naïve* algorithm without autoincrementing. Both algorithms used to generate these curves omit the final bit-reversing. The x-axis uses a $\log_2$ scale for the FFT sizes, which range from N = 4 to 8192.

The two curves practically coincide for the smaller FFT sizes. They move farther apart for a short distance where the 20% improvement is realized, and then they begin to converge so the gap between them steadily decreases. We can therefore infer from Figure 5 that as the FFT size increases, there is a high probability that the increase in MFLOP/s performance contributed by auto-incrementing will become negligible and insignificant. For this reason we must find a more effective way to optimize the performance of the *naïve* algorithm.

4.2.2 Bit Reversal Rearrangement

As just described, the top and bottom curves in Figure 5 represent the performance of the *naïve* algorithm when 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

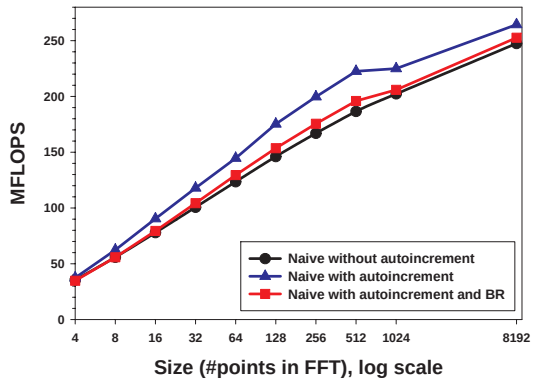


Figure 5: **VIRAM performance, measured in MFLOP/s, of three implementations of the *naïve* FFT algorithm for various FFT sizes. The first implementation, *Naïve without autoincrement,* is the original *naïve* algorithm with no bit reversal and no auto-incrementing. The second implementation, *Naïve with autoincrement,* just adds auto-incrementing to the first. The third implementation, *Naïve with autonincrement and BR,* adds the bit reversal rearrangement to the second. A $\log_2$ scale is used for the x-axis. Memory: 32MB, 16 Banks, No Subbanks.**

a significant amount of address (pointer) arithmetic that is computed using scalar code. Adding more scalar code to the original *naïve* algorithm would only exacerbate the problem that we are 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. For this reason we did not implement a version of the original, non-auto-incrementing *naïve* algorithm with the bit reversal rearrangement. Instead we added the bit reversing to the auto-incrementing *naïve* algorithm.

The data in Figure 4 corroborate the claim that adding the bit reversal rearrangement does exacerbate the scalar code bloat and therefore the performance of any version to which it is added. We see that adding it to the auto-increment version of the *naïve* algorithm adds an average of approximately 12% overhead to the MFLOP/s rate of the auto-increment version slowing it down by up to a factor of 1.09.

We now examine the middle curve in Figure 5 which shows the MFLOP/s performance of the *naïve* algorithm that has been implemented with both auto-incrementing and the bit reversal rearrangement. Although this third curve is between the two other curves, it is very close to the bottom curve.

The actual percent improvement of the auto-incrementing, bit reversing implementation over the original *naïve* implementation that had no auto-incrementing and no bit reversing is listed in the rightmost column of Figure 4. Although the improvement follows the same pattern as that described above for auto-increment in that the FFTs of size 64 to 512 have the peak improvement of 5% with the smaller and larger FFT sizes showing improvements less than 5%, the improvement percentages are much smaller. We can thus conclude that whatever performance is gained from utilizing the auto-increment feature is all but lost doing the final bit reversal rearrangement.

4.3 Optimization #2: Transpose-based algorithm

Due to the negligible performance gains from the use of the auto-increment feature in the *naïve* algorithm, alternative approaches that might yield a more significant gain need to be considered if the *naïve* algorithm is to be optimized. One such approach is to reorganize the data layout in memory in order to maximize vector lengths in the later stages of an FFT. In particular, 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 *naïve* algorithm. However, to keep full vector lengths, one may have to do several in-memory transposes, *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. Furthermore, since an in-memory transpose in the VIRAM architecture is implemented by doing an indexed load or an indexed store, both of which cause functional unit pipeline stalls as described above, it became clear that in-memory transposes would have a negative effect on the performance of the *naïve* algorithm. Therefore this in-memory transpose alternative was not considered a viable choice for optimizing the

naïve algorithm on VIRAM.

4.3.1 In-register transposes

However, 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 *naïve* algorithm, thus eliminating our short vector length problem. Our new in-register transpose algorithm, called the *"vhalf" algorithm*, uses the *naïve* algorithm 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 the vector length 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 and each butterfly group uses a different root of unity. 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, another register could have its first 32 elements equal to the first root of unity and its second 32 elements equal to the second root of unity so that two butterfly groups could be 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 using another vector register having 16 copies of each of the first four roots of unity. In this manner, each basic computation could be performed on vector registers with VL = MVL, so the algorithm would be optimized for all of the short vector length stages. Note that in the VIRAM architecture, to set the vector registers up with the proper pattern of the roots of unity, an indexed load must be used.

4.3.2 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 *naïve* algorithm using a maximum vector length of 8 as pictured in Figure 1. Since stage 1 in this example has one butterfly group with 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 first stage in which the in-vector register transposes begin. The first stage 2 butterfly group needs to pair elements 0-3 with their corresponding elements 4-7, while its second butterfly group needs to pair elements 8-11 with their corresponding elements 12-15 as depicted under stage 2 of Figure 1.

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 stage 2 butterfly groups to be done together using one vectorized basic computation with a VL of 8, and it extends this concept to all the remaining stages so that no matter which stage is being computed VL is always equal to MVL and multiple butterfly groups are computed using one basic computation.

Consequently, after the stage 2 in-vector register 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 the last four elements, 4-7, in vr1 with the first four elements, 8-11, in vr2, and it

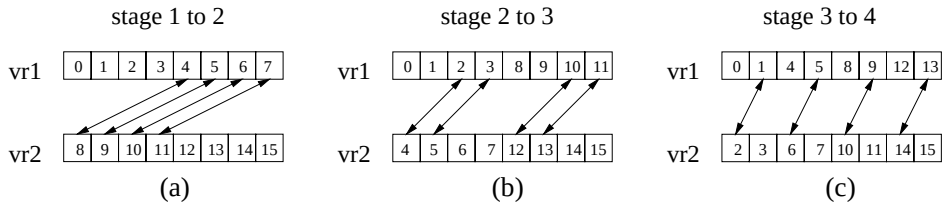


Figure 6: **In-register movements for the final 3 stages of a 16-point FFT, illustrated with 8 elements per register. Each diagram illustrates which points occupy each of the vector registers' element slots before the indicated swap. This means that diagram (b) illustrates the position of the elements after the swap indicated in diagram (a), and diagram (c) illustrates the position of the elements after the swap indicated in diagram (b). Although it is not shown in the figure, after the swap indicated by diagram (c), all the even numbered points are in vr1 and all the odd numbered points are in vr2. 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**

does an identical swap for the imaginary values in vr3 and vr4.

This element swapping within the vector registers from stage 1 to stage 2 is illustrated in Figure 6. In this Figure, diagram (a) shows the points that are occupying each element slot for vr1 and vr2 before the indicated swap with arrows indicating the elements that will be swapped, and diagram (b) shows the points that are occupying the element slots for both vector registers after the swap in diagram (a) has occurred. Notice that for stage 2, with the elements swapped as in diagram (b) and with another vector register holding the first root of unity in the first four element positions and the second root of unity in the second four element positions, the vectorized basic computation is being done once for two shorter butterfly groups instead of twice, which illustrates the second optimization.

After the stage 2 computations have thus been performed 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 illustrated by diagrams (b) and (c) in Figure 6.

Similarly, diagrams (b) and (c) illustrate the points that are occupying each of the vector registers' element slots before the indicated swaps and arrows indicated the elements that will be swapped. Diagram (c) shows the points that are occupying the element slots of vr1 and vr2 after the swap indicated by diagram (b) has happened. Although not illustrated in Figure 6, after the swap indicated by diagram (c) has occurred, vr1 holds all the even numbered points and vr2 holds all the odd numbered points. 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. What is different is that multiple butterfly groups are being computed using one vectorized basic computation (with VL = MVL = 8 for this example).

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 real and imaginary elements, *i.e.*, the next *outer iteration* butterfly group, would be loaded into the four registers and stages 2 through 4 would be performed for them in a similar fashion. This procedure would iterate until the computations for all the remaining groups of 16 elements had been completed, *i.e.* all points in the N-point FFT had been computed. 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 $MVL * 2$).

4.3.3 Implementation of The 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. Recall

that Figure 6 shows the desired pattern of data movement between register pairs in the final stages of the 16-point 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 a specified number of contiguous 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 (vindex) indicates the number of contiguous elements to be moved as well as the number to skip when more than one group of elements will be moved. This number must be a power of two and is expressed as an exponent. For instance if 32 elements are to be moved, then the number in the control register should be 5, because $2^5 = 32$. If 16 contiguous elements are to be moved using **`vhalfup`** and MVL = 64, the number in the control register should be 4. This would cause the first group of 16 elements to be moved into higher numbered slots, the next group of 16 elements to be untouched, the third group of 16 to similarly be moved to higher numbered slots, and the last group of 16 to be left alone. The pattern would be slightly different for **`vhalfdn`**. In that case the first group of 16 elements would be untouched, the second group would be moved to lower numbered slots, the third group would be left untouched, and the fourth group would be similarly moved to lower numbered slots.

Using the same 16-point FFT example as above, the VIRAM code with the new **`vhalfup`** and **`vhalfdn`** instructions that will accomplish the stage 2 swap described above is shown in Figure 7.

For stage 2 the vector control register (vindex) is set equal to 2, since we want to move 4 contiguous elements and $2^2 = 4$. To do the stage 2 swaps, first vr1 is copied to a temporary vector register, vr5, using the `vmerge.vv` instruction. The **`vhalfup`** instruction moves the first four elements in vr2 into the last four element slots of vr1. The **`vhalfdn`** instruction then moves the last four elements in vr5, which are the original last four elements that were in vr1, into the first four element slots of vr2. This then completes the swap for stage 2.

Continuing this pattern for stage 3, the vector control register, vindex, is set to 1, since we want to move 2 conntiguous elements and $2^1 = 2$. The remaining code is identical to the code given above, however the swapping pattern is very different. Refer to diagram (b) in Figure 6 for a visual illustration of the stage 3 swapping pattern. In particular, after copying the vr1 elements into the temporary vector register, vr5, as before, the **`vhalfup`** causes the first and second elements of vr2 to be put into the third and fourth element slots of vr1 and the fifth and sixth elements of vr2 to be put into the seventh and eigth element slots of vr1. This is then followed by the **`vhalfdn`** which causes the third and fourth elements of vr5, which are identical to the original vr1 contents, to be put into the first and second element slots of vr2 and the seventh and eighth elements of vr5 to be put into the fifth and sixth element slots of vr2. This then completes the swap for stage 3.

Diagram (c) in Figure 6 shows which points occupy the elements slots for vr1 and vr2 after the stage 3 swaps have been completed. Notice that `vhalfup/dn` move mulptiple groups of contiguous elements and the number of groups moved is controlled by the value in the vector control register, vindex. With these new instructions, any FFT of size 128 or larger (for 32-bit values) and of size 256 or larger (for 16-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 one. 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

```
li          t0, 2          # Loads 2 into general purpose register t0
ctc2        t0, vindex     # Loads 2 into vector control register vindex

vmerge.vv   vr5,vr5,vr1    # Copies all the elements in vector register 1
                           # into temporary vector register 5

vhalfup     vr1,vr2        # Moves the first 4 elements of vector
                           # register 2 {8-11) into the last 4 element
                           # slots of vector register 1 {4-7)


vhalfdn     vr2,vr5        # Moves the last 4 elements of temporary
                           # vector register 5 (4-7) into the first 4
                           # element slots of vector register 2 (0-3)
```

Figure 7: **This code illustrates the instructions that are necessary to do the stage 1 to stage 2 swap shown in Figure 6 for the 16-point FFT example. The vector control register, `vindex`, indicates how many and which elements the `vhalfup` and `vhalfdn` operations should move. The code assumes that vr1 holds the first 8 elements (0-7) and vr2 holds the second 8 elements (8-15). vr5 is a temporary vector register used to hold a copy of vr1.**

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 hardware cost and much to be gained in implementing them.

4.3.4 In-register Transpose Performance Results

Figure 8 shows the MFLOP/s rate for each stage of the optimized *vhalf* FFT algorithm. This implementation includes the *new* **`vhalfup`** and **`vhalfdn`** instructions, the auto-incrementing, software pipelining, code scheduling, and it uses indexed loads in each of the last 6 stages to load the roots of unity into the vector registers with the correct pattern; the final output points are bit reverse rearranged. The double circles indicate the stages in which VL = MVL = 64, and the open circles indicate the stages in which VL = 8, the number of virtual lanes. As in Figure 2, the 2 GFLOP/s line on the plot shows the maximum performance that might be ideally attainable on VIRAM, taking into account only the arithmetic operations.

As we observed in Figure 2, for all the FFT sizes but the 128-point, the first stage is somewhat slower than the second because the program start-up overhead is included with the first stage only, and as long as the VL is larger or equal to MVL = 64, the performance is maintained at a respectable 1.8 GFLOP/s. The performance curve for the 128-point FFT is the exception because its second stage, having a vector length of $MVL/2 = 32$, does not meet this criteria. In fact it is the first stage of the 128-point FFT that has a $VL = 64 = MVL$, and were it not for the start up overhead being included, its first stage GFLOP/s would be close to the 1.8 mark as well.

Recall that the first stage in which the in-register transposes occur is the stage in which VL = 32 = MVL/2. For each of the curves we see a steep dip in the performance of this stage because the GFLOP/s rate for this stage includes the overhead for switching from the *naïve* algorithm to the *vhalf* algorithm, which incorporates the *vhalf* algorithm setup for the roots of unity and the bit reversal. The stage immediately following this first vhalf stage recovers, and for all the curves the next four stages maintain a performance of 1.2 GFLOP/s, which indicates 60% uti-

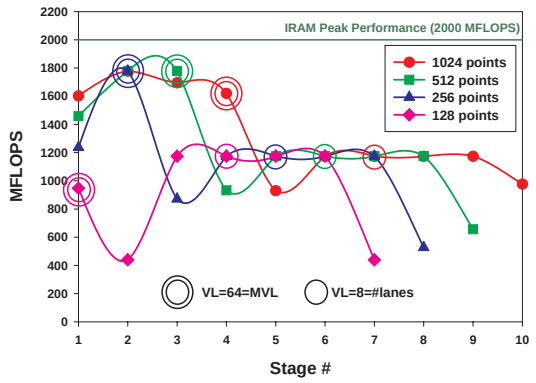


Figure 8: **Performance of each stage of a floating point, single precision, 32-bit, N-point FFT using the optimized *vhalf* FFT algorithm on VIRAM for N = 128, 256, 512, and 1024. The circled points indicate the stage at which the VL = 8 = the number of virtual lanes. The double circles indicate the stage at which VL = 64 = MVL. The 2 GFLOP/s line shows the maximum performance for the 32-bit, floating-point FFT computation that might be ideally attainable on VIRAM, taking into account only the arithmetic operations. Memory: 32MB, 16 Banks, No Subbanks.**

lization during these vhalf stages. The reasons that the GFLOP/s rate for these stages is not closer to the VIRAM peak of 2.0 GFLOP/s will be discussed in Section 6.

Notice also that for all the curves in Figure 8, the GFLOP/s rate for the last stage shows a steep degradation. This is the effect of the bit reversal that is being done at the end of the last stage. Specifically, the indexed store of the contents of each of the four vector registers that hold the results is slowing the memory functional unit pipeline, and therefore the arithmetic functional unit pipeline, by at least a factor of two. This occurs because the arithmetic functional unit pipeline is not decoupled from the memory functional unit pipeline. If the memory unit experiences any bank conflicts from the random accesses into memory, then these conflicts will stall the memory functional unit even more, which in turn will stall the arithmetic functional units for the same number of cycles.

Figure 8 shows the MFLOP/s rate for the last stage of the 128-point FFT to be 440, of the 256-point FFT to be 527, of the 512-point FFT to be 656, and of the 1024-point FFT to be 977. This tells us the MFLOP/s rate for the last stage is getting better as the number of points in the FFT increases. Since the ratio of floating-point operations to store operations is always 8 to 4, we would expect the MFLOP/s rate for this last stage to be the same regardless of the size of the FFT, and therefore we should not be seeing this behavior. There are two possible reasons we see this improvement in the vhalf last stage MFLOP/s rate as the FFT sizes increase.

The first reason for the increase in performance of the last stage as the FFT size increases is that larger point FFTs provide more chance for amortizing the cost of certain memory operations. In particular, the latency of the last store of the last vhalf stage is completely exposed for the 128-point FFT, since that store is the last operation of the entire FFT calculation, and the stage is not considered complete until all its operations have been completed. When the FFT size is 256-points or larger, the last memory store for the last vhalf stage is executed multiple times, once for each 128-point group, or, in other words once for each outer loop iteration. Since the last vhalf stage is considered complete when the last outer loop iteration completes, only the last store in the last vhalf stage of the last outer loop iteration has its latency exposed; the latencies of the last stores of the last vhalf stage in earlier iterations are hidden by the later iterations, since there are multiple functional units working at the same time. Since the number of outer loop iterations increases with the size of the FFT, larger point FFTs have more opportunity for amortizing the memory store overhead and correspondingly the last stages of the larger FFTs have higher performances than those of the smaller FFTs.

The second reason that the MFLOP/s rate for the last stage gets better as the size of the FFT increases has to do with memory bank conflicts. Specifically, since we know that memory bank conflicts will be dependent on the layout of the data in memory, and since the data will have a different layout for each different size FFT, the differences we observe in the MFLOP/s rate for the last stage of the various sized FFTs are partially attributable to different patterns of memory conflicts. Because the points of a smaller FFT are closer together in memory than the points of

a larger one, there is a higher probability that the consecutive addresses generated by the indexed stores for the smaller FFT sizes are in the same bank and therefore these smaller FFTs experience more memory bank conflicts during the indexed stores of the last stage than the larger ones do. For both of these reasons the last stage of the 1024-point FFT completes with a better MFLOP/s rate than the others.

Notwithstanding the slowdown in the last stage of each curve, Figure 8 verifies that with the new *vhalf* algorithm the total time is no longer dominated by the time spent in the later stages of the FFT where the vector length falls below MVL. As a matter of fact, the time spent in the last 6 stages drops from the 94% that we saw with the *naïve* algorithm to 70% with the *vhalf* algorithm, which includes the bit reversal slowdown, and which comes closer to the 60% of total work that the last 6 stages represent in a 10 stage, 1024-point FFT. Since we observe no degradation in performance similar to what we saw with the *naïve* algorithm when the vector lengths become smaller than MVL, clearly the in-register transposes have solved the short vector length problem exhibited by the *naïve* algorithm.

To corroborate this observation, Figure 9 compares the overall MFLOP/s rate for each size FFT for the two implementations: the *naïve* algorithm and the *vhalf* algorithm. Both implementations do the bit reversing and the auto-incrementing. The *vhalf* algorithm in this figure is the same as the one in Figure 8. As in previous figures, the 2 GFLOP/s line on the plot shows the maximum performance that might be ideally attainable on VIRAM, taking into account only the arithmetic operations and the x-axis uses a $\log_2$ scale for the FFT sizes.

In Figure 9 observe that for all the FFT sizes, the *vhalf* 32-bit, floating-point implementation has a much higher MFLOP/s rate than the corresponding ones from the *naïve* implementation, so clearly the in-register transposes are a useful feature for FFTs on VIRAM. Looking at Figure 9, we also observe that as the FFT size increases, the *vhalf* implementation yields a higher MFLOP/s rate than that of the previous smaller size FFT until the 1024-point FFT, after which it begins to degrade for the 2048-point FFT and then seriously drops for 4096- and 8192-point FFTs. The large drop in performance starting at an FFT size of 4096 for the *vhalf* algorithm is due to memory bank conflicts. We discuss this phenomenon and explain its performance impact later in Section 6. Nonetheless, even the 1024-point rate of 1.267 GFLOP/s, which is the highest rate in the figure, represents only a 63% utilization of the VIRAM full potential. This behavior is due to architectural idiosyncrasies that will also be discussed more fully in Section 6.

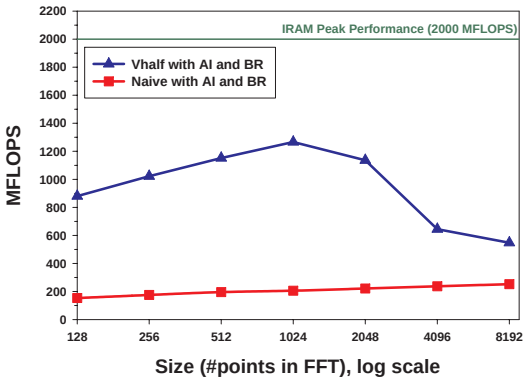


Figure 9: **Comparison of the performance in MFLOP/s of a floating point, single precision, 32-bit, N-point FFT *naïve* implementation with the optimized *vhalf* FFT implementation on VIRAM for N = 128, 256, 512, 1024, 2048, 4096, and 8192. The 2 GFLOP/s line shows the maximum performance for the 32-bit, floating-point FFT computation that might be ideally attainable on VIRAM, taking into account only the arithmetic operations. A $\log_2$ scale is used for the x-axis. Memory: 32MB, 16 Banks, No Subbanks.**

5 Fixed-Point FFT Vector Implementation

In this section we describe how we extended the fully-optimized floating-point algorithm to the 16-bit, fixed-point domain in an attempt to squeeze even more performance out of the VIRAM architecture.

5.1 Adaptation For Fixed-Point Data

As a last step in implementating an efficient FFT algorithm for VIRAM, we adapted the optimized floating-point algorithm developed in the previous

sections to operate on fixed-point data. In doing so, we hoped that we would be able to leverage VIRAM's increased integer performance (relative to floating-point performance) to achieve a commensurate increase in the performance of the FFT. In particular, recall that each VIRAM virtual lane has two integer functional units, only one of which also serves as the floating-point functional unit. Since fixed-point data is done using integer computations, it seemed reasonable to expect the fixed point version of the *vhalf* algorithm to double the performance of the floating-point version.

Converting the floating-point algorithm to a fixed-point one required changes in two areas of the algorithm. First the data size and type had to be converted from 32-bit floating-point, single precision to 16-bit integers. In fixed-point mode, each integer has an assumed binary point position, and its location after each computation must be carefully tracked. Furthermore, care must be taken after each computation to insure that the result does not overflow the 16 bits. For these reasons, the second area that had to be changed to convert the floating-point algorithm to a fixed-point one was the basic computation. We will discuss each of these changes in turn in the following sections.

5.2 Fixed-Point Data Size and Type

Our implementation of the fixed-point FFT algorithm is designed to operate on input points that are complex numbers where both the real and imaginary parts are postive numbers between 0 and 2^{15}-1. Therefore the binary point for each part is assumed to be to the right of the rightmost binary digit, which we illustrate as follows: Sbbb bbbb bbbb bbbb$_{\odot}$, where S is the sign bit and $\odot$ is where the binary point is assumed to be.

When computing any FFT, the roots of unity are always between 1 and -1 inclusively, and, as stated above, for all our FFT algorithms they are precomputed. As part of this precomputation for the fixed-point algorithm, each root of unity is first computed in floating-point form, then converted to integer form and shifted to the left 15 binary positions. After a correction is done for the special case of 1, we can thus assume that the binary point for both parts of each root of unity is between the leftmost binary digit, which is the sign bit, and the binary digit immediately to its right, which we illustrate as follows: S$_{\odot}$bbb bbbb bbbb bbbb. These are the assumptions we made about the input points and the roots of unity when we converted the complex floating-point basic computation into a comparable fixed-point basic computation.

5.3 The Fixed-Point Basic Computation

Below, we repeat the illustration from Section 3 of the complex floating-point basic computation between two complex points, X_0 and $X_{N/2}$, which we will now call the *top* point and the *bottom* point, respectively:

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

where ω is one of the roots of unity.

As can be seen, the basic computation consists of three complex operations; the first is a complex multiply between a root of unity and the *bottom* point, which produces a complex product; the second and the third are a complex add and a complex subtract between the *top* point and the newly computed complex product. The possibility of an overflow exists for all three of these complex computations, so first we discuss the fixed-point complex multiply and then the fixed-point complex add and subtract.

5.3.1 The Fixed-Point Complex Multiply

In the complex multiply, since the roots of unity are between 1 and -1 inclusively, the magnitude of the complex product can never be larger than the magnitude of the *bottom* point, so no overflow will occur. However, recall that the complex multiply consists of 4 scalar multiplies. Two of the resulting products are then added together, and the remaining two products are subtracted, one from the other to obtain the desired complex product.[17]

Consequently the possibility exists that an overflow might occur in one or more of these intermediary products. However, should an intermediary

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

product overflow, it will not cause an incorrect result for the following reasons; the final product cannot be greater in magnitude than that of the *bottom* point; the way that the two's complements wrap from positive to negative and back to positive insures that the sum or difference of these intermediary products, when left in a *wrapped* state if they happened to have overflowed, are nevertheless correct in their magnitude and sign after the entire complex multiplication has been completed.[18] Therefore we can safely assume that the complex product will not exceed the 16-bit data width.

However, when doing the complex multiplication, precision must be considered in addition to magnitude and overflow. Recall that the real and imaginary parts of the bottom point have the form Sbbb bbbb bbbb bbbb$_{\odot}$, while the corresponding parts of the root of unity have the form S$_{\odot}$bbb bbbb bbbb bbbb. Since the product will have the same magnitude as the bottom point, it will have 32 total bits and have the form SSbb bbbb bbbb bbbb b$_{\odot}$bbb bbbb bbbb bbbb.

In the VIRAM architecture this multiplication is implemented using the **`vmulhi`** instruction that takes two vector register operands each with 16-bit elements and returns the high order 16 bits of the product for each corresponding element operand pair. Thus the binary point for each of these products is assumed to follow a non-existent bit immediately to the right of the rightmost bit, which is depicted by the long form of the product given above. The magnitude of the product is the same as that of the bottom point, but one binary bit of precision has been lost in the product as a result of the 16-bit multiplication. In effect we have shifted the product one bit to the right, which is the same as dividing it by 2.

[18] An example of this wrap around effect follows. In two's complement representation, usng 4 binary digits, one can represent decimal values in the following range: -8,...,-1,0,1,...,7. If we multiply $2 * 7 = 14$, although the 2 and the 7 are within this interval, the $14 = 1110$ is technically an overflow. So the 14 here is an example of an intermediary product that has overflowed but has been left in its *wrapped* state. Assuming -8 is a second intermediary product, the sum, $14 + (-8) = 6$, produces a result, which expressed in binary is 1110 + 1000 = 1 0110, where the leftmost 1, which is in the carry out position, is completely dropped. The remaining 0110 is equal to 6, which is within the interval and which is the correct final sum despite the intermediary overflow and the 1 dropped from the carry out position.

5.3.2 The Fixed-Point Complex Addition and Subtraction

As stated above, once the multiplication has been computed, the resulting complex product must be added to and subtracted from the top point to complete the basic computation. Since the product has been shifted one bit to the right, before these calculations can be performed the assumed binary points of both operands must be aligned. Therefore the top point's real and imaginary parts must be shifted one bit to the right so that both operands have the form, SSbb bbbb bbbb bbbb b$_{\odot}$. A side benefit of this shift is that the results can then accomodate a carry to the left if the addition or subtraction causes the magnitude of the result to increase by one bit. Thus the new intermediate top and bottom points will have the form Sbbb bbbb bbbb bbbb b$_{\odot}$, where the rightmost bit is dropped after the basic computation for the first stage has of the fixed-point algorithm has been completed.

In subsequent stages, this pattern repeats. The input points for each stage start out with the assumed binary point moved one more binary position to the right than was the case for the inputs of the stage immediately preceding it. Thus for each stage of a fixed-point FFT, one bit of precision and magnitude is lost. In other words the results of a fixed-point FFT with N points must be shifted $\log_2 N$ bits to the left (or multiplied by $2^{\log_2 N}$) in order to obtain the correct magnitude, since the final results must compensate for these repeated shifts (equivalent to repeated divisions by 2) of the assumed binary point for each stage.

5.3.3 The Fixed-Point Rounding Mode

Notice that the implementation of the fixed-point basic computation just described does not use the multiply-add instruction that the floating-point version used. Instead the **`vmulhi`** instructon was utilized because it does the necessary rounding and truncation to keep the fixed-point results from overflowing. VIRAM allows the programmer to choose one of four kinds of rounding modes that, once chosen, will be subsequently used by all fixed-point instructions that provide rounding as part of their specification. For our implementation we used the *round*

to nearest even mode so that there would be no bias caused by constantly rounding in one direction. The *round to nearest even* mode avoids such biases by sometimes rounding up and sometimes rounding down since the rounding is done in the direction of whichever of the two even numbers on either side of the digit being rounded is closest numerically to it.

5.4 The Ramifications of Using 16-bit Wide Data

Besides making the changes to the basic computation just described, one additional change is required to actually complete the convertion of the floating-point implementation of the optimized algorithm to a fixed-point version; the **vpw** vector control register needs to be reset to indicate 16-bit wide data instead of 32-bit wide data. Although this is a simple task, there are several important ramifications from making such a change. Specifically, the number of virtual lanes doubles from 8 to 16, and the number of elements per vector register, the MVL, doubles from 64 to 128. In addition, recall that in the optimized algorithm, the stage whose vector length equals MVL/2 is the first stage for which the in-vector register transposes are utilized. For the floating-point version, since $MVL/2 = 32$ the in-register transposes are utilized in the last 6 stages; for the fixed-point version, however, since $MVL/2 = 64$, the in-register transposes are done for the last 7 stages.

Furthermore, just as they are used in the optimized *vhalf* floating-point implementation, indexed accesses are similarly used in the last stages of the *vhalf* fixed-point implementation. Specifically, indexed loads are used in the vhalf stages to set up the proper pattern of the real and imaginary roots of unity in vector registers in order to perform only one vectorized basic computation on multiple butterfly groups and thereby achieve VL = MVL for all computations. In addition, recall that indexed stores are used at the end of the last stage to do the bit reversal rearrangement.

Therefore, another important ramification of changing the **vpw** from a 32-bit width to a 16-bit width concerns these indexed memory accesses. In particular, because only four addresses can be generated per cycle, for 32-bit data the indexed memory accesses are two times slower than the unit-stride accesses, but for 16-bit data they are four times slower than the unit-stride accesses. As we shall see in Section 6, this ramification has an impact on the performance of the fixed-point implementation of the *vhalf* algorithm.

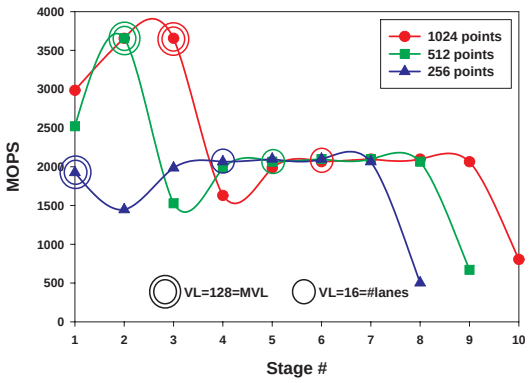


Figure 10: **Performance of each stage of a fixed-point, 16-bit, N-point FFT using the optimized *vhalf* FFT algorithm on VIRAM for N = 256, 512, and 1024. The circled points indicate the stage at which the VL = 16 = the number of virtual lanes. The double circles indicate the stage at which VL = 128 = MVL. Memory: 32MB, 16 Banks, No Subbanks.**

5.5 Performance of the Fixed-point Vhalf Implementation

As with the floating-point implementation, we present the fixed-point implementation of the *vhalf* algorithm in the context of computing radix-2 FFTs; all intermediate performance figures that appear in this subsection assume 16-bit, fixed-point, complex arithmetic and give performance numbers for FFT sizes that are assumed to be powers of 2 and range between 256 and 8192. For all the figures in this section it was also assumed that there were 32MB of memory divided into 16 banks with no subbanks.

5.5.1 Performance per Stage of the Fixed-point Vhalf Implementation

Figure 10 shows the MOP/s rate for each stage of the optimized *vhalf* FFT algorithm. Like the floating-point version, this implementation includes the *new*

vhalfup and **vhalfdn** instructions, the auto-incrementing, software pipelining, and code scheduling. This fixed-point version also uses indexed loads in its vhalf stages to load the roots of unity into the vector registers with the correct pattern, but in this case there are 7 such stages instead of 6, since the MVL is 128 instead of 64; like the floating-point version, the final output points for the fixed-point version are bit reverse rearranged. As before, the double circles indicate the stages in which VL = MVL = 128, and the open circles indicate the stages in which VL = 16, the number of virtual lanes.

Although it is not shown in Figure 10 note that 6.4 GOP/s is the maximum performance for the radix-2 complex 16-bit fixed-point FFT computation that might be ideally attainable on VIRAM, taking into account only the arithmetic operations[19]. As explained above, unlike the floating-point *vhalf* implementation, the fixed-point version does not utilize the multiply-add instruction. Thus in the fixed-point computation there are 10 arithmetic operations per basic computation, all of which have the same 6.4 GOP/s as the maximum rate attainable. Consequently, no adjustment for this fixed-point mix of operations needs to be made to calculate the maximum attainable GOP/s rate on VIRAM as was done for the floating-point mix of operations.

Figure 10, which illustrates the fixed-point *vhalf* performance, is very similar to Figure 8, which illustrates the floating-point *vhalf* performance, except for the actual values of the GFLOPS/GOPS rates. Because of this similarity, many of the same observations that were made about the floating-point *vhalf* performance can be made for this fixed-point *vhalf* performance. In particular, for all the FFT sizes except the 256 point, the first stage is somewhat slower than the second because the program start-up overhead is included with the first stage only, and as long as the VL is larger or equal to MVL = 128, the performance is maintained at 3.67 GOP/s after the first stage. Just as in the floating-point 128-point curve, the performance curve for the fixed-point 256-point FFT is the exception because its second stage, having a vector length of $MVL/2 = 64$, does not meet this criteria. In fact it is the first stage of this 256-point FFT that has a VL = 128 = MVL, and were it not for the start up overhead being included, its first stage GOP/s rate would be close to the 3.67 GOP/s as well.

Notwithstanding their similarities, the floating-point and fixed-point implementations differ greatly in their hardware untilization for these earlier *naïve* stages, which are all the stages whose VL is greater than or equal to MVL. For the floating-point version, the *naïve* stages had a utilization of 90%[20], but for the fixed-point version the utilization in the same earlier stages is a mere 57%[21]. This is due to architectural idiosyncrasies that will be discussed further in Section 6.

Recall that the first stage in which the in-register transposes occur is the stage in which VL = 64 = MVL/2. For each of the curves in Figure 10 we see a steep dip in the performance of this first stage because the stage includes the overhead for switching from the *naïve* algorithm to the *vhalf* algorithm, which incorporates the *vhalf* algorithm setup for the roots of unity and the bit reversal. The stage immediately following this first vhalf stage recovers, and for all the curves, the next five stages maintain a performance of 2.1 GOP/s, which indicates a 33%[22] utilization during these vhalf stages. Recall that the floating-point version had a 60%[23] utilization for the same stages. This under-utilization for the fixed-point version will also be discussed in more detail in Section 6.

Analogous to the floating-point performance, for all the fixed-point curves in Figure 10, the GOP/s rate for the last stage shows a steep degradation. As before, this is the effect of the bit reversal that is being done at the end of the last stage. Specifically, the indexed store of the contents of each of the four vector registers that hold the results is slowing the memory functional unit pipeline, and therefore the arithmetic functional unit pipeline, by at least a factor of four instead of two as it is for the floating-point version. If the memory unit experiences any bank conflicts from the random accesses into memory, then, as with the floating-point, these conflicts will stall the memory

[19] 6.4 GOP/s = 16 virtual lanes/cycle * 2 integer functional units/virtual lane * 1 integer operation/functional unit * 200 Mcycles/second

[20] 1.8 GFLOPS/2.0 GFLOPS * 100 = 90%

[21] 3.67 GOPS/6.4 GOPS * 100 = 57%

[22] 2.1 GOPS/6.4 GOPS * 100 = 33%

[23] 1.8 GFLOPS/2 GFLOPS * 100 = 60%

functional unit even more, which in turn will stall the arithmetic functional units for the same number of cycles as well. Since the data width is smaller for the fixed-point implementation, this means that more points fit into one bank, and therefore there is a higher liklihood of having memory bank conflicts for the fixed-point narrower data widths of 16-bits than with the floating-point 32-bit wide data.

Figure 10 shows the MOP/s rate for the last stage of the 256-point FFT to be 502, of the 512-point FFT to be 670, and of the 1024-point FFT to be 805. Similar to Figure 8 for the floating-point data, this indicates that the MOP/s rate for the last stage for the fixed-point data is getting better as the number of points in the FFT increases.

As is the case for the floating-point version, we can conclude that the differences we observe in the MOP/s rate between the various last stages are attributable to the amortization of the memory latency for the very last store as well as memory bank conflicts. As a matter of fact, for the fixed-point narrower data it is even more likely than with the wider floating-point data that the addresses generated by the indexed stores for the smaller FFT sizes are in the same bank since more points fit into one row of the memory bank. Consequently the smaller FFTs experience more memory bank conflicts during the indexed stores than the larger ones do. For these reasons the last stage of the 1024-point FFT in Figure 10 completes with a better MOP/s rate than the last stage of all the smaller sized ones.

Figure 10 verifies that for the fixed-point *vhalf* aglorithm, the total time is also not dominated by the time spent in the later stages of the FFT where the vector length falls below MVL. However the percentages of time and work spent in these later stages must be adjusted appropriately for the fixed-point stages, since the *vhalf* algorithm is utilized for the last 7 stages instead of the last 6 as in the floating-point version. For the fixed-point version illustrated in Figure 10, the time spent in the last 7 stages drops from the 96% that was seen in the *naïve* floating-point algorithm to 82% with the *vhalf* fixed-point algorithm, which includes the bit reversal slowdown, and which comes closer to the 70% of total work that the last 7 stages represent in a 10 stage, 1024-point FFT. As with the floating-point case, the in-register transposes have solved the short vector length problem exhibited by the *naïve* algorithm since we do not observe the same degradation in performance that we saw with the *naïve* algorithm when the vector lengths become smaller than MVL.

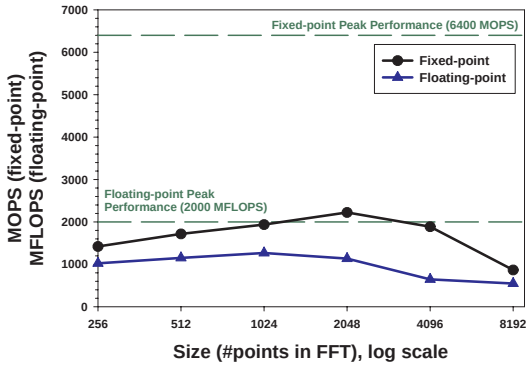


Figure 11: **Comparison of the performance in MFLOP/s for the *vhalf* floating-point, single precision, 32-bit, N-point FFT implementation with the performance in MOP/s for the *vhalf* fixed-point, 16-bit, N-point FFT implementation on VIRAM for N = 128, 256, 512, 1024, 2048, 4096, and 8192. The 6.4 GOP/s line shows the maximum performance for the 16-bit fixed-point FFT computation and the 2 GFLOP/s line shows the maximum performance for the 32-bit, floating-point FFT computation that might be ideally attainable on VIRAM, taking into account only the arithmetic operations. A $\log_2$ scale is used for the x-axis. Memory: 32MB, 16 Banks, No Subbanks.**

5.5.2 Performance Comparison of the Fixed-point and Floating-point Vhalf Implementations

However, since the floating-point implementation has already proved that the new vhalf instructions remedied the short vector length problem exhibited in the *naïve* algorithm, that was not the objective of converting the floating-point *vhalf* version into a fixed-point version. Our real objective, as stated at the beginning of this section, was to leverage VIRAM's increased integer performance relative to its floating-point performance to achieve a commensurate increase in the performance of the FFT. In particular we expected that the fixed-point implementa-

tion MOP/s rate might approach or exceed twice the MFLOP/s rate of the floating-point implementation since VIRAM's peak fixed-point performance for the FFT is 6.4 GOP/s compared to 2.0 GFLOP/s for the floating-point mix of operations, which is a ratio of 3.2 GOPS to 1 GFLOPS.

To see how we did, Figure 11 compares the overall MFLOP/MOP rates for each size FFT for the two implementations: the fixed-point algorithm and the floating-point algorithm. Both implementations do the bit reversing and the auto-incrementing. The floating-point implementation in this figure is the same as the one in Figure 8 and the fixed-point implementation in this figure is the same as the one in Figure 10. As in previous figures, the 6.4 GOP/s line shows the maximum performance for the 16-bit fixed-point FFT computation and the 2 GFLOP/s line shows the maximum performance for the 32-bit floating-point FFT computation that might be ideally attainable on VIRAM, taking into account only the arithmetic operations. A $\log_2$ scale is used for the x-axis and the memory configuration is assumed to be 32MB of DRAM with 16 Banks and No Subbanks.

In Figure 11 we observe that for all the FFT sizes, the 16-bit fixed-point MOP/s rate is approximately 1.5 faster than the corresponding 32-bit floating-point MFLOP/s rate. Although this is a healthy improvement, it is less than half of the potential 3.2 speed up. In addition, similar to the floating-point case, we observe in Figure 11 that as the FFT size increases the fixed-point implementation yields a higher MOP/s rate than that of the previous smaller size FFT until the 2048-point FFT, after which it begins to degrade for the 4096-point FFT and then seriously drops for the 8192-point FFT. The large drop in performance starting at an FFT size of 8192 for the fixed-point *vhalf* algorithm is due to memory bank conflicts. We will discuss this phenomenon and explain its performance impact later in Section 6.

Furthermore, the 2048-point GOP/s rate of 2.22, which is the highest rate in the figure, is achieving only a 35% utilization of the VIRAM full potential yet it still doubles the 1.13 GFLOP/s rate of the 2048-point 32-bit floating-point implementation. As is the case for the 1024-point floating-point FFT, the exceptional performance of the 2048-point 16-bit fixed-point FFT is due to architectural idiosyncrasies, which will also be discussed more fully below in Section 6.

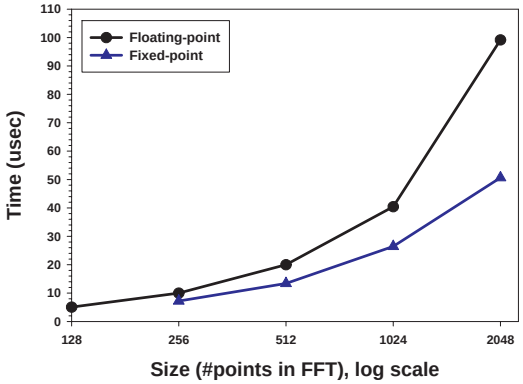


Figure 12: **Comparison of the performance in microseconds between the *vhalf* floating-point, single precision, 32-bit, N-point FFT implementation and the *vhalf* fixed-point, 16-bit, N-point FFT implementation on VIRAM for N = 256, 512, 1024, and 2048. A $\log_2$ scale is used for the x-axis. Memory: 32MB, 16 Banks, No Subbanks.**

Although the running times of the fixed-point implementation did not halve the running times of the floating-point implementation for all FFT sizes, in Figure 12 we see that for the 2048-point FFT, it certainly did. This figure compares the running times in microseconds of the same fixed-point and floating-point implementations used for Figure 11 for FFT sizes from 256 points through 2048 points. Both Figure 11 and Figure 12 confirm that even though the fixed-point implementation had a lower utilization than the floating-point implementation, the fixed-point implementation is still faster than the floating-point implementation for all FFT sizes. How to improve these times even further and why the fixed-point implementation utilization is lower than the floating-point implementation utilization are questions that will also be discussed in Section 6.

5.6 Error Analysis For The Fixed-point Results

The FFT algorithm is designed to work with continuous data (*i.e.*, the set of real numbers), which are typically handled using floating-point computations.

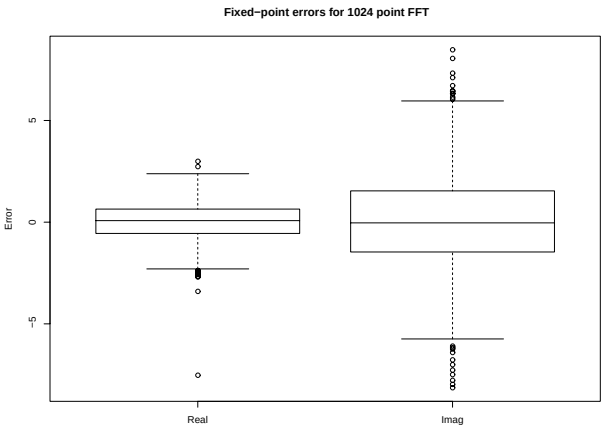


Figure 13: **Side-by-side box plots showing the error distribtions for the fixed point, 1024-point FFT. The left box shows the errors from the real components of the FFT results, and the right box shows the errors from the imaginary components of the FFT results.**

However, as we have seen above, it is possible to obtain a respectable performance improvement in the FFT by carrying out its computations using fixed-point arithmetic. But, by performing the computations using fixed-point, we run the risk of introducing a significant amount of error into the computation. As described above, the result of a single execution of the fixed-point version of the FFT's basic computation loses one bit of accuracy compared to its inputs; as a result, after running the entire FFT algorithm, each input loses one bit of precision per stage.

In this Error Analysis Section, we attempt to quantify the impact of this loss of intermediate precision on the quality of the final fixed-point output of the FFT. To form a basis for comparison, we took the same set of 16-bit integer-valued inputs used for the fixed-point FFT experiments and ran them through one of our floating-point FFT implementations. We then divided each of the resultant outputs by $2^{log_2 N}$ in order to rescale these floating-point output values to match the range of the fixed-point FFT algorithm's output. The resulting values represent the "correct" answer, that is the most precise result possible given the limitations of the floating-point representation.

To estimate the errors introduced by the fixed-point algorithm, we took each output value and subtracted the corresponding "correct" value obtained via the floating-point code. Figure 14 summarizes some of the statistical properties of these estimated errors for several FFT sizes in table form. Notice that in all cases the mean error is approximately zero, and that the standard deviations are small. We computed 95% confidence intervals for the mean errors, and in all cases 0 was contained in the interval, indicating that it is statistically likely that the true error is zero.

Figure 13 shows the error distribution for the 1024-point FFT in a more graphical manner. The figure shows side-by-side box-and-whisker plots for the errors in both the real and imaginary components of the FFT results. Recall that the box in a box-and-whisker plot includes the data from the lower quartile to the upper quartile (with the horizontal line in the box being the median). The distance between the whiskers is four times the distance between the upper and lower quartiles, and thus the whiskers give an indication of the overall spread of the data. The small circles indicate points that are statistically outliers.

Notice in the figure that both the real and imaginary errors are centered at zero and have symmetric distributions. Interestingly, the imaginary values have greater standard deviation and correspondingly more spread. We have been unable to find a satisfactory explanation for this pheonomenon. The one noticeable outlier in the real case corresponds to y_0, the first FFT output point. The first output point of every FFT corresponds to the vertical distance the entire curve is shifted above (or below) the x-axis. Specifically, if one draws a horizontal line whose equation is $y = y_0$, then all the sine and cosine curves associated with a particular FFT would have this line as their center line. Since we used only positive 16-bit integers for the fixed-point FFT input numbers for all our experiments, y_0 will be a large number. Graphically this means the center line is located a large distance above the x-axis. With such a large number, it is understandable that the error as we calculated it would also be larger than the other errors.

Finally, we hypothesize that the error distribution is statistically normal. Figure 15 shows two normal probability plots that graph the quantiles of the 1024-point real and imaginary error data against the quantiles of a standard normal distribution. The fact that these graphs show straight-line behavior indicates that the errors are most likely normally distributed.

FFT Real Component					
FFT Size	**Mean**	**Median**	**Min**	**Max**	**Std.Dev.**
1024	0.0117	0.0758	-7.5330	2.9930	0.9991
512	0.0039	0.0483	-6.9470	2.2510	0.9870
256	-0.0078	0.0184	-4.6720	2.8250	0.9986

FFT Imaginary Component					
FFT Size	**Mean**	**Median**	**Min**	**Max**	**Std.Dev.**
1024	0.0215	-0.0346	-8.1450	8.4810	2.2950
512	-0.0039	0.1078	-8.5540	7.6060	2.3215
256	-0.0391	-0.0319	-6.8770	5.5980	2.2063

Figure 14: **The mean, median, minimum, maximum, and standard deviation for the real and imaginary components of the estimated errors for each fixed-point, 16-bit, N-point FFT where N = 256, 512, and 1024.**

The imaginary values show a similar behavior as the reals but with slightly heavier tails (as suggested by the larger number of outliers in the box plot).

From all of this data we can conclude that the error behavior of the fixed-point algorithm is in fact quite well-behaved. The errors are normally distributed and centered around zero (and thus have a symmetric distribution). Furthermore, they have a small standard deviation relative to the absolute size of the actual result values. DSP experts who have inspected our fixed-point output values say this is the best accuracy that can be achieved when using fixed-point arithemetic to compute FFTs. Therefore we are satisfied that the fixed-point results that we obtained from our fixed-point FFT implementation are at least as accurate as those obtained by current fixed-point DSPs.

6 Architectural Analysis of the Performance Results

Since the best hardware utilizations achieved for the fixed-point *vhalf* implementation was 35% and for the floating-point *vhalf* implementation was 63%, we now investigate where the cycles not being utilized are going and why both the fixed-point and floating-point implementations do not perform closer to their respective attainable peak performance on the VIRAM architecture.

In the VIRAM architecture, there are three obvious candidates for the cause of under utilization of the hardware: indexed memory accesses, memory bank conflicts, and idle arithmetic functional units. In this section we will investigate all three of these possibilities in an attempt to identify the architectural causes of the under-utilization of both the fixed-point and floating-point implementations. In doing so we will also be able to explain the behaviors that have been observed but not fully discussed in previous sections.

As was discussed in Section 2, memory functional unit stalls cause the arithmetic functional unit pipelines to also stall because the pipelines of these functional units are not decoupled. As a result, computational cycles are not being fully utilized. Most memory functional unit stalls occur as a result of either memory bank conflicts or non-unit-stride memory accesses. A memory bank conflict means that the memory unit is attempting to access a memory bank that is already busy satisfying another memory request. The memory functional unit must then wait until the bank is free to handle its request, which happens after the bank has satisfied the first request. During the time the memory unit is waiting for the bank to become free, the arithmetic functional unit pipeline also stalls. We will investigate the impact of the non-unit-stride memory accesses and the impact of the memory bank conflicts in turn in the following two sections.

6.1 Analysis of the Indexed Memory Accesses

Recall that indexed memory accesses are non-unit-stride accesses that take twice as long as unit-stride

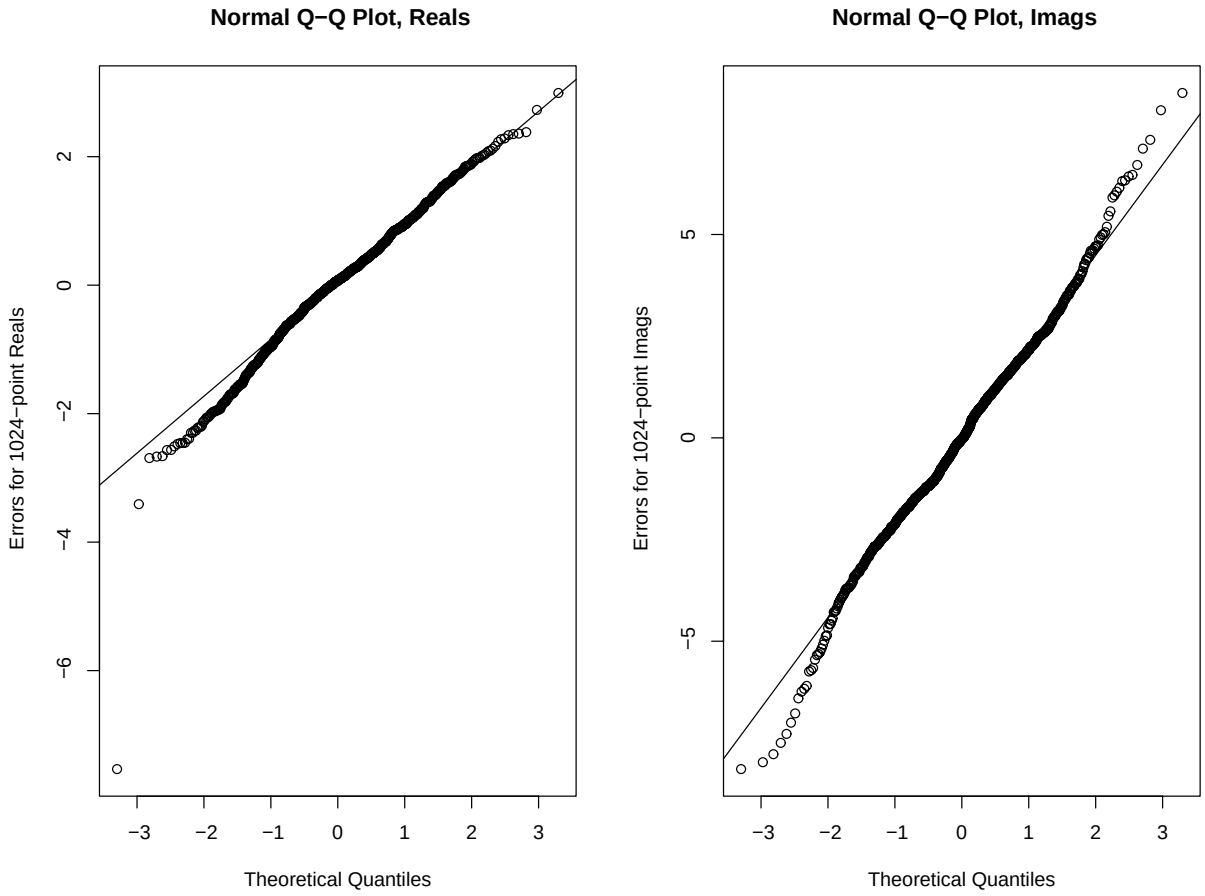


Figure 15: **Normal probability plots of the real and imaginary error components for the fixed-point 1024-point FFT.**

accesses for 32-bit data and four times as long as unit-stride accesses for 16-bit data. This means that during an indexed memory access the memory unit, and therefore the arithmetic functional units, stall every other cycle for 32-bit data and three out of every four cycles for 16-bit data. Since both the fixed-point and floating-point *vhalf* implementations use indexed loads in the vhalf stages to set up a vector register with the proper pattern of the roots of unity as well as indexed stores at the end of the last vhalf stage to do the bit reversal rearrangement, we first investigate the impact that these two indexed memory accesses are having on the performance of both the fixed-point and floating-point implementations.

Figure 17 compares the performance in MOP/s of four versions of the *vhalf* 16-bit, fixed-point implementation for N $= 256, 512, 1024,$ 2048, 4096 and 8192. Similarly Figure 16 compares the performance in MFLOP/s of four versions of the 32-bit, floating-point *vhalf* implementation for N $= 128, 256, 512,$ $1024,$ 2048, 4096, and 8192. Both experiments assumed a 32MB memory configured with 16 banks and no subbanks and both figures use a $\log_2$ scale for the FFT sizes on the x-axis. The four versions of the *vhalf* implementation are the same for both figures. The first version is the original implementation that contains both the indexed loads of the roots of unity and the bit reversing indexed stores. In the second version the indexed loads have been replaced with unit-stride loads but the indexed stores are still present. The third version mirrors the second version by keeping the indexed loads but replacing the indexed stores with unit-stride stores. Finally the fourth version replaces both the indexed loads and the indexed stores with unit-stride versions of each. Replacing the indexed accesses with unit-stride ac-

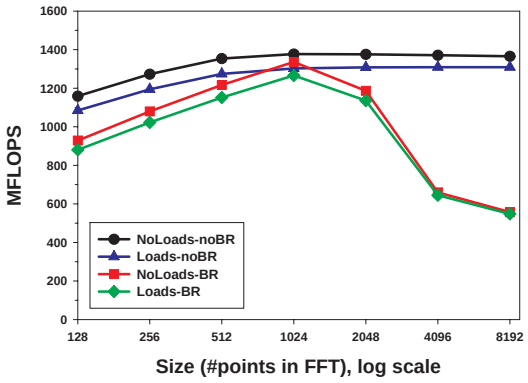


Figure 16: **Comparison of the performance in MFLOP/s of four different versions of the *vhalf* floating-point, single precision, 32-bit, N-point FFT implementation for N = 128, 256, 512, 1024, 2048, 4096, and 8192. The Loads-BR version is the original implementation with the indexed loads and stores. The NoLoads-BR version replaces the indexed loads with unit-stride loads but keeps the indexed stores. The Loads-noBR version keeps the indexed loads but replaces the indexed stores with unit-stride stores, and the NoLoads-noBR version replaces both the indexed loads and stores with unit-stride loads and stores. A $\log_2$ scale is used for the FFT sizes on the x-axis. Memory: 32MB, 16 Banks, No Subbanks.**

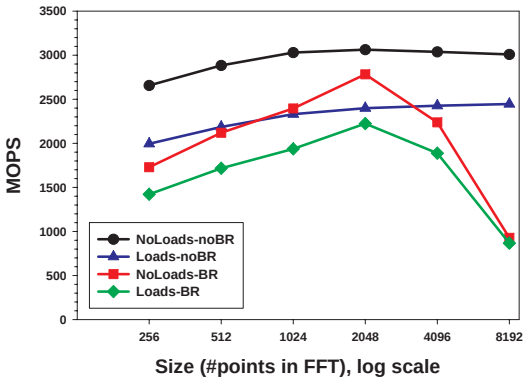


Figure 17: **Comparison of the performance in MOP/s of four different versions of the *vhalf* fixed-point, 16-bit N-point FFT implementation for N = 256, 512, 1024, 2048, 4096, and 8192. The Loads-BR version is the original implementation with the indexed loads and stores. The NoLoads-BR version replaces the indexed loads with unit-stride loads but keeps the indexed stores. The Loads-noBR version keeps the indexed loads but replaces the indexed stores with unit-stride stores, and the noLoads-noBR version replaces both the indexed loads and stores with unit-stride loads and stores. A $\log_2$ scale is used for FFT sizes on the x-axis. Memory: 32MB, 16 Banks, No Subbanks.**

cesses allows us to see what the performance of our algorithms would be if the indexed accesses were as fast as the sequential, unit-stride accesses. We can then more precisely evaluate the current VIRAM implementation of the indexed accesses.

In those versions in which indexed accesses have been replaced by unit-stride accesses, the results are not computationally correct since the wrong roots of unity are being used in the basic computation and since there is no bit reversal being performed after the last stage. Since the purpose of this experiment was to assess the impact that the indexed memory accesses are having on the *vhalf* implementations, the fact that the results are incorrect has no relevance in this case.

6.1.1 The "Ideal Size" FFT

There are several points to be made about the data presented in Figures 16 and 17. First, notice that in Figure 16, all the floating-point curves peak for the 1024-point FFT, whereas in Figure 17, the fixed-point curves peak for the 2048-point FFT. This effect is due to the memory layout of the FFT data. With the 16-bank, 2048-bit-wide memory configuration we assumed, 1024 32-bit points (a total of 16*2048 bits) can be accessed in a sequential manner such that every bank is accessed only once (each bank access extracts a 2048-bit row containing 64 consecutive points). Similarly, 2048 16-bit points can be accessed sequentially without bank conflicts.

These facts explain the peaks in the FFT performance curves: for the floating-point FFT, which uses 32-bit data, the best performance is achieved when the FFT operates 1024 points (a size which we will denote the *ideal size*), since the computation accesses every bank in the memory system sequentially before returning to the first bank. For smaller FFTs, the computation uses only a fraction of the banks,

returning to the first bank more quickly and potentially causing stalls until that bank is ready to service a second request. For larger FFTs, multiple accesses are required to at least some of the banks, increasing the potential for memory conflicts and thus lowering performance. An analogous explanation applies to the fixed-point case, except there the "ideal size" for the FFT is 2048 points.

6.1.2 The Indexed Loads

Another key point to be made concerning the data in Figures 16 and 17 is in the effect of removing the indexed loads. If we temporarily ignore FFT sizes greater than the "ideal size", we see that the lines corresponding to the case with indexed loads removed (labeled "NoLoads" in the graphs) are a small but constant amount higher than the corresponding curves that include the loads. The fact that these differences are constant indicates that the overhead cost of the indexed memory load operations relative to unit-stride load operations is fixed, and the difference between the lines measures this cost.

The fact that this cost is fixed and rather low is in itself surprising. The explanation lies in the implementation details of our *vhalf* algorithm. Recall that the indexed loads are loading the real and imaginary components of the roots of unity in specific patterns into vector registers. For indexed accesses, one address is generated for each element being accessed and for each such address a separate memory request is made. Since the same root of unity is loaded into several consecutive element slots of one vector register, several consecutive individual accesses have identical addresses, are therefore clearly going to the same bank, and as a consequence cause memory bank conflicts.

However, for each vhalf stage but the last one, only 2 * MVL roots, which is enough to fill two vector registers, are loaded in this manner;[24] one vector register receives a set of real roots and the other receives a set of imaginary roots. Therefore the impact of these indexed loads is held constant and does not grow with the size of the FFT. The impact itself is small because only 2 * MVL roots of unity are being loaded via these indexed operations per group per stage.

[24]In the last vhalf stage, the roots of unity are loaded using unit-stride loads, since each butterfly has a VL=1.

6.1.3 The Indexed Stores

In contrast, if we compare the lines in either Figure 16 or Figure 17 that correspond to the cases that remove (noBR) and include (BR) the indexed store of the bit reversal, we see that these lines are not separated by a small or constant amount. Instead, the gulf between the lines starts out large for small FFTs, but gets much smaller as the FFT size approaches the ideal. This is due to the fact that for the smaller FFT sizes the points are not spread out among all the memory banks as they are for the larger FFT sizes, especially for the "ideal size" FFT, which uses all the banks. Consequently the indexed stores for the smaller FFTs incur more bank conflicts than the larger FFTs up to and including the "ideal size" FFT.

The sharp drop for the FFTs whose size is larger than the "ideal size" is due to the fact that the indexed stores are doing the final bit reversals. This means that there is one independent store for every point in the FFT. Clearly the number of indexed stores grows linearly with the FFT size. As the number of points in the FFT increases, so do the indexed stores and, for sizes larger than the "ideal size", their accompanying pipeline stalls. (The reasons for this special behavior for FFTs that have the "ideal size" and for the extreme degradation when the FFT size is larger than the "ideal size" will be discussed in more detail below.) Consequently the impact of these indexed stores is far more deleterious on the performance of our *vhalf* implementation than the indexed loads are, and the performance impact increases with increasing FFT size after the "ideal size".

6.1.4 Fixed-point Verses Floating-point Gaps

Next, in comparing Figures 16 and 17, notice that the gaps between the Loads/NoLoads lines and the BR/noBR lines are significantly larger in the fixed-point case than in the floating-point case. This discrepancy is due to the fact that the fixed-point algorithm operates with a **vpw** of 16, compared to the **vpw** of 32 for the floating-point case. With a 16-bit **vpw**, the 16-bit indexed accesses take not two (as is the case for the 32-bit accesses), but four times

longer than the unit-stride accesses. Thus the increased gaps for fixed-point are caused by the inherent slowdown that arises when indexed memory operations are used on narrower data on VIRAM.

6.1.5 Utilization In The Vhalf Stages

Before addressing the question concerning the deleterious effect of the indexed stores for FFTs larger in size than the "ideal size", while we are on the subject of the indexed loads in the vhalf stages, let us briefly digress. The purpose of this digression is to now explain why the utilization for each individual vhalf stage is relatively low for both the fixed-point implementation and the floating-point implementation.

Recall that we expected the utilization for each of these interim vhalf stages to be high since the interim stages do not load and store the intermediary values for the input points as the *naïve* algorithm does, so they contain far fewer memory accesses. Fewer memory accesses means fewer possible memory conflicts, which means fewer arithmetic functional unit stalls.

By now part of the answer should be clear. Because of their inherent slowdowns, the indexed loads of the roots of unity are stalling the arithmetic functional units, and therefore the MFLOP/MOP rates suffer and the utilization is poor. These effects are more pronounced in the vhalf stages for the fixed-point implementations (33% utilization) than for the floating-point implementations (60%) because of the factor of four over the factor of two slowdowns. Without the indexed loads in these vhalf stages the sustained utilization for each vhalf stage but the last increases to 49% for the fixed-point and 72% for the floating-point. Later in this section we will revisit and complete this discussion of why the vhalf stages do not have better hardware utilization even in the absence of the indexed loads.

6.1.6 Impact of the Indexed Accesses

What can be said about the indexed accesses at this point in our discussion is that the narrower the data width, the more impact indexed accesses will have on the performance of the FFT on VIRAM. For floating-point 32-bit wide data the *vhalf* implementation utilization for its "ideal size" 1024-point FFT without the indexed accesses is 69%. With the indexed accesses the utilization drops to 63%, a delta of only 6%. On the other hand, for the fixed-point 16-bit wide data, the corresponding utilizations go from 47% to 30%, a delta of 17%, which is three times the delta of the floating-point implementation. Furthermore, as the FFT size increases, the deltas for both the fixed-point and the floating-point will also increase since the price paid for the indexed stores grows with the FFT sizes larger than the "ideal size".

6.2 Analysis of Memory Bank Conflicts

Let us now return to the question: why do the indexed stores have such a deleterious effect on the performance of both the fixed-point and floating-point implementations for FFTs larger than 2048-point FFTs? Recall that we saw this performance drop of the 4096- and 8192-point FFTs in Figure 9 for the floating-point *vhalf* implementation and in Figure 11 for the fixed-point *vhalf* implementation in the previous two sections and we deferred our discussion of these observations to this section.

We can now account for this behavior. Above we established that the impact of the indexed stores grows with the size of the FFT. The larger the FFT size, the more indexed stores. The more indexed stores, the more stalls. But these stalls alone do not account for the extreme drop in the MOP/s & MFLOP/s rate that we observe in our figures. For both the fixed-point and the floating-point implementations, the 8192-point size is a multiple of their respective "ideal size"s. Specifically, 1024 goes into 8192 eight times while 2048 goes into 8192 four times. Thus for the first *naïve* stage where the butterfly starts with points x_0 and $x_{N/2}$, there will be memory bank conflicts in accessing the top and bottom halves of the butterfly because both access the same bank.

Furthermore, as long as the distance between the top point and the bottom point of the basic computation is a multiple of 1024 for floating-point and 2048 for fixed-point, these same types of memory conflicts exist and consequently they, along with the incurred stalls of the larger FFT sizes, impair the 8192-point FFT performance of both implementations. One solution to this problem is to make the number of banks

in memory not a power of two, but this is not really an option for VIRAM since it introduces too much complexity which would eat up area and power, and thus violate our design objectives. Another is to either increase the number of memory banks, which increases the total size of memory, or introduce subbanks into the memory configuration, which does not increase the total size of memory.

6.2.1 Subbanks

Subbanks within the same bank allow multiple accesses to the same bank to be pipelined. Although the bank has a data bus that handles only one request at a time, with subbanks, more than one row within the bank can be accessed and made ready to go as soon as the bank's data bus is free. The number of subbanks within a bank determines the number of rows that can be active at any one point within the same bank since the subbanks within the same bank have independent accessing to their own rows much like banks have independent accessing to their own rows (and subbanks). Without the existence of subbanks, a second access to the same bank has to wait for the access ahead of it to complete, which takes an entire memory cycle time, and no overlapping of the row fetching can be done. The more banks a memory system has the more independent accesses it can handle without producing a memory bank conflict. The more subbanks a bank has the more pipelined accesses it can handle without producing a subbank conflict.

6.2.2 Varying the Memory Configuration

Figure 18 and Figure 19 answer the question of what happens to the MFLOP/s & MOP/s performance of both the floating-point, 32-bit, and the fixed-point, 16-bit, N-point FFT, *vhalf* algorithm when we vary the memory configuration. In both figures a $\log_2$ scale was used for the FFT sizes on the x-axis and the *vhalf* implementations used to generate the data in both of these figures contains the indexed loads of the roots of unity in the vhalf stages and the bit reversing indexed stores after the last vhalf stage. Five different memory configurations were used as follows: 1) 32MB, 16 Banks, No Subbanks, 2) 32MB, 16 Banks, 4 Subbanks, 3) 32MB, 8 Banks, 8 Subbanks, 4) 16MB, 8 Banks, No Subbanks, 5) 16MB, 8 Banks, 4 Subbanks. Therefore each figure has five performance lines. The 16MB memory configurations were included here because it was decided that the VIRAM prototype will have 16MB instead of the original 32MB. As stated earlier, this change of memory design occurred after much of the work for these experiments had been done. Consequently we wanted to see the impact that the smaller memory size would have on the performance of the *vhalf* algorithm.

Floating-point Memory Configurations In Figure 18 we see that all five curves coincide and are increasing in MFLOP/s rate for FFT sizes 512 and below. This indicates that for these size floating-point FFTs there are no significant memory bank conflicts. In this figure, the curve for the 16MB, 8Bank, no subbank configuration takes a sudden dive down after the 512-point data point. This is because the 16MB memory size with 8 banks and no subbanks, allows 512 32-bit points to be accessed without revisiting the same bank. Since this 16MB configuration is exactly half of the 32MB, 16 bank, no subbank configuration we saw in Figure 16, it makes sense that half the number of accesses can occur without revisiting the same bank for the 16MB memory as for the 32MB memory with a very similar no subbank configuration.

In addition, notice that the 32MB, 16 bank, no subbank curve practically coincides with the 16MB, 8 bank, 4 subbank curve in Figure 18, although the 16MB curve ends up yeilding a slightly higher MFLOP/s rate for the 8192-point FFT than does the 32MB curve. Both of these curves peak at the 1024-point data point and then take a dive. Again, this is because 1024 points can be accessed from these memory configurations without revisiting the same bank or subbank.

The final two 32MB curves, one with 16 banks and 4 subbanks, the other with 8 banks and 8 subbanks also coincide. Furthermore both of these curves yeild a much higher MFLOP/s rate for the 8192-point FFT since these memory configurations ameliorate the memory conflict problems experienced when there are fewer banks or no subbanks.

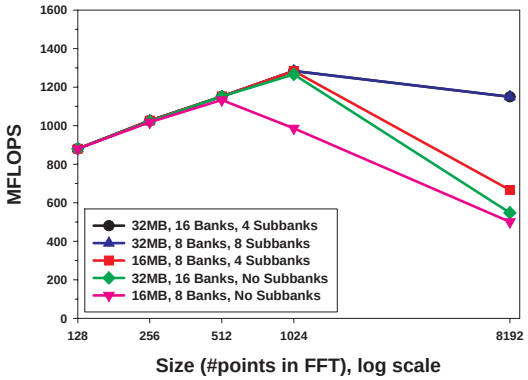


Figure 18: **Comparison of the performance in MFLOP/s of the *vhalf* floating-point, 32-bit N-point FFT implementation for N = 128, 256, 512, 1024, and 8192 with the following five memory configurations: 1) 32MB, 16 Banks, No Subbanks, 2) 32MB, 16 Banks, 4 Subbanks, 3) 32MB, 8 Banks, 8 Subbanks, 4) 16MB, 8 Banks, No Subbanks, 5) 16MB, 8 Banks, 4 Subbanks. A $\log_2$ scale is used for the FFT sizes on the x-axis.**

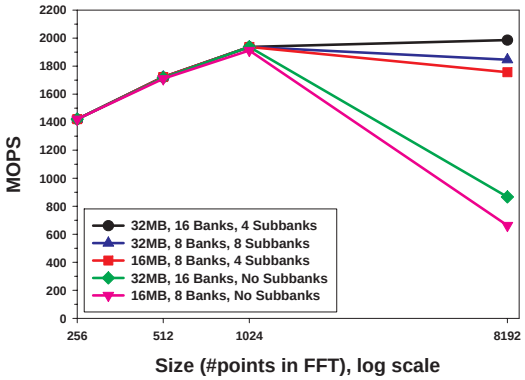


Figure 19: **Comparison of the performance in MOP/s of the *vhalf* fixed-point, 16-bit N-point FFT implementation for N $= 256, 512, 1024,$ and 8192 with the following five memory configurations: 1) 32MB, 16 Banks, No Subbanks, 2) 32MB, 16 Banks, 4 Subbanks, 3) 32MB, 8 Banks, 8 Subbanks, 4) 16MB, 8 Banks, No Subbanks, 5) 16MB, 8 Banks, 4 Subbanks. A $\log_2$ scale is used for the FFT sizes on the x-axis.**

Fixed-point Memory Configurations The fixed-point curves in Figure 19 behave in a very similar fashion as their floating-point counterparts in Figure 18 except for the 16MB, 8 bank, 4 subbank configuration curve. Assuming this configuration and 16-bit data points, 4096 points can be accessed without revisiting the same subbank. Therefore the MOP/s rate for the 8192-point FFT falls only slightly below the ”ideal size” MOP/s rate for this configuration. This is because for the 8192-point FFT, the memory bank conflicts only affect the one and only butterfly group in the very first stage of the FFT calculation where the top and bottom points for every basic calculation are exactly 4096 points apart. However, after this first stage, the gap between the top and bottom points continuously halves, so for the second stage the gap is only 2048. Thus after its first stage, the 8192-point FFT with the 16MB, 8 bank, 4 subbank memory configuration does not have to absorb such an ill-fated memory conflict pattern again.

6.2.3 The Impact of Memory Size and Configuration

The conclusion we can draw from Figure 18 and Figure 19 is quite clear. Performance will drop significantly for larger FFT sizes due to memory bank conflicts. This deleterious effect can be ameliorated best by adding subbanks. Increasing the memory size without utilizing subbank configurations might temporarily help since it increases the number of independent banks, but for a more robust solution against memory conflict thrashing, the more subbanks per bank, the better.

6.3 Analysis of Bottlenecks and Poor Hardware Utilization

Although additional banks and subbanks will considerably ameliorate the memory bank conflicts, we cannot assume that this is all that is necessary to increase the hardware utilization. As we saw in Figures 16 and 17, eliminating all the indexed accesses from both the fixed-point and floating-point *vhalf* implementations achieved a higher utilization of the hardware, but this utilization was still much lower

than peak for both. For the floating-point implementation, after replacing all the indexed accesses with unit-stride accesses, the highest overall MFLOP/s rate attained regardless of the memory configuration was 72%. Not surprisingly, this is the same utilization that was achieved by each of the interim vhalf stages under the same conditions. For the fixed-point utilization the story is the same, but the utilization is a much lower 49%.

Recall that in the interim vhalf stages, input points are not loaded and result points are not stored since the elements are transposed within the vector registers eliminating the need to access memory each stage. Therefore for both the fixed-point and floating-point implementations, we expect the vhalf interim stage MOP/s & MFLOP/s rates to be close to the highest possible. A 72% utilization for the floating-point implementation and a 49% utilization for the fixed-point implementation for these vhalf interim stages indicates that something is happening with these implementations that is preventing even these stages from attaining closer to peak MOP/s & MFLOP/s rates. It is the goal of this subsection to clarify what exactly is happening.

To do so we must ascertain whether or not the arithmetic functional units are being kept fully busy. Furthermore, if the functional units are not being kept fully utilized we must ascertain where in the implementation this is the case and why they are not. On the other hand, if the functional units are being kept fully utilized we must then ascertain why the MOP/s & MFLOP/s rates are not higher. To do this assessment, we inspect carefully selected pipeline traces from our simulator for the fixed-point and floating-point *vhalf* implementations. Specifically, to generate the fixed-point and floating-point traces we use the version of these implementations that replace the indexed loads and stores with unit-stride loads and stores. We use these versions of the implementations to generate the traces because, with all the effects of the indexed accesses removed, these versions have the best performance and utilization of the VIRAM hardware that we have been able to achieve so far. Why do they not do better?

Furthermore, the traces are generated using the 1024-point FFT size for the floating-point version and the 2048-point FFT size for the fixed-point version because these two cases experience the fewest memory bank conflicts. Eliminating as many of the memory effects as possible from the traces we are inspecting will allow us to ascertain what else is effecting the performance and utilization apart from the memory system.

We will start at the beginning of the algorithm and inspect each section of the algorithm as well as the transitions between the sections in the traces. Recall that the first section of both implementations starts with the stages whose VL is greater than or equal to MVL and uses the *naïve* algorithm to vectorize these stages. The next section of both implementations starts when VL = MVL/2 and uses the vhalf method to vectorize the remaining stages. The *vhalf* algorithm divides all the points into groups containing 2*MVL points and then processes each group through all the vhalf stages, thus completing all the remaining FFT stage calculations for that group before starting with the next group. When all the groups have been processed by the vhalf section, the FFT implementation is done.

6.3.1 The Naïve Stages of the Vhalf Algorithm

In the first section of the algorithm, during the *naïve* algorithm stages, for both the fixed-point and the floating-point implementations, the potential bottleneck is the memory functional unit. It is being utilized 100% since for each group of MVL elements there are four unit-stride loads and four unit-stride stores. In the floating-point implementation the floating-point functional unit is idle for only 8 cycles per 2*MVL elements which is a decent use of resources and which indicates that it is not stalled waiting for the memory unit to feed operands to it.

Recall that even in the *naïve* algorithm we saw a 1.8 GFLOP/s rate for those stages whose VL was greater or equal to MVL. Since the peak is 2.0 GFLOP/s, the utilization for these stages is 90% for the floating-point implementation. In this case, resources and demand for them are decently balanced. Since the floating-point functional unit is idle for only 8 cycles per 2 * MVL points despite the memory unit being fully utilized, the memory unit is not a bottleneck.

In the same first section the trace for the fixed-point implementation exhibits more idle cycles for its two arithmetic functional units, FU1 and FU2, in

these *naïve* algorithm stages than the floating-point did for its one floating-point functional unit. Specifically FU1 is idle for 30 cycles and FU2 is idle for 10 non-overlapping cycles. Since there are twice as many arithmetic functional units and since the potential bottleneck is the memory unit, this makes complete sense.

In these *naïve* stages of the fixed-point implementation, the memory unit is limiting the amount of work coming in, and there are twice as many resources to do the work once it comes in. Thus this implementation gets about 57% utilization and operates around 3.7 GOP/s, where the peak is 6.4 GOP/s. This under-utilization for the fixed-point implementation is easy to understand. The execution can only go as fast as one memory unit, but the peak performance, and therefore the utilization, is based upon having two functional units going 100% of the time. Unless a fixed-point or integer application is very compute intensive, meaning it has a computation to I/O ratio larger than 2:1, which the FFT, whose ratio is approximately 1:1, does not have, high utilization is difficult to achieve for this resource configuration and is what we are experiencing for the *vhalf* fixed-point implementation. Thus for this fixed-point case the memory unit is definitely a bottleneck in the *naïve* stages of the algorithm.

6.3.2 The Transition from the Naïve Stages to the Vhalf Stages

During the time that the algorithm is transitioning from the *naïve* section to the vhalf section, in both traces the memory unit is 100% busy and there are idle cycles in the arithmetic functional units. Specifically, for the floating-point trace the floating-point functional unit has 26 idle cycles and for the fixed-point trace, FU1 is idle 51 cycles while FU2 is idle for 54 cycles, only some of which overlap. At this point the memory unit is the scarce resource for both traces, but the situation is exacerbated for the fixed-point implementation because the one memory unit must keep twice the number of functional units busy. Thus in this transition section of the algorithm, the memory unit is the bottleneck in both traces.

Specifically, in the vhalf section, at the beginning of the execution of the first vhalf stage, six unit-stride loads are done in both traces. Therefore we see the memory unit once again being the bottleneck for both traces during this first vhalf stage. However, since all the interim vhalf stages, which are all the vhalf stages but the first and the last, have only two unit-stride loads, which are supposed to access the roots of unity, we do not expect the memory unit to be the bottleneck for these interim stages in our traces. Recall that these interim vhalf stages access the correct roots of unity by doing indexed loads, but for analysis purposes, our traces are from versions that have replaced the indexed loads with the incorrect but non-stalling, more efficient unit-stride loads.

Thus during transition from the *naïve* section to the vhalf section and in the first vhalf stage, the algorithm must store the output values from the last *naïve* stage and then set up for the vhalf stages by doing overhead loads in addition to the loads that access the next set of input FFT points. It is these additional loads that upset the balance that achieved the 90% untilization for the *naïve* stages in the floating-point trace and that exacerbate the fixed-point situation even further.

6.3.3 The Floating-point Vhalf Interim Stages

During the vhalf interim stages, the floating-point implementation keeps its one floating-point functional unit almost 100% busy. However, many of the vector instructions being executed are not part of the 8 being counted for the basic operation. As illustrated in Figure 7, two **`vmerges`** and four **`vhalfup/dn`** instructions per 2*MVL points per vhalf stage are necessary to accomplish the in-register element transposes. VIRAM carries out the execution of these overhead instructions on both of the arithmetic integer functional units, often in parallel.

However, since only one of these two functional units can execute floating-point operations, it is often the case that the floating-point operations making up the basic computation are blocked by overhead instructions executing on that one FP-capable functional unit. Thus, although this one FP-capable functional unit is utilized almost 100%of the time, the GFLOP/s rate that is achieved during each of these interim vhalf stages for the floating-point implementation is 1.2 which translates to a 60% utilization.

At this point in the floating-point implementation, just as we predicted, the bottleneck is not the mem-

ory unit starving the arithmetic functional unit. Instead, the floating-point arithmetic functional unit is 100% utilized but much of the work that it is doing is overhead work that does not contribute to a faster GFLOP/s rate. This is the price that must be paid to use the in-register transpose operations to ameliorate the ill effects of the shorter vector lengths experienced in the *naïve* algorithm.

6.3.4 The Fixed-point Vhalf Interim Stages

The situation during the vhalf interim stages for the fixed-point trace is slightly different than that of the floating-point trace just discussed because the operations making up the basic computation can execute on either of the arithmetic functional units in the fixed-point implementation. The fixed-point trace shows for each interim stage FU1 is idle only 2 cycles while FU2 is idle 17 cycles. Since the total work is being divided up between the two arithmetic functional units fairly evenly and since both functional units are being kept relatively busy, it would seem that most of the work is getting done in parallel and the GOP/s rate should be close to peak. Yet for these stages the GOP/s rate is only 2.1 and the utilization is a disappointing 33%.

There are two reasons that the GOP/s rate is only a third of peak for each interim vhalf stage for the fixed-point implementation. The first is the same as in the floating-point implementation above; overhead work is being done which does not contribute to a faster MOP/s rate. However, for the fixed-point implementaion, the situation is exacerbated by the additional vector instructions, such as the vector shifts, that must be used as part of the fixed-point basic computation in order to track the assumed binary point. These additional vector instructions are used each time the basic computation is performed, but they are not counted as one of the 10 fixed-point operations that compose the basic computation for the fixed-point calcuation when computing the peak rate since they are considered overhead instructions for doing the fixed-point arithmetic.

The second reason that the GOP/s rate is only a third of peak for each interim vhalf stage in the fixed-point trace is that there is not enough work to keep two arithmetic functional units fully occupied, so not all the resources are being utilized all of the time. It should be noted, however, that the fixed-point GOP/s rate of 2.1 is almost twice the 1.2 floating-point GFLOP/s rate. Consequently, having the two arithmetic functional units working in parallel despite the fact that they are not kept fully utilized definitely has a positive impact on performance.

6.3.5 The Last Vhalf Stage and the Transition Back to the First

For both implementations, the last vhalf stage does a store of the four vector registers, for the top and bottom reals and the top and bottom imaginaries, before giving control back to the first vhalf stage in order to reiterate the sequence of vhalf stages for the next group of 2 * MVL points. Recall that the stores in this stage should really be the bit reversing indexed stores. But for the purposes of this analysis, we are using the traces which are generated after replacing these indexed stores with unit stride stores so that we can separate what is going on as a result of the memory system from what is going on as a result of the vector processor architecture.

For both implementations, the stores in the last vhalf stage keep the memory unit fully utilized. When control returns to the first vhalf stage six more loads are issued, so once again the memory unit is the bottleneck for the first vhalf stage. No vhalf first stage basic operations can be executed until the memory unit is free to read in the new values for the next group of points, which can only happen after the results from the last vhalf stage for the previous group of points have been stored. In the floating-point trace, the floating-point functional unit is idle after returning back to the vhalf first stage for 14 cycles. In the fixed-point trace both FU1 and FU2 are each idle 26 cycles waiting for the memory unit to supply the next basic operations's operands. Once again, the performance effects of the memory unit bottleneck for the fixed-point trace are worse than for the floating-point trace because not one but two functional units are idle and they are idle for a longer period of time. After this first vhalf stage has completed, the vhalf interim stages are repeated, followed by the vhalf last stage once again.

6.3.6 Summary of the Analysis

From the above analysis of both the fixed-point and floating-point traces it is now clear when the functional units are being kept fully busy and when they are not. In addition, when the arithmetic functional units are not being fully utilized it is clear where in the algorithm this is happening and why this is the case. When the arithmetic functional units are being fully utilized we see why the MOP/s & MFLOP/s rates, and therefore the utilization percentage, are not higher. Specifically, we observe three different behaviors throughout the course of the *vhalf* algorithm. The first behavior only occurs for the floating-point implementation. The second and the third occur for both the floating-point and the fixed-point implementations.

The first behavior occurs in the *naïve* section of the algorithm only for the floating-point implementation, which has one floating point functional unit and one integer functional unit. For all the *naïve* stages in the trace, the one floating-point functional unit is kept busy while the memory unit is also 100% fully utilized. In this case the memory unit is not the bottleneck because the amount of work requiring both resources is well balanced. The memory unit, while operating at its fullest capacity of 100% utilization, supplies exactly the right amount of data to the floating-point functional unit to keep it busy and not under-utilized. But this balance is precarious, since as soon as more memory accesses are required, as they are when the algorithm transitions to the vhalf section, functional unit utilization significantly drops because the memory unit has become the bottleneck. This is the dip we saw for the first vhalf stage in Figure 8 for the floating-point implementation.

The second behavior occurs in the fixed-point trace during the *naïve* section of the algorithm, and in both traces when the algorthm transitions from the *naïve* section to the vhalf section and then from the last vhalf stage back to the first vhalf stage. At each of these places in the traces, the memory unit is the bottleneck, and the arithmetic functional units are idle for a significant amount of time. The impact of such a bottleneck for the fixed-point implementation is exacerbated by the fact that there are two functional units that become idle instead of one, and therefore there is a *double* impact on the GOP/s rate. This behavior now explains the poor GFLOP/s rate observed for the first and last vhalf stages for the floating-point implementation in Figure 8 as well as the poor GOP/s rate observed for these same stages for the fixed-point implementation in Figure 10.

The third behavior occurs for both traces during the vhalf interim stages. In each of these stages for both of the traces the functional units are doing overhead work that does not contribute to the improvement of the GFLOP/s or GOP/s rate. In the floating-point trace the floating-point functional unit is almost fully utilized and the integer functional unit is being used in parallel for part of the time during these stages. In the fixed-point trace, since there are two equivalent functional units, because of data dependencies neither is fully utilized despite the fact that there is additional overhead work to do for the fixed-point basic computation.

We now have an explanation of why the best utilization we can achieve even when we take out all the indexed accesses is 72% for the floating-point implementation and 49% for the fixed-point implementation. To achieve better utilization for both implementations, at the very least the one memory functional unit needs to be decoupled from the arithmetic functional unit pipeline. To attain an even better utilization for both implementations the memory bottlenecks would have to be eliminated. Two memory functional units might ameliorate the bottleneck, but unless both memory units were decoupled from the arithmetic functional unit pipeline, the stalls to the functional unit pipeline caused by each memory unit would no doubt exacerbate rather than ameliorate the situation. In addition, having two memory functional units would undoubtedly cause additional memory bank conflicts since there would be a higher probability of accesses being made to the same bank. Therefore adding another memory functional unit makes it all the more imperative to also have a healthy number of subbanks configured into the memory system to ameliorate some of these inevitable memory bank conflicts[25].

[25] Although the results are not reported in this paper, several experiments were run assuming two memory units per lane. The results corroborated the claim made here; two memory units without decoupling them from the arithmetic functional unit pipeline and with no subbanks in the memory configuration ex-

7 VIRAM vs. DSP Performance

In this section we will compare VIRAM's performance results with both fixed-point and floating-point DSP performance.

Both Figure 20 and Figure 21 show the running times in microseconds for various size FFTs for two implementations of the optimized *vhalf* algorithm. In both figures a $\log_2$ scale is used for the FFT sizes on the x-axis. The single difference in the implementations is the memory configuration. The first memory configuration is 16MB, 8 banks, and no subbanks (labeled "16MB"). The second memory configuration is 32MB, 16 banks, and no subbanks (labeled "32MB"). Recall that many of the experiments in this paper were performed before the final design decisions were made about the VIRAM chip. For these experiments a 32MB, 16 bank, no subbank configuration was assumed. At a later date it was decided that the VIRAM prototype would have a 16MB, 8 bank, no subbank configuration. For this reason we have included performance times for both memory configurations.

The *vhalf* implementation in both these figures utilizes the *new* **`vhalfup`** and **`vhalfdn`** instructions, the auto-increment feature, software pipelining, and code scheduling; the final output points are bit reverse rearranged. In Figure 20 the *vhalf* results are for single precision, floating-point (32-bit), complex, radix-2 FFTs. In Figure 21 the *vhalf* results are for fixed-point (16-bit), complex, radix-2 FFTs.

Also included in Figures 20 and 21 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 22). 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.

In Figure 20 we start to see the effects of fewer banks in the 1024-point FFT since the time on VIRAM for the 16MB memory implementation is 52 microseconds while it is 40 microseconds for the 32 MB implementation. In Figure 21 the times for both memory configurations is identical, and therefore there is only one line drawn for both memory configurations.

As Figures 20 and 21 indicate, VIRAM is competitive with the high-end, specialized DSPs for both the fixed-point and floating-point *vhalf* implementations. For example, for the 32 MB floating-point FFT, it outperforms the TigerSHARC by a factor of 1.73, the ADSP-21160 by a factor of 2.3, and the TMS320C6701 by an impressive factor of 3.1. VIRAM is also within a factor of 2.5 of the performance of two other high-end DSPs: the Wildstar runs at 1.6 times the performance of VIRAM, and the Pathfinder-2 is 2.5 times faster.

As seen in Figure 21 for the fixed-point FFT, VIRAM slightly betters the performance of the Pathfinder-1 by a factor of 1.06, but it outperforms the Carmel by a factor of 1.78, and the TMS320C6201 by an impressive factor of 4.3 for the 1024-point FFTs. For the 256-point FFTs VIRAM equals the performance of the TigarSHARC, and it outperforms the Pathfinder-1 by a factor of 1.125, the Carmel by a factor of 1.25, the PowerPC 604e by an incredible factor of 12.1, and the Pentium I by an even more impressive factor of 21! As a matter of fact, we could not find published numbers for any CPU/DSP that outperformed VIRAM's fixed-point time.

We believe that VIRAM's performance could match or exceed the floating-point performance of the Wildstar and the Pathfinder-2 and that it could outperform the rest, in both fixed-point and floating-point by an even bigger margin 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.

8 Conclusions

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 that range in size from 256 points to 2048 points at performance levels comparable to or exceeding those of high-end floating-point and fixed-

acerbated the situation. For this reason we abandoned the idea of putting two memory functional units into the VIRAM prototype.

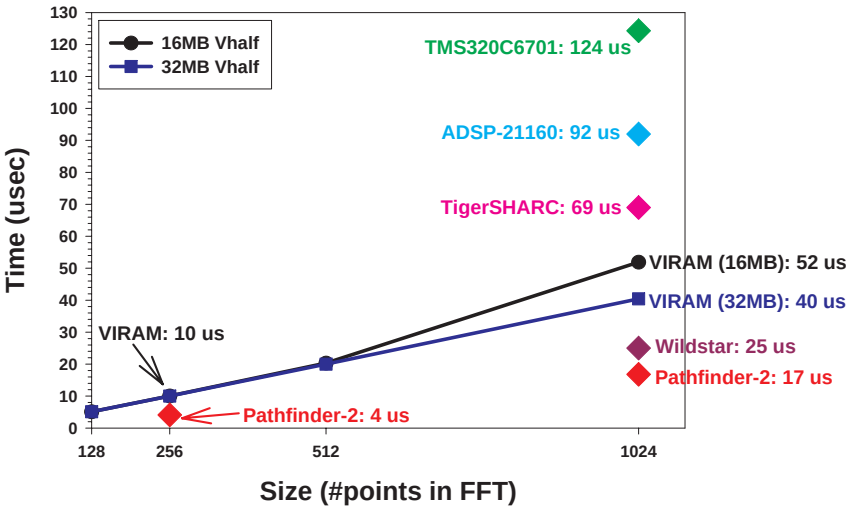


Figure 20: **Performance in microseconds for the *vhalf* 32-bit, single precision, floating-point, N-point FFT implementation for N = 128, 256, 512, and 1024 for two memory configurations: 1) 16MB, 8 banks, and no subbanks (labeled "16MB"), and 2) 32MB, 16 banks, and no subbanks (labeled "32MB"). A $\log_2$ scale is used for the FFT sizes on the x-axis.**

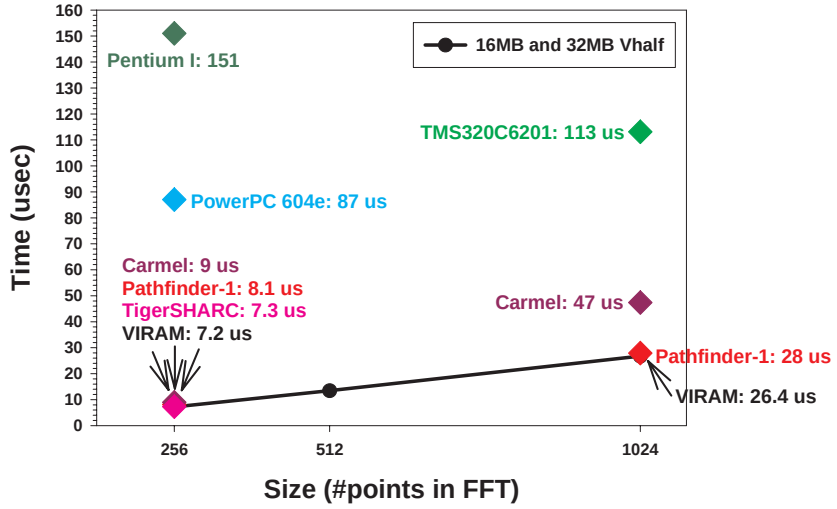


Figure 21: **Performance in microseconds for the *vhalf* 16-bit, fixed-point, N-point FFT implementation for N = 256, 512, and 1024 for two memory configurations: 1) 16MB, 8 banks, and no subbanks (labeled "16MB"), and 2) 32MB, 16 banks, and no subbanks (labeled "32MB"). A $\log_2$ scale is used for the FFT sizes on the x-axis.**

Floating-point DSPs					
Processor	**MHz**	**FFT Size**	**μsec**	**Reference**	**Notes**
Pathfinder-2	133	1024	16.8	[Inc]	Estimated final MHz
Wildstar	N/A	1024	25	[AMS]	FPGA, 1 proc. elmt, no streaming, 32-bit
VIRAM	200	1024	36	this paper	Vector
TigerSHARC	150	1024	69	[Deva]	32-bit, 4-way VLIW
ADSP-21160	100	1024	92	[Devb]	32-bit, Radix 4, SIMD, w/bit rev.
TMS320C6701	167	1024	124.3	[Ins]	Radix 2, w/bit rev, 8-way VLIW
Pathfinder-2	133	256	4.1	[Inc]	Estimated final MHz
VIRAM	200	256	9.5	this paper	Vector
Fixed-point DSPs					
Processor	**MHz**	**FFT Size**	**μsec**	**Reference**	**Notes**
Pathfinder-1	80	1024	27.9	[Inc99]	32-bit, Block FP, used on Scorpio Board
Carmel	250	1024	47.4	[INF]	32-bit, Bit rev?, Custom LIW
TMS320C6201	200	1024	113.1	[Ins]	Radix 2 w/bit rev
VIRAM	200	1024	26.8	this paper	W/Index Load
TigerSHARC	150	256	7.3	[Deva]	Radix 2
TigerSHARC	250	256	4.4	[Deva]	Radix 2
Pathfinder-1	80	256	8.1	[Inc99]	Block FP
Carmel	250	256	9	[INF]	Bit rev?
PowerPC 604E	200	256	87	[Dub98]	AltiVec SIMD, 3-way superscal.
Pentium I	200	256	151	[Dub98]	MMX SIMD
VIRAM	200	256	7.2	this paper	W/Index Load

Figure 22: **Floating-point and Fixed-point running times for 1024-point and 256-point complex FFTs on VIRAM and various DSPs and processors.**

point DSPs and DSP-like architectures. VIRAM outperforms all of the fixed-point DSPs and all but two of the special-purpose floating-point FFT DSPs. Specifically, on 1024-point FFTs, VIRAM achieves 1.3 GFLOP/s in floating-point mode, and 1.9 GOP/s in fixed-point mode. At the same time VIRAM has not compromised accuracy to achieve such performance given that its fixed-point results are at least as accurate as those generated by the current fixed-point DSPs.

VIRAM 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. 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.

By extending the ability of the memory functional unit to decouple itself from the arithmetic functional unit pipeline, by increasing the number of subbanks in the memory system in order to minimize memory bank conflicts, by adding an additional memory functional unit to take advantage of such a robust memory configuration, and by speeding up the indexed memory accesses, a VIRAM commercial version could not only improve its already outstanding performance for the small sized FFTs in the 128- to 2048-point range, but it could also significantly improve its performance on the 8192-point and larger sized FFTs as well.

Such enhancements would further increase the utilization of the VIRAM hardware especially for the narrower data widths where the VIRAM architecture is currently being under utilized despite its high performance relative to the DSPs.

We have seen that the VIRAM architecture's "system on a chip" approach is pushing the envelope of the memory hierarchy to newer levels. We found in our archtectural analysis no evidence of the old bottlenecks of memory latency and bandwidth constraints. Instead we found the new boundaries and limits in the realm of memory bank conflicts and memory cycle times. VIRAM has changed not only the order of magnitude of the bottlenecks but also the whole philosophy of optimizing for them by exposing the memory configuration as part of the memory heirarchy.

In particular we found that for larger sized FFTs, the number of memory banks and subbanks plays a crucial role in the scalability of our algorithm's performance to large FFT sizes.

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.

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.

[INF] Siemens (Infineon) Carmel http://www.elecdesign.com/magazine/1999/nov2299/dsp/1122dsp1.shtml http://www.carmeldsp.com/products/benchmarks.html http://www.carmeldsp.com/products/product_brief.html.

[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. ADSP-TS001 http://products.analog.com/products/info.asp?product=ADSP-TS001M.

[Deva] Analog Devices. ADSP-TS001 http://content.analog.com/pressrelease/prdisplay/0,1622,125,00.html. http://products.analog.com/products/info.asp?product=ADSP-TS001M.

[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/CRIProducts/chips/pathfinder2.htm.

[Inc99] Catalina Research Inc. Cri web site: Press releases, April 1999. http://www.cri-dsp.com/CRIProducts/chips/pathfinder.htm.

[Ins] Texas Instrument. Tms320c6000 platform overview. http://www.ti.com/sc/docs/products/dsp/c6000/62bench.htm.

[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.