Real-Time High-Throughput Sonar Beamforming Kernels Using Native Signal Processing and Memory Latency Hiding Techniques

Gregory E. Allen
Applied Research Laboratories:
The University of Texas at Austin
Austin, TX 78713-8029
gallen@arlut.utexas.edu

Brian L. Evans and Lizy K. John
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084
{bevans,ljohn}@ece.utexas.edu

Abstract

We evaluate the use of native signal processing with loop unrolling and software prefetching to achieve high-performance digital signal processing on general-purpose processors. We apply these techniques to minimize the number of processors necessary for real-time implementation of a 3-D sonar beamformer. Because our beamforming kernels operate on high-throughput (~100 MB/s) input/output streams, memory latency hiding techniques are key for maximum performance. On the Sun UltraSPARC-II processor, we find speedups of 2.4 for hand loop unrolling, 1.46 for the Visual Instruction Set over floating-point arithmetic in C, and 1.33 for software prefetching.

1. Introduction

Today's high-performance general-purpose CPUs make it feasible to perform substantial native signal processing (NSP) [1,13] on the main CPU of a workstation. In addition to single-cycle multiply-accumulates (MACs), several manufacturers have added single-instruction multiple-data architecture extensions to their general-purpose processors, intended to enhance performance in multimedia applications. One such example is the Visual Instruction Set (VIS) in the Sun Microsystems [8] UltraSPARC processor.

Real-time sonar beamforming algorithms can require on the order of billions of MACs per second, and traditionally been implemented in custom embedded hardware. Native signal processing on multiprocessor workstations make a real-time implementation with commodity hardware possible, at a fraction of the development of custom hardware solutions. Modern workstations can also provide fixed-priority scheduling for real-time applications without an embedded real-time operating system.

G. Allen was supported by the Independent Research and Development Program at Applied Research Laboratories: The University of Texas at Austin. B. Evans was supported by the Defense Advanced Research Projects Agency (DARPA) and the US Army under DARPA Grant DAAB07-97-C-J007 through a subcontract from the Ptolemy project at the University of California at Berkeley.

The objective of this research is to develop, optimize, and evaluate the performance of two key sonar beamforming kernel routines on Sun UltraSPARC-II processors, and to attempt to obtain maximum performance. These kernels implement a high-resolution beamformer which combines the outputs of vertical and horizontal sensor elements to image an underwater environment in three dimensions. The multi-fan vertical beamforming kernel requires integer-to-float conversion and multiple dot products, and is implemented with VIS. The horizontal beamforming kernel performs index lookup and digital interpolation, and is implemented in single-precision floating-point format.

Even with NSP advances, access to high-latency main memory causes the processor to stall. The stream-oriented nature of beamforming amplifies this problem because memory I/O is high. Memory latency hiding techniques [2] can be used to help alleviate this bottleneck and improve kernel performance. Therefore, an additional goal of this research is to examine these techniques as they apply to signal processing kernels.

2. Sonar beamforming

High-resolution sonars generally consist of an array of underwater sensors along with a *beamformer* to determine from which direction a sound is coming. The sensor element outputs must be combined to form multiple narrow beams, each of which "looks" in a single direction and is insensitive to sound in neighboring directions.

Time-domain beamforming is realized by weighting, delaying, and summing the array outputs. The time delays are determined by geometrically projecting the sensor elements onto a plane that is perpendicular to the "pointing angle" for the desired beam, as demonstrated in Fig. 1. The distance from each element location to the perpendicular line, divided by the speed of sound, is the necessary time delay for the corresponding element. In Fig. 1, only 56 of the 80 elements are used. The response in the direction of interest is relatively small for the remaining elements, and

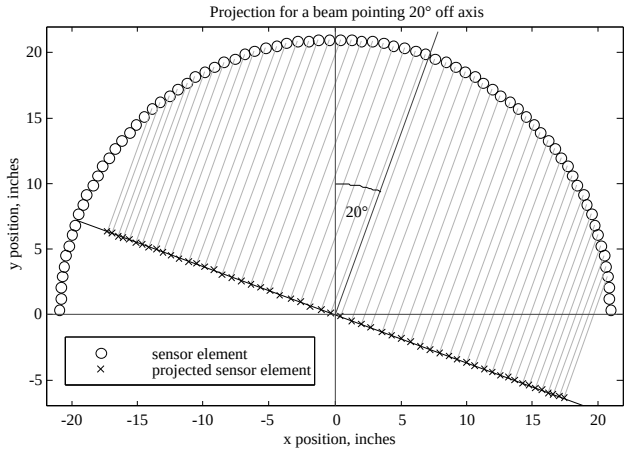

Fig. 1: Projection of sensor elements from a semi-circular array

leaving them out substantially reduces computation.

In a digital system, quantization of the time delays can distort the beam pattern. Generally, the time delay resolution required to alleviate this distortion is much finer than the sampling period that satisfies the sampling theorem [4]. Rather than sampling at a higher rate, digital interpolation beamforming achieves the desired time-delay quantization by interpolating the sampled sensor data. This reduces the quantization error at the expense of increased computation.

We form 61 beams from 80 elements, using on average 50 horizontal elements per beam, with two-point time-delay interpolation. For each beam sample output y_b, two consecutive time samples of each sensor input x_s are delayed, weighted, and summed. The delay τ_{bs} is an integer sample delay for beam b and sensor s. The weighting α_{bsj} contains the delay fraction and any desired beamformer shading. For each beam sample, this requires on

$$y_b[i] = \sum_{s=0}^{S-1} \sum_{j=0}^{1} \alpha_{bsj} x_s[i - \tau_{bs} - j]$$

average 50 index lookups and 100 MACs. To compute one sample of all 61 beams, we must perform approximately 3000 index lookups and 6100 MACs.

For 3-D coverage, there are multiple vertical transducers for every horizontal element. Prior to horizontal beamforming, vertical beamforming groups each vertical transducer column into logical horizontal elements. The vertical outputs are formed as a dot product of the associated transducers – to form 3 sets of 80 elements with 10 vertical transducers, 2400 MACs are required. The vertical beamformer input comes directly from an A/D converter, so samples are in integer format. The output must be in floating-point format for the subsequent horizontal beamformer. Therefore the MACs may be performed in either integer or floating-point format, and integer-to-float conversion must be performed. This is an ideal candidate for VIS, which performs integer arithmetic.

3. The Visual Instruction Set

Sun's UltraSPARC includes the Visual Instruction Set (VIS), which is specifically optimized for video and image processing [5]. VIS adds over 50 new CPU instructions, including basic arithmetic and logic, packing and unpacking partitioned data types, alignment, data conversion, and others. VIS treats a 64-bit register as 2, 4, or 8 partitioned data words, and performs operations on multiple words with a single instruction. VIS instructions can be called as inline functions from C. Although not specifically designed to support 1-D signal processing, VIS is an attractive target with substantial performance on fixed-point algorithms [6].

The vertical beamformer requires the slowest, highest precision mode of VIS – signed 16-bit by 16-bit multiplication and signed 32-bit accumulation. VIS does not directly implement a 16x16-bit multiply, but provides two different 8-bit by 16-bit multiplies (fmuld8sux16 and fmuld8ulx16) for this operation. Two partitioned 16x16->32 MACs are performed with two 8x16 multiplies and two 32-bit adds. The peak pipelined performance for this mode is one operation per clock (where a MAC is two operations). To approach peak performance, we employ memory latency hiding techniques.

4. Memory latency hiding techniques

As processors become faster and memories grow larger, the latency for accesses to global memory will increase. To reduce this latency, systems offer cache memories which can be accessed very rapidly, but have limited storage capacity. Enormous benefits can be achieved by arranging computation to reuse cached data, but this approach still must pay the initial cache loading penalty. Furthermore, some algorithms cannot easily be arranged to take advantage of the sequential access model of caches, and must pay the penalty for many cache misses. Software techniques which can help to hide these memory access delays include loop unrolling, software pipelining, and software prefetching.

Loop unrolling is a technique to enlarge a program's basic blocks – multiple loop copies are combined to form a new, larger loop. The instructions in this new loop can be carefully rescheduled to improve performance (in addition to the reduced looping overhead). By increasing the time between the data request and data consumption, the memory latency can be overlapped with useful computations. Loop unrolling is also important for increasing instruction-level parallelism. It can relieve data dependencies, allowing multiple independent operations in a single cycle. Loop

unrolling has few risks and no overhead associated with it that may nullify its benefits, given sufficient registers to unroll the loop. This optimization technique is a commonly used by modern compilers.

Software pipelining is a technique in which memory accesses and computations are overlapped from different iterations in a program loop. This technique can provide performance gains, especially in low-latency environments [2]. However, this technique increases the number of registers required and the register lifetimes because a register has to be associated with pre-loaded data. This technique is more complex than loop unrolling for a compiler to implement. Loop unrolling with instruction rescheduling can be considered a form of software pipelining.

In the software prefetching [3] technique, the processor includes a non-blocking *prefetch* instruction that causes data at a specified memory address to be brought into the cache. These prefetch instructions can be issued at some time prior to when the data is needed, so that the memory latency can be overlapped with other useful computation. Later, when the load instruction is issued, the data is already cached and can be quickly accessed.

Many programs have memory access patterns which are highly predictable, allowing the compiler to manage prefetching effectively. Unlike loop unrolling, prefetching must be used correctly so that the benefits are not nullified. Prefetching consumes some overhead to calculate effective addresses and issue prefetch instructions, and it consumes extra cache space. On machines that issue multiple instructions per cycle, this technique is particularly useful because the prefetch instructions can be executed for free.

One of the enhancements to the UltraSPARC-II architecture is the implementation of prefetch instructions [7], which were implemented in the spirit of the literature [3]. In this paper we wish to evaluate the performance gains achievable with the inclusion of software prefetching.

5. Tools and methodology

Our goal is to achieve maximum performance for the horizontal and vertical beamforming kernels, measured as execution time, using every means at our disposal. Other statistics gathered (such as cache performance) provide insights into kernel performance, but are of secondary importance. In pursuit of this goal, several tools were utilized, including the SPARCompiler 5.0, Shade, INCAS, and the UltraSPARC performance counters.

The SPARCompiler is Sun's compiler for the SPARC processor, and is generally regarded as the best optimizing compiler for the SPARC. The recent major release (5.0) adds the ability to issue prefetch instructions. For programming with VIS, the compiler includes inline assembly macros which are called like C functions, and are optimized to a single opcode. For this project, similar macros were written for prefetch and fitos (an opcode to convert 32-bit integers to 32-bit floating-point). The SPARCompiler is used to compile all beamforming kernels. Level 5 compiler optimization is turned on at all times, but profiling feedback optimization was not used. For all results in this paper, the compiler is attempting to perform its own optimizations in addition to the loop unrolling (and other such optimizations) in the source code.

Shade [9] is a performance analysis tool from Sun which can perform instruction set simulation, trace generation, and custom trace analysis. It is useful for obtaining detailed, dynamic, instruction-level information about a program. We developed and verified a Shade analyzer called `pficount` , which counts the number of prefetch instructions issued by a compiled program.

INCAS (It's a Near Cycle Accurate Simulator) [10] is Sun's near cycle accurate model of the UltraSPARC-I processor. It offers a convenient way to count program execution cycles and remove pipeline stalls at the cycle level. Although INCAS models the UltraSPARC-I instead of the UltraSPARC-II (which has larger caches and implements the prefetch instruction), it is very useful for code optimization. Drawbacks of INCAS are that it is computationally intensive (and therefore slow for large benchmarks), and that it requires examining code at the instruction level. By pre-warming the cache, using small benchmarks, and assuming perfect prefetching, very insightful kernel optimization clues can be obtained from this tool.

Many processors (including the UltraSPARC) have hardware performance counters which can count events such as instructions, load stalls, and cache hits. The `perfmonitor` [11] tool allows user access to these features on the UltraSPARC-II. This tool consists of a loadable kernel module that accesses these performance instrumentation counters and a configurable application tool which can measure the countable performance parameters of an executable (similarly to the Unix `time` tool). It is also possible to access the instrumentation counters directly so that only the NSP kernel performance is measured. This tool is almost completely non-invasive and non-sanitized, and gives actual run-time performance results at real time.

Because execution time is our primary indication of performance, care must be taken in its measure. Benchmarks are performed on a Sun Ultra Enterprise 4000 workstation with eight 336 MHz UltraSPARC-II processors, 2 GB of RAM, and Solaris 2.6. This sever class multiprocessor machine is a realistic target for computationally intensive beamforming algorithms [12]. In this paper, kernel performance is measured on a single processor, and scaling performance is purposely ignored (see [12]). Software prefetching has been shown to perform well in high memory latency systems [2], such as this server.

Execution time is calculated as the average of ten trials for a reasonably sized benchmark. Execution is performed with all memory allocated and locked, with real-time scheduling priority, thus preventing virtual memory page faults and preemption by other processes. Benchmarks are performed when the machine is not heavily loaded, to reduce unrelated memory traffic. These precautions prevent large time variations across multiple trials; typical standard deviations are below 0.5% of the mean.

DSP algorithms, including beamforming, are commonly specified and measured in millions of floating-point operations per second (MFLOPS), where each multiply and each add count as an operation. After measuring the execution time, we calculate the useful MFLOPS from the known number of operations for the algorithm. MFLOPS is a reasonable number for comparison, and can be compared to the known peak for the processor. To make a fair comparison, we use MIOPS (millions of integer operations per second) when the algorithm is implemented in integer math. It is again computed from execution time and the known number of operations for the algorithm. This is *not* MIPS – looping and addressing additions are not counted – only the same operations that would be counted for MFLOPS. For comparison, instructions per cycle is also presented in select locations.

6. Results and analysis

Level 5 optimization for the SPARCompiler5.0 Developer Release was used, with prefetching turned on, and profiling feedback optimization turned off. Despite extensive attempts with compiler flags, we could never get the SPARCompiler to automatically issue prefetch instructions (as verified with the `pficount` tool). For all results presented, each evaluated optimization case (such as loop unrolling) is implemented in the source code, with the compiler attempting to perform its own subsequent optimizations. Results are presented as useful MFLOPS (or MIOPS), which are calculated from the measured execution time and the number of operations in the algorithm.

The horizontal beamforming kernel performs approximately 3000 index lookups and 6100 MACs per sample. For 64K samples, about 804 million floating-point operations are performed. Fig. 2 shows performance results vs. unrolling the outer loop (by time) in the source code. The best performance obtained was 444.3 MFLOPS, or 1.81 seconds. The processor is computing 1.32 FLOPs per cycle (66% of peak), and issuing 2.19 instructions per cycle.

This experiment clearly demonstrates the positive effect of loop unrolling. Here, unrolling the outer loop reduces the number of loads. Computing a single point in the inner loop requires 5 loads for 4 FLOPs. When unrolling, each iteration requires only 1 more load for 4 more FLOPs.

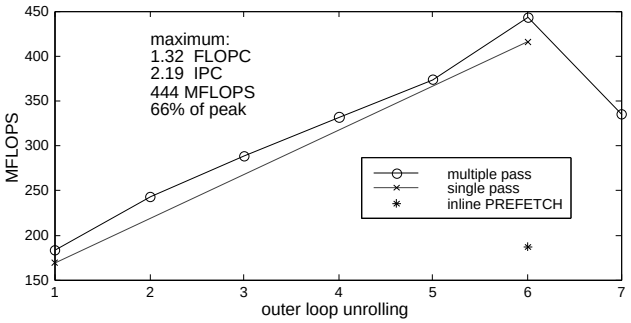

Fig. 2: Horizontal beamformer kernel performance

Continuing to unroll until all registers are exhausted gives the best performance.

This experiment also indicates a benefit for reducing the problem size to fit in the (internal) D-Cache. Calculation of each sample requires approximately 36 Kbytes for coefficients and 12 Kbytes of data, but the cache is only 16 Kbytes. By making multiple passes through the data and calculating a subset at each pass (6 were used), we reduce the problem size. If we keep the data used for all passes small enough to fit in (external) E-Cache we must fetch from main memory only on the first pass. For the 4 MB external cache used in this experiment, this is several thousand samples. The improvement for multiple passes is helpful but not outstanding. At the peak, a 6.7% improvement is measured. Other statistics include a 4.1% improvement in D-Cache read hit-ratio (92.8%) and a 4.9% reduction in memory load pipeline stall cycles (27.3%).

These optimizations require too much high-level knowledge of the algorithm for the compiler to perform them. However, the compiler optimizes very well when exposed to this much parallelism in a basic block, and performs its own inner loop unrolling. This loop is in fairly high-level C, written especially to suit the compiler. By generating assembly output, one can determine the compiler's optimization performance. The compiler seems to perform best on simple loops with large expressions. The form of these loops is surprisingly fragile – attempts at low-level optimizations that the compiler can perform (like common subexpression elimination) are best left to the compiler.

Software prefetching is also better left to the compiler, but could not be made to work. By inserting inline function calls from C, we can issue prefetch instructions. In an attempt to evaluate prefetching with the horizontal beamformer, a single prefetch instruction was inserted into the inner loop of the best-performing code. As the asterisk in Fig. 2 shows, this gave very poor results – a slowdown of 2.37. This clearly demonstrates the fragility of the compiler loops. Despite the fact that only one assembly opcode has been added, the compiler no longer performs aggressive optimization on the loop (verified with assembly output).

The vertical beamforming kernel performs 2400 MAC

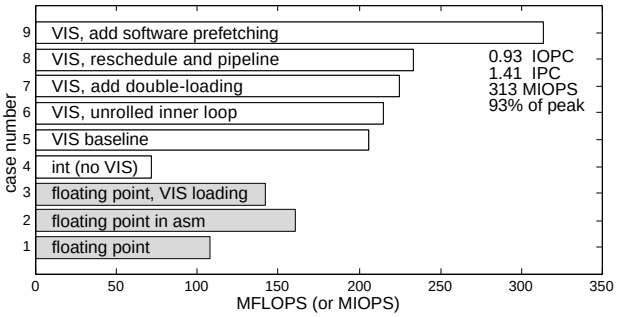

Fig. 3: Vertical beamformer kernel performance

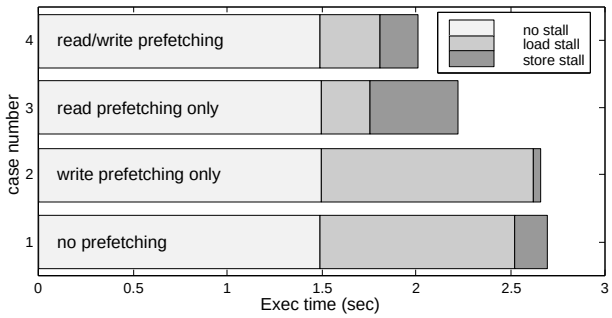

Fig. 4: Vertical kernel prefetch performance

operations per sample. For this benchmark 128K samples are computed, requiring about 629 million operations. Fig. 3 shows performance results in MFLOPS or MIOPS for several different cases. The best performance obtained was 313.3 MIOPS, or 2.008 seconds. This is 93% of peak for 16x16->32 MACs, and issuing 1.41 instructions per cycle.

The computation can be performed in integer or floating-point, with format conversion after or before the calculation. Although the UltraSPARC can perform two FLOPs per cycle compared to only one of these VIS operations, data conversion significantly slows the floating-point version. In Fig. 3, floating-point implementations are grey and integer implementations are white. VIS with unrolled with rescheduling and pipelining (case 8) offers a 46% performance boost over the floating-point version (case 2).

Using the fastest VIS vertical beamformer, we examine the effects of software prefetching. Note that (internal) D-Cache statistics do not change with the addition of prefetch instructions. However, the number of load and store stall cycles change dramatically. Fig. 4 shows a breakdown of execution time, categorized as load stalls, store stalls, or no stall (execution) time. (All other stalls combined are less than 1%). The execution time remains consistent throughout the trials, but read/write prefetching reduces load/store stalls. Prefetching gives the vertical beamformer a 33% performance gain.

7. Conclusion

Native signal processing on the UltraSPARC-II can give very good results. On a 336 MHz processor we achieved 444.3 MFLOPS with the horizontal beamformer, and 313.3 MIOPS with the vertical beamformer. Loop unrolling as a means for memory latency hiding and increased instruction-level parallelism provided excellent results in both kernels. In the horizontal kernel, it gave us a speedup of 2.4. In the vertical kernel, VIS gave a speedup of 1.46 over floating-point, and software prefetching gave an additional speedup of 1.33. The lack of compiler-generated prefetch instructions in our kernels was disappointing.

Low-level programming with VIS or prefetch instructions is difficult and time consuming, similar to assembly or DSP code. For this code, the compiler instruction scheduling is poor, and we must be concerned with instruction-level detail with a tool like INCAS. However, excellent performance gains can be achieved if the expense of hand optimization is justifiable.

References

[1] P. Lapsley, "NSP Shows Promise on Pentium, PowerPC," *MicroProcessor Report*, 1995.

[2] L. John, V. Reddy, P. Hulina, and L. Coraor, "A Comparative Evaluation of Software Techniques to Hide Memory Latencies." *Proc. of the 28th Hawaii Int. Conf. on System Sciences*, Vol. I, pp. 229-238, Jan. 1995.

[3] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching." *Proc. IEEE Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 40-52, Apr. 1991.

[4] R. G. Pridham and R. A. Mucci, "A Novel Approach to Digital Beamforming." *Journal Acoustical Society of America*, vol. 63, no. 2, pp. 425-434, Feb. 1978.

[5] *Visual Instruction Set User's Guide*, Sun Microsystems, 1995.

[6] W. Chen, H. Reekie, S. Bhave, and E. Lee, "Native Signal Processing on the UltraSPARC in the Ptolemy Environment." *Proc. IEEE Asilomar Conference on Signals, Systems & Computers*, pp. 1368-1372, Nov. 1996.

[7] D. L. Weaver and T. Germond, eds. *The SPARC Architecture Manual, Version 9*, Prentice-Hall, Inc., 1994.

[8] The Sun Microsystems Web Page: `http://www.sun.com/`

[9] *The Shade User's Manual*, Sun Microsystems, 1998.

[10] *Incas User's Guide 2.0*, Sun Microsystems, 1993.

[11] The Perf-monitor Web Page: `http://www.sics.se/~mch/perf-monitor/`

[12] G. Allen and B. Evans, "Real-Time Sonar Beamforming on a UNIX Workstation Using Process Networks and POSIX Threads." *IEEE Trans. on Signal Processing*, to appear.

[13] R. Bhargava, R. Radhakrishnan, B. Evans, and L. John, "Evaluating MMX Technology Using DSP and Multimedia Applications." *Proc. IEEE Int. Sym. on Microarchitecture*, pp. 37-46, Nov. 1998.