AVS Optimization for Cray Y-MP Vector Processing

Karen L. Woys and *Mitchell G. Roth*, Arctic Region Supercomputing Center, University of Alaska, PO Box 756020, Fairbanks, AK 99775-6020

ABSTRACT: *The* Application Visualization System (AVS) *is an interactive tool for scientific visualization distributed by Advanced Visual Systems, Inc. The AVS source code is modularized and highly portable, but is not vectorized to take advantage of the Cray Research, Inc. (CRI) parallel vector architecture. For this project, the following AVS modules were analyzed and optimized for vector processing on a CRAY Y-MP M98 supercomputer: Field Math, Interpolate, Compute Gradient, Field to Mesh, Downsize, and Read Field. Vectorization techniques included the use of CRI library routines, compiler directives, inlining functions, loop unwinding, loop transposition, data type conversion, and creation of temporary variables. Based on user CPU time, speedup factors of up to 345 were obtained through optimization, and vector MFLOPS (millions of floating point operations per second) of up to 187 were attained in the vectorized code.*

1 Introduction

The goal of this project was to optimize several modules of the *Application Visualization System (AVS)*, an interactive visualization product of Advanced Visual Systems, Inc., to run on a Cray Research, Inc. (CRI) CRAY Y-MP supercomputer. The AVS source code is modularized and highly portable, but is not vectorized to take advantage of the CRI parallel vector architecture. The Y-MP uses pipelining and vectorization to overcome the inherent performance limitations of traditional *von Neumann* style computer architectures. Pipelining and vector processing improve the speed at which a computer can perform computations. For this project, the following AVS modules were analyzed and optimized for vector processing on a CRAY Y-MP M98 supercomputer: Field Math, Interpolate, Compute Gradient, Field to Mesh, Downsize, and Read Field.

1.1 Vector Processing

A loop that is executed with vectorized code typically runs 10 times faster than when executed with scalar code. [2] The CRI compiling system automatically vectorizes code when it determines that the changes will not affect the program results. Certain conditions preclude automatic vectorization, however, and require manual intervention by the programmer. Code that is already vectorized may also be further optimized to increase vector lengths. This project shows that minor code changes can yield significant performance improvements on vector processors such as the CRAY Y-MP. Vectorization techniques used include the use of CRI library routines, compiler directives, inlining functions, loop unwinding, loop transposition, data type conversion, and creation of temporary variables.

These techniques, combined with pipelining, can yield very high performance. As a result, huge amounts of data may be generated in a short time period. Analyzing the data becomes a new challenge. Data visualization tools greatly facilitate the seemingly insurmountable task of interpreting page after page of numeric information. A typical visualization may require processing more than a million data points for a single image. Smooth animation requires at least twelve images per second. Since visualization ideally occurs interactively in real time, optimization for speed becomes a particularly meaningful consideration.

1.2 Y-MP Architecture

The CRAY Y-MP is a vector-register machine--all vector operations except load and store are performed between the vector registers. Each CPU consists of registers and functional units. In addition to address and scalar registers, each CPU contains eight 64-element vector registers. An element is 64 bits long. A vector length (VL) register specifies the number of elements in the vector register to be processed by the vector instruction. A 64-bit vector mask (VM) register represents a selection of vector elements stored in a V register--each bit in the VM register corresponds to a single element in the vector. Each functional unit implements portions of the instruction set and can be accessed by one group of registers: address functional units are Address Add and Address Multiply; vector functional units include Vector Add, Vector Shift, Full Vector Logical, Second Vector Logical, and Vector Population/Parity; scalar functional units consist of Scalar Add, Scalar Shift, Scalar Logical, and Scalar Population/Parity/Leading Zero; and floating-point functional units are Floating-point Add, Floating-point Multiply, and Reciprocal Approximation. Because the functional units operate independently, operations can occur in parallel. [2].

The system used for this project, *Denali*, is a CRAY Y-MP M98 supercomputer located at the Arctic Region Supercomputing Center (ARSC), University of Alaska Fairbanks. It has a clock speed of 6.0 nanoseconds (167 Mhz.) and is currently configured with eight processors and one gigaword (8 gigabytes) of main memory. The theoretical peak floating point performance is 333 MFLOPS for each processor. Codes achieving sustained rates of 100 MFLOPS or more are considered to be well optimized.

2 Application Visualization System (AVS) Overview

AVS is a software tool designed to facilitate the analysis of scientific data using real-time interactive display techniques. The AVS structure embodies the principles of modularization, abstraction, and information hiding. The package is composed of modules written in C and FORTRAN, each of which performs a specific function. Using these building blocks, AVS users can interactively construct their own visualization programs by combining the modules into executable flow *networks*. Implementation details are completely hidden from the user, and no knowledge of programming or programming languages is necessary. The user connects the various modules through graphical interface ports to build applications tailored to their particular needs. Figure 1 shows an example of a flow network.

AVS also allows for easy development of new modules by providing a "Module Generator" which serves as a template for creating routines that conform to the standard AVS interfaces. These new modules can be added to existing libraries, or new libraries can be created.

2.1 Optimized Modules

Although hundreds of modules comprise AVS, time and resource constraints allowed analysis and optimization of only six for this project, all of which were from the *Supported* module library. A new library, *Vectorized*, was created to contain the optimized modules. The modules that were optimized for this project were chosen based on frequency of use and potential benefit to be gained from optimization. They represent a cross section of the available types of modules and are frequently used in the typical AVS network. A brief description [1] of each of these modules follows:

- **Field Math** (Filter) - This module performs math operations between fields.
- **Interpolate** (Filter) - This module inputs a 2D or 3D scalar field (any data type, any vector length) and computes intermediate values to change the size of the field.
- **Compute Gradient** (Filter) - This module computes gradient vectors for 2D or 3D data sets.
- **Field to Mesh** (Mapper) - This module transforms a 2D scalar field to a surface in 3D space, represented as a geometry-formatted mesh.
- **Downsize** (Filter) - This module accepts a 2D or 3D data field (any data type, any vector length) and reduces its size by sampling.
- **Read Field** (Data input) - This module reads an AVS field from a disk file or imports data files into AVS field format (Native Field Input).

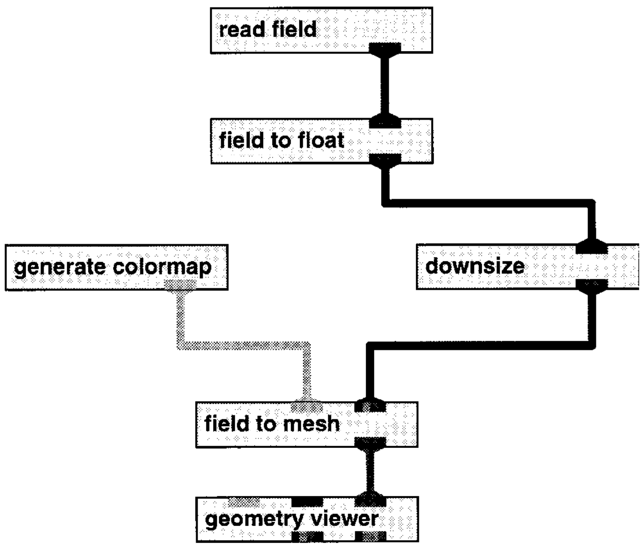


Figure 2: AVS Flow Network

2.2 Remote Module Execution

Another convenient feature of AVS is that it supports the execution of modules on remote AVS hosts of heterogeneous hardware types. Multiple hardware types are possible because the network communication and data transfer mechanism between the AVS kernel and the remote module are based on standard Unix TCP/IP network protocols, and data representation is based on Sun's External Data Representation (XDR) [1]. For this project, to avoid adding to the already heavy user load on *Denali*, the AVS kernel and modules other than the one being analyzed/vectorized were run on an SGI Onyx, with a single module running remotely on *Denali* at any given time.

Remote module execution involves three aspects: remote system requirements; the local *hosts* file that AVS uses to access remote modules; and the *AVS Network Editor* user interface to remote modules. [1]

3 Optimization

Computer performance is generally measured in terms of either time (how long a program takes to execute) or rate (a count of operations over a specified time period). Time may be defined in different ways, including elapsed time, user CPU time, and system CPU time. The measures of performance used in this project are user CPU time for each module and millions of floating point operations per second (MFLOPS). Both of these are industry standards, and CRI's *Perftrace* [3] utility readily provides the desired information.

3.1 Perftrace

Perftrace is a utility that provides statistics about CRI computer hardware performance, detailed for individual program units. *Perftrace* monitors scalar activity, hold-issue conditions, memory use, and vectorization. It can be enabled with FORTRAN, C, and Pascal compilers by using the *-F -lperf* command line option with the compile/load commands. [3] *Perftrace* statistics were collected for each module before and after optimization.

After running a program with *Perftrace* enabled, the *Perfview* command can be used to generate either a default or custom designed report from the raw-format output. *Perfview* has a number of command line options that allow reports to be generated in either interactive or noninteractive modes. An *X-Window* interface is available but not required.

3.2 Vectorizable Expressions

Most vectorization criteria are based on the requirement that a vectorized program produce the same results as its scalar counterpart. In general, a *for*, *while*, *do while*, or *goto* loop is a candidate for vectorization as long as it is an *innermost* loop and consists entirely of vectorizable expressions. The following conditions inhibit vectorization [2]:

- Subprogram calls
- I/O statements
- Nested Loops (except the innermost loop)
- Complexity of expressions within the loop
- Backward branches (except "increment-test-jump" to the top of a loop)
- Bit field manipulations
- Statement branch into a loop from outside the loop
- Data dependencies that produce different answers in scalar mode vs. vector mode
- Ambiguous subscript or pointer references
- Calling functions that do not have a vector version
- Explicitly turning off vectorization via compiler directives or command line options

3.3 Common Vectorization Techniques

CRI provides vector versions of many commonly used *library routines* such as *MIN*, *MAX*, *PACK*, *UNPACK*. If used within a loop, these routines can allow the loop to vectorize. Outside of loops, the routines can speed up processing because they are written specifically to take advantage of the vector processing capabilities of the machine.

The compiler is sometimes unable to determine if an expression is "safely" vectorizable (i.e., it will produce the same result in both scalar and vector calculations). If the programmer can determine that the loop is free of dependencies, a *compiler directive* can be explicitly stated that will force vectorization of a loop.

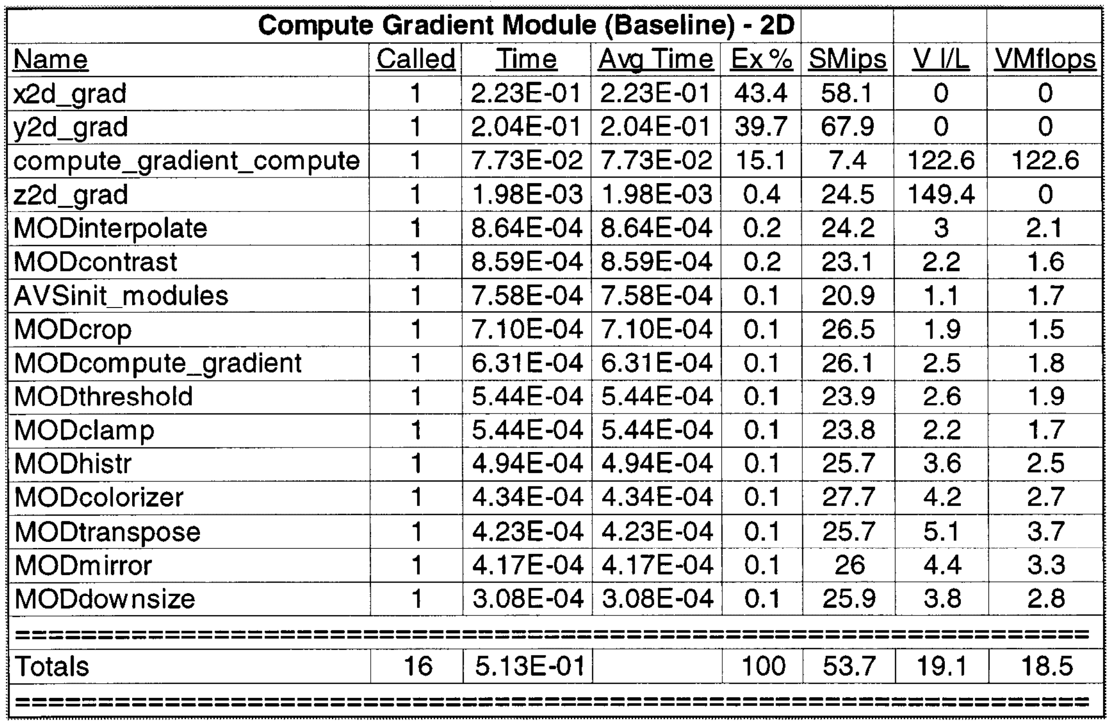
- If source code is available, functions and other subprograms may be *inlined* to allow vectorization. Note that the inlined subprogram must also consist entirely of vectorizable expressions in order for the loop in which it is contained to vectorize.
- If an innermost loop contains a small number of iterations compared to an outer loop, *unrolling the inner loop* to allow the outer loop to vectorize can increase performance.
- *Inverting the order of loops* can remove data dependencies or increase vector lengths to improve performance.
- Character types and bit operations are not vectorizable. *Temporary* conversions to other types can allow vectorization.
- Temporary variables may be created to eliminate dependencies and simplify complexities that inhibit vectorization.
- *Switch Statements* may be converted to equivalent if-then-else structures.

4 Performance Analysis

Perfview may be run either interactively or in batch mode to examine the performance results of the run. Of particular use is the *Perftrace Statistics Report* showing traced routines sorted by CPU time. The following column headings are included in this report [3]:

- **Name** - The name of the individual routine or marked section of code.
- **Called** - How many times the named routine was called.
- **Time** - The total CPU time spent in the named routine.
- **Avg Time** - The average CPU time for each routine call.
- **Ex %** - The execution time percentage of the named routine relative to the total execution time of all routines in the module.
- **SMips** - Scalar MIPS (millions of instructions per second) achieved for the named routine.
- **V I/L** - Vector integer/logical operations per second.
- **VMflops** - Vector megaflops achieved for the named routine.

The report includes totals for all columns. Figure 2 shows the *Perftrace Statistics Report* generated for the Compute Gradient module. From observing the execution times, it is



Compute Gradient Module (Baseline) - 2D							
Name	Called	Time	Avg Time	Ex %	SMips	V I/L	VMflops
x2d_grad	1	2.23E-01	2.23E-01	43.4	58.1	0	0
y2d_grad	1	2.04E-01	2.04E-01	39.7	67.9	0	0
compute_gradient_compute	1	7.73E-02	7.73E-02	15.1	7.4	122.6	122.6
z2d_grad	1	1.98E-03	1.98E-03	0.4	24.5	149.4	0
MODinterpolate	1	8.64E-04	8.64E-04	0.2	24.2	3	2.1
MODcontrast	1	8.59E-04	8.59E-04	0.2	23.1	2.2	1.6
AVSinit_modules	1	7.58E-04	7.58E-04	0.1	20.9	1.1	1.7
MODcrop	1	7.10E-04	7.10E-04	0.1	26.5	1.9	1.5
MODcompute_gradient	1	6.31E-04	6.31E-04	0.1	26.1	2.5	1.8
MODthreshold	1	5.44E-04	5.44E-04	0.1	23.9	2.6	1.9
MODclamp	1	5.44E-04	5.44E-04	0.1	23.8	2.2	1.7
MODhistr	1	4.94E-04	4.94E-04	0.1	25.7	3.6	2.5
MODcolorizer	1	4.34E-04	4.34E-04	0.1	27.7	4.2	2.7
MODtranspose	1	4.23E-04	4.23E-04	0.1	25.7	5.1	3.7
MODmirror	1	4.17E-04	4.17E-04	0.1	26	4.4	3.3
MODdownsize	1	3.08E-04	3.08E-04	0.1	25.9	3.8	2.8
================================							
Totals	16	5.13E-01		100	53.7	19.1	18.5
================================							

Figure 2: Perfview Statistics Report Showing CPU Time, Sorted by Time Used

readily apparent that most of the CPU time is being spent in routines *x2d_grad* (43.4%), *y2d_grad* (39.7%), and *compute_gradient_compute* (15.1%). The other thirteen routines combined used less than 2% of the total time (Figure 3).

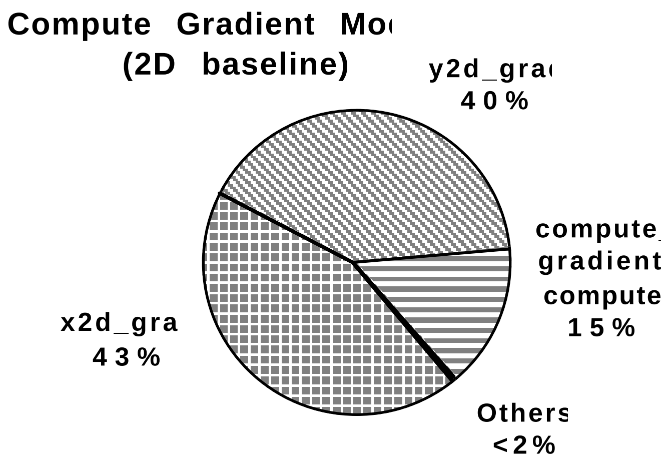


Figure 3: Proportional CPU Times

It is important to direct optimization efforts toward the areas of code that use the most execution time. Therefore, in the Compute Gradient module, *x2d_grad, y2d_grad,* and *compute_gradient_compute* are the routines that are good candidates for optimization. After determining which routines are using the most CPU time, an analysis of the optimization report generated by the C compiler shows which parts of the code are candidates for vectorization.

Figure 4 shows a portion of the baseline optimization report generated for the Compute Gradient module. The messages in this report provide information about vector and scalar code optimizations that were attempted by the compiler, including whether the attempts were successful and reasons for success or failure. These messages indicate that the loops starting at lines 2622 and 2628 are candidates for vectorization. A complete explanation of these and other compiler messages is found in [2].

4.1 Code Changes

An incremental change and test approach was taken. Code was recompiled and tested during intermediate stages to assess the performance impact of individual changes and to facilitate debugging when problems occurred. Source code changes were made conservatively, with the general philosophy that the fewer changes required the better. In many cases, loops are not automatically vectorized simply because the compiler cannot determine if data dependencies exist, particularly when pointers are involved. Often, it easily can be determined by the programmer if these loops are actually safe. If so, the addition of the compiler directive "*#pragma _CRI ivdep*" can cause a loop to be

```
cc-8059 scc: VECTOR File = generic/vex_filters.c, Line = 2622
  Loop starting at line 2622 was not vectorized.  Too much run-time
  analysis is required to safely vectorize the loop.
cc-8043 scc: VECTOR File = generic/vex_filters.c, Line = 2622
  Loop starting at line 2622 was not vectorized.  A recurrence was
  found on a pointer reference at line 2623.
cc-8035 scc: VECTOR File = generic/vex_filters.c, Line = 2627
  Loop starting at line 2627 was not vectorized.  It contains an
  inner loop.
cc-8065 scc: VECTOR File = generic/vex_filters.c, Line = 2628
  Loop starting at line 2628 was not vectorized.  It contains
  character data and unvectorizable operations or data types.
cc-8135 scc: SCALAR File = generic/vex_filters.c, Line = 2622
  Loop starting at line 2622 was unrolled 16 times.
cc-8135 scc: SCALAR File = generic/vex_filters.c, Line = 2618
  Loop starting at line 2618 was unrolled 16 times.
cc-8004 scc: VECTOR File = generic/vex_filters.c, Line = 2640
  Loop starting at line 2640 was vectorized.
```

Figure 4: Excerpt from CRI Standard C Compiler Optimization Report

vectorized. Changes of this type were made first. *Switch* statements easily can be converted into equivalent *if-then-else* statements. Unvectorizable character data can be converted to long integers for intermediate processing and reconverted to bytes afterwards using the vectorized CRI library routines, *_unpack* and *_pack*. Changes of these types were made next. Additional changes, described in greater detail in the following sections, were made based on the remaining compiler messages generated for each module. After loops were vectorized, nested loops were examined to determine if better vector lengths could be achieved by reordering the loops.

4.2 Testing

To verify that the vectorized code produced the same results as the baseline code, the *Write Field* module was included in each AVS test network for baseline modules, connected directly to and immediately following the module being optimized. This module writes the input field to a specified file. The same AVS networks were used to test the optimized modules except that the *Write Field* module was replaced with the *Compare Field* module, connected directly to the optimized module on the left port and connected to the baseline *Read Field* module on the right port to input the previously written baseline data. The *Compare Field* module accepts two input fields and either shows the differences between the fields or indicates that the fields are alike.

5 Field Math Module

The Field Math module is used to perform arithmetic and logical operations on fields such as *NOT*, *AND*, *OR*, *XOR*, left shift, right shift, +, -, *, /, square, square root, and root mean square. The inputs consist of one or two fields of any type and size. Binary operations may be performed either with a single field and a constant or between two fields. The two fields must have the same dimensionality, size, and vector lengths. If data types differ, the simpler type is converted to the more elaborate. For computations, bytes are converted to integers, and shorts, integers, and floats are converted to doubles. Output values are clamped to applicable ranges. Values may also optionally be normalized. The output is a field matching the type and size of the input. This module is computationally intensive.

Before optimization, the Field Math module contained 157 loops, seventy-one of which vectorized fully. Thirty-two loops vectorized with a computed safe vector length. This means that vector code was generated for the loops but depending on run-time conditions, the vector code may be executed with a vector length of only one (effectively, a scalar loop). Eighteen loops were unrolled. Of the 54 loops that did not vectorize, 5 had pointer ambiguities, 40 contained character data, and 9 contained inner loops. According to comments in the code, all functions within the Field Math module except those that performed byte operations had previously been vectorized for a Stellar GS computer.

The AVS networks used to test the Field Math module are shown in Figures 5-6. The Field to Byte, Field to Int, Field to Float, and Field to Double modules were used to convert data to different types for each test case. The Field to Byte module was used to reconvert the data to a format suitable for displaying images for the first test case. The second, third, and fourth test cases did not produce a display. All modules except Field Math were run on an SGI Onyx workstation. The Field Math modules were run remotely on *Denali*.

The field used for the first test case (Figure 5) was the mandrill image supplied with AVS version 5.0 as sample data (*/usr/avs/data/field/mandrill.fld* on *Denali*). This 500 x 480 two-dimensional field with vector length of four was multiplied by -1, and 255 was then added to all elements (field * const + const), producing the complement of the original image. This test case was run separately with byte, integer, float, and double data.

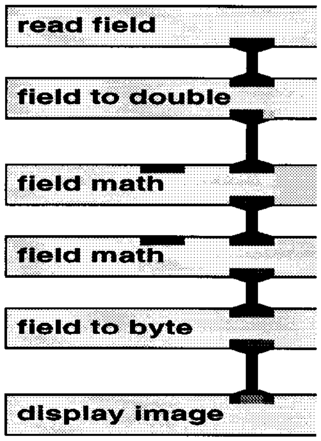


Figure 5: Field Math Module AVS Network for Test Case 1

For byte data, vector MFLOPS were zero, and the module executed for 3.22 seconds. With integer data, vector MFLOPS were 3.9, and execution time was 1.95 seconds.

The routine attained 3.1 vector MFLOPS and executed for 1.84 seconds with float data. For double data, vector MFLOPS were 4.2, and CPU time was 1.36 seconds.

Test cases two, three, and four (Figure 6) used digitized Alaska terrain elevation data (*/u1/uaf/woys/PROJECT/NETS /dem1426x1051.fld* on *Denali*). Test cases 2 and 4 were run separately with byte, integer, float, and double data. Test case 3 was run separately with byte and integer data.

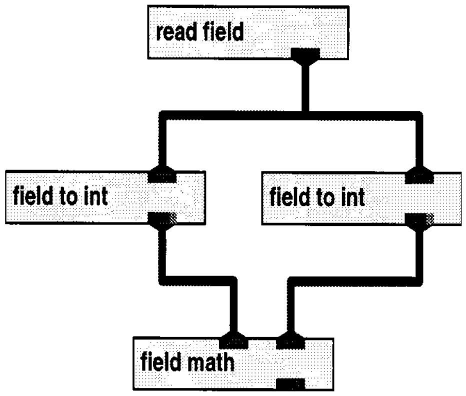


Figure 6: Field Math Module AVS Network for Test Cases 2, 3, 4

The second test case consisted of multiplying this 1426 x 1051 two-dimensional field by itself (field * field). The '*' operator rather than the SQR function was specified in the field math routine.

With byte data, the module executed 4.05 seconds with zero vector megaflops. CPU time was 3.02 seconds with integer data, and vector megaflops were 4.2. For float data, the routine attained 1.5 megaflops and ran 2.93 seconds. With double data, vector megaflops were 4.2 and execution time was 1.08 seconds.

In the third test case, the terrain field was '*XOR*'ed with itselflf (field XOR field). Float and double data did not apply to this test case since they are not legal types for logical operations.

With byte data, vector megaflops were zero and CPU time was 3.87 seconds. The module attained 1.5 megaflops and ran for 3.01 seconds with integer data.

For the fourth test case, the RMS (root mean square) operator was applied to the field. For byte data, CPU time was 5.97 seconds and MFLOPS were zero. The module attained 13.3 MFLOPS and ran for 3.15 seconds with integer data. With float data, 12.7 MFLOPS were obtained, and the module ran for 3.07 seconds. CPU time was 1.28 seconds and vector MFLOPS were 30.5 for the double data case.

The code optimizations for the Field Math module consisted of the following:

1. Character data were converted to long using the vectorized *_unpack* CRI library routine.
2. Pointer ambiguities were eliminated through the addition of "*#pragma _CRI ivdep*" statements.
3. Nested loops were reordered to increase vector lengths.
4. The compiler option "*-h inline3*" caused 11 functions to be inlined.

After optimization, the Field Math module contained 164 loops. Of these, 130 (79%) vectorized fully and fourteen vectorized with a computed safe vector length. Thirty-six loops were unrolled.

In the first test case (field * const + const), integer data yielded 94.4 MFLOPS. With float a speedup of 12.9 was obtained, compared to a speedup of 10.7 for integers. MFLOPS for float data were 67.0. For byte data, the speedup was 8.1, and vector megaflops were 41.3. With double data, speedup was 6.3, and megaflops were 44.3.

In the second test case (field * field), speedups were 29.8, 9.4, 22.5, and 8.8 for float, byte, integer, and double data, respectively. 112.1 MFLOPS were obtained in this test case with integer data, compared to 76.4 for float data, 45.3 for byte data, and 61.1 for double data.

In the third test case (field XOR field), CPU time was reduced to 0.116 seconds for integers, a speedup of 25.9. Vector megaflops were 103.3. For byte data, the speedup was 12.0, and vector MFLOPS were 32.6.

The most impressive speedup in terms of vector megaflops occurred in the fourth test case as shown in Figure 7. 187.4 MFLOPS and a speedup of 11.93 were obtained with integer data. Floating point data yielded 149.5 MFLOPS and a speedup of 10.9. Double data performed nearly as well, with 135.2

MFLOPS and a speedup of 4.1. For byte data, MFLOPS were 105.0 and speedup was 11.0.

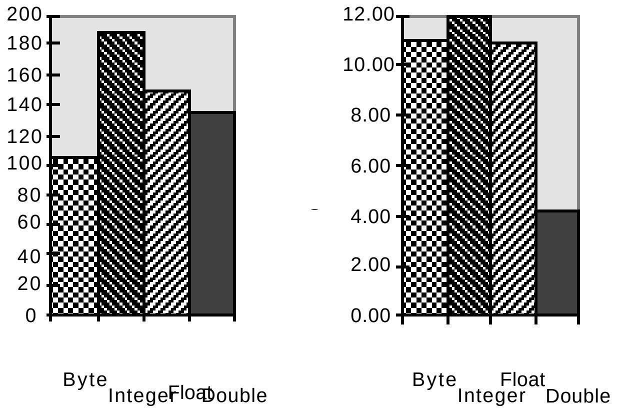


Figure 7: Optimized Field Math Module (Root-Mean-Square(field))

6 Interpolate Module

The Interpolate module uses either point sampling or bi/trilinear sampling to compute intermediate values to change the size of a field. The input consists of a two-dimensional (2D) or three-dimensional (3D) scalar field of any data type. *Interpolate* changes the size of its input data either by subsampling or interpolating. The algorithm first selects, for each output point, its floating-point position in the input data set. With the point sampling method, it selects the closest pixel (voxel) to the computed one. With bilinear (2D) or trilinear (3D) sampling, it finds the four pixels (2D) or eight voxels (3D) around the computed point and does linear sampling for in-between fields. Point sampling is much faster than bi/trilinear sampling. The output is a proportionally resized field matching the type and dimensionality of the input. This module is computationally intensive.

Prior to optimization, the Interpolate module had a total of 32 loops, none of which vectorized. The number of vector MFLOPS was zero for both 2D and 3D test cases, all interpolation methods, and all data types. No loops were unrolled.

The AVS networks used to test the Interpolate module are shown in Figures 8-9. The Field to Byte, Field to Int, Field to Float, and Field to Double modules were used to convert data to different types for each test case. The Field to Byte module was used to reconvert the data to a format suitable for displaying images for the 2D test cases. For the 3D test cases, the Isosurface and Volume Bounds modules were used to produce a displayable geometric image. All modules except Interpolate were run on an SGI Onyx workstation. The Interpolate module was run remotely on *Denali* for all test cases.

The field used for the 2D test cases (Figure 8) was the same mandrill image that was used for the Field Math module. It was interpolated with an x-factor of 2.0 and a y-factor of 1.5 for both point and bilinear methods. Separate runs were made for byte, integer, float, and double data, a total of eight runs to test two interpolation methods and four data types.

For 2D point interpolation, baseline CPU times were 4.87 seconds, 3.29 seconds, 3.09 seconds, and 3.11 seconds for byte, integer, float, and double data, respectively. Bilinear interpolation times for respective data types were 10.0 seconds, 6.03 seconds, 6.52 seconds, and 5.81 seconds.

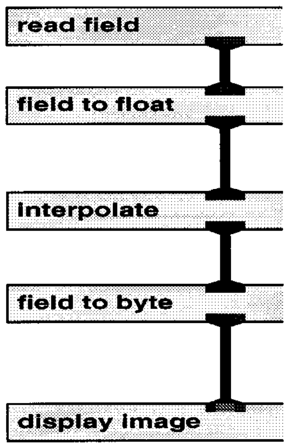


Figure 8: Interpolate Module AVS Network for 2D Test Cases

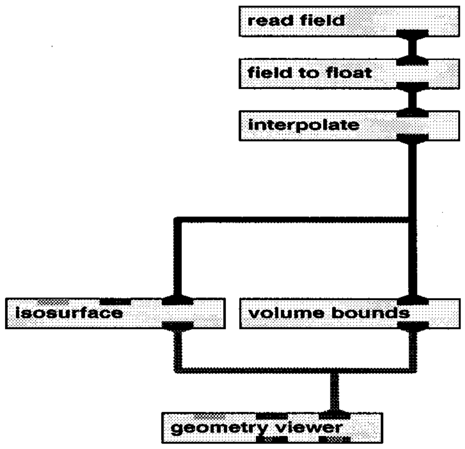


Figure 9: Interpolate Module AVS Network for 3D Test Cases

The 3D test cases (Figure 9) used another field supplied with AVS, a 120 x 120 x 34 element representation of a lobster (*/usr/avs/data/field/lobster.fld* on *Denali*). It was interpolated with an x-factor of 1.5, a y-factor of 1.0, and a z-factor of 2.0 for both point and trilinear methods. Again, separate runs were made for byte, integer, float, and double data for the two interpolation methods.

For 3D point interpolation, byte data ran 6.72 seconds, integer data ran 4.99 seconds, float data ran 4.84 seconds, and double data ran 4.86 seconds. Trilinear interpolation was slower, requiring 27.4 seconds, 17.7 seconds, 15.4 seconds, and 16.0 seconds, respectively, for the four data types.

The code optimizations for the Interpolate module consisted of the following:

1. *Switch* statements were converted into nested *if-then-else* statements.
2. Byte data were converted to type *long* using the vectorized *_unpack* CRI library routine.
3. A temporary float output buffer was created to achieve an acceptable mix of integer and floating point operations.
4. Loops were reordered to achieve greater vector lengths.
5. The "*#pragma _CRI ivdep*" was inserted before innermost loops.

After optimization, there were a total of 46 loops, 26 of which vectorized (57%). For the 2D test cases, maximum vector MFLOPS were 72 and minimum were 45. The maximum speedup was 7 and the minimum was 5. For the 3D test cases, a maximum of 85 MFLOPS and a minimum of 50 MFLOPS were obtained. The maximum speedup was 16 and the minimum was 6.

For the 2D point test case, speedups of 5.1, 8.3, 10.1, and 10.2 were obtained for byte, integer, float, and double data, respectively. Vector megaflops for these respective data types were 45.4, 65.4, 56.7 and 56.1.

For the 2D bilinear case, speedups for byte, integer, float, and double data were 10.0, 6.0, 6.5, and 5.8, respectively. Vector MFLOPS were 64.6, 71.7, 62.0, and 62.0.

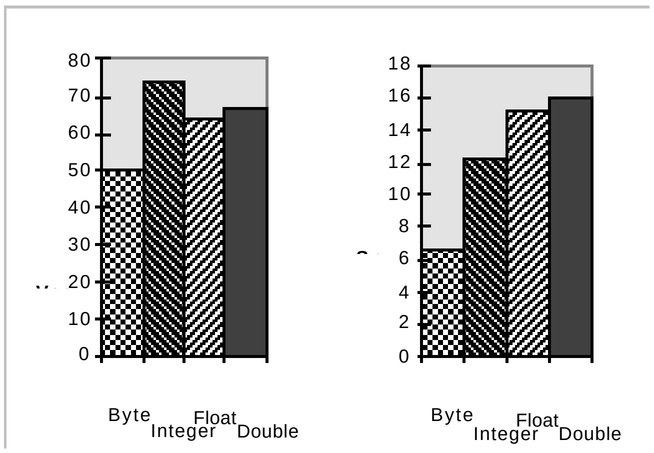


Figure 10: Optimized Interpolate Module (3D Point Method)

As demonstrated in Figure 10, CPU times for the 3D point case were reduced to 1.0 seconds for byte data and less than a second for integer, float, and double data--0.4, 0.3, and 0.3 seconds, respectively. Speedups of 6.6, 12.2, 15.3, and 16.0 and vector MFLOPS of 50.1, 74.3, 63.9, and 63.8 were obtained for the four data types.

The 3D trilinear case obtained speedups of 9.9, 10.2, 9.2, and 9.1 for byte, integer, float, and double data. MFLOPS were 74.9 for bytes, 85.7 for integers, 68.4 for floats, and 65.0 for doubles.

7 Compute Gradient Module

The Compute Gradient module is used to compute gradient vectors for 2D or 3D data sets. The input consists of a scalar byte field. This module computes the gradient vector at each point in a 2D or 3D field of scalar byte data. Data values are in the range 0-255. A "*nearest neighbor*" approach is used to compute the gradient: in each direction, the component of the gradient vector is the previous data minus the next data. This is backwards from the standard definition of a gradient which usually subtracts the previous value from the next. The reason for the change is that the standard definition yields gradients in which the *Z* components typically point in the negative direction. A "*FLIP*" button is included in the module which will calculate the gradient in the conventional fashion. The output is a field matching the dimensions of the input but each field element is a 3-dimensional vector of reals representing the gradient at a particular point. The gradient can be used as a "*pseudo-surface normal*" at each point.

Originally, only 10 of the 55 loops vectorized. Four required too much run-time analysis to be vectorized; in nine loops, the compiler perceived aliased variables; five loops contained character data, and 26 contained inner loops.

The Compute Gradient module accepts only byte data. The AVS network used for the 2D test case is shown in Figure 11.

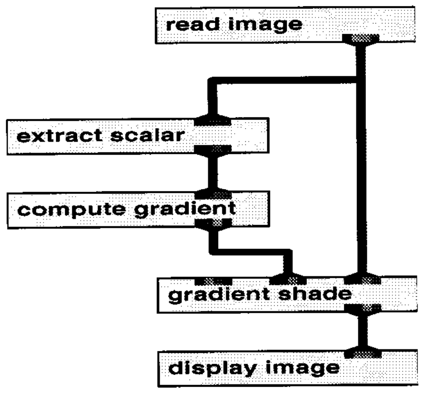


Figure 11: Compute Gradient Module AVS Network for 2D Test Case

The 2D case used the same Alaska digitized terrain field that was used for testing the Field Math module (Section 5). Read Image, Extract Scalar, Compute Gradient, and Display Image modules were run on an SGI Onyx workstation while the Compute Gradient module was run remotely on *Denali*. This network reads an image and uses the computed gradient for shading of particular regions in the digitized terrain image. The 2D test case required 0.5 seconds CPU time and achieved 18.5 MFLOPS.

The 3D case used the lobster field that was used for testing the Interpolate module (Section 6). The AVS network used for this test case is shown in Figure 12. Read Field, Generate Colormap, Colorizer, Gradient Shade, Euler Transformation, Tracer, and Display Image modules were run on an SGI Onyx workstation while the Compute Gradient module was run remotely on *Denali*. This network causes various regions of the lobster image to be shaded based on computed gradient values. The 3D test case ran for 0.7 seconds and attained 12.3 MFLOPS.

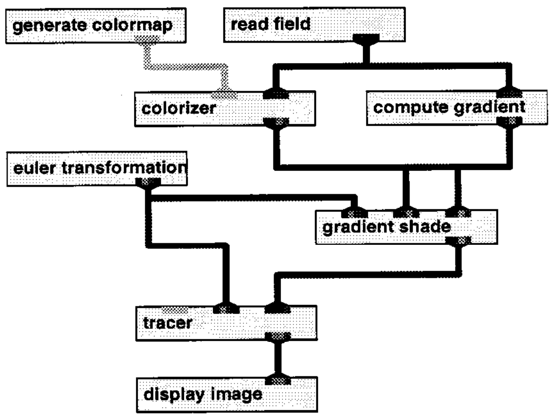


Figure 12: Compute Gradient Module AVS Network for 3D Test Case

The code optimizations for the Compute Gradient module consisted of the following:

1. Invariant expressions calculated inside loops were moved outside the loops.
2. Pointer ambiguities were eliminated by the addition of "*#pragma _CRI ivdep*" statements.
3. A loop with a safe recurrence was vectorized with the addition of a "*#pragma _CRI ivdep*" statement.
4. Character data were converted to type long using the vectorized *_unpack* CRI library routine.

After optimization, fifty-three percent (29) of the 55 loops vectorized fully. Figure 13 shows the results for the Compute Gradient module. Speedups of 3.2 and 2.0 were obtained for the 2D and 3D cases, respectively. Vector MFLOPS were 88.6 for the 2D case and 73.5 for the 3D case.

8 Field to Mesh Module

The Field to Mesh module is used to transform a 2D scalar field to a surface in 3D space. The input consists of a field of any type. Each element of the field is mapped to a point in a base plane. The height of the mesh above each point in the plane is proportional to the scalar value of the field. For irregular fields, the 2D grid of field elements is mapped into 3D space using the coordinate array included in the field description. An optional colormap colors each vertex of the mesh according to the data value at that point. The output is a geometry-formatted mesh.

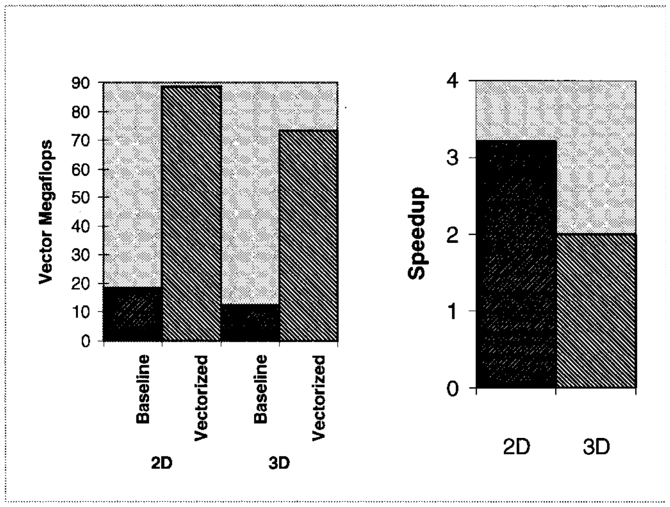


Figure 13: Compute Gradient Module

None of the loops in the unoptimized code vectorized. 20 loops were nested, 2 loops contained character data, 4 loops were too complex, 6 loops contained function calls, and 16 loops had pointer ambiguities.

The AVS Network used to test the Field to Mesh module is shown in Figure 14. Read Field, Downsize, Generate Colormap, and Geometry Viewer modules were run on an SGI Onyx workstation while the Field to Mesh module was run remotely on *Denali*. The test cases for this module used the same Alaska terrain data field that was used for the Field Math and Compute Gradient modules, as described in Section 5.

CPU times for integer and float data were 8.04 seconds and 8.12 seconds, respectively, for the regular field. For a rectilinear field, times were 1.97 seconds and 1.96 seconds for the two data types. Vector MFLOPS were zero in all cases.

The code optimizations for the Field to Mesh module consisted of the following:

1. Pointer ambiguities were eliminated through the addition of "*#pragma _CRI ivdep*" statements.
2. Two loops that were too complex to vectorize were replaced with function calls to the vectorized FORTRAN functions *ISMIN*, *ISMAX*, *INTMIN*, and *INTMAX*.

After optimization, fifteen of the 46 loops vectorized fully, and one vectorized with a computed safe vector length. Speedups obtained were 1.1 for both integer and float data for the regular field and 1.3 for the rectilinear field. Vector MFLOPS for regular and rectilinear fields, respectively, were 0.3 and 0.2 for integer data and 0.2 and zero for float data. The poor speedups were a result of the proportionally large amount of time spent in unvectorized functions that were not inlined.

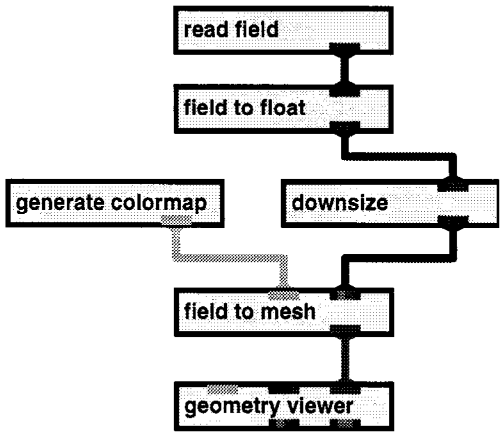


Figure 14: Read Field, Downsize, and Field to Mesh Modules, AVS Test Network

9 Downsize Module

The Downsize module is used to reduce the size of a field while maintaining its aspect ratio. The input consists of a 2D or 3D field of any type. The module extracts the *Nth* element of the field along each dimension, where *N* is the value of the downsize factor parameter. This technique preserves the aspect ratio. The output is a field with the same dimensionality as the input field but the number of elements in each dimension is reduced.

In the baseline code, only one of the sixteen loops vectorized. Six of the loops were not vectorized because of pointer references, one loop contained character data, seven contained inner loops, and one loop contained a function call. The module processed all data types in a common loop by aliasing pointers to access the data as characters. In addition, portions of the code were written with unnecessary conditional statements and modulo operations that slowed processing and inhibited vectorization.

The AVS Network used to test the Downsize module is shown in Figure 14. Read Field, Generate Colormap, Field to Mesh, and Geometry Viewer modules were run on an SGI Onyx workstation while the Downsize module was run remotely on *Denali*. The test case for the Downsize module used the Alaska terrain data field described in Section 5. The CPU time for this test case was 1.64 seconds for byte data and 4.93 seconds for integer, float, and double data. MFLOPS were 13.8 for byte data and 4.6 for integer, float, and double data.

The code optimizations for the Downsize module consisted of the following:

1. Pointer ambiguities were eliminated through the addition of "*#pragma _CRI ivdep*" statements.
2. The nested *if* statements and modulo operations were eliminated by changing the index increments on the nested *for* loops.
3. Byte data were converted to long using the vectorized *_unpack* CRI library routine.
4. Integers were converted to floats temporarily to obtain a proper mix of types for full vectorization.
5. Float and double data were processed in separate loops that maintained original types.
6. The order of loops was rearranged to achieve greater vector lengths. The innermost loop required "*#pragma _CRI ivdep*" statements before it would vectorize.

Figure 15 summarizes the results for the Downsize module. Twelve of the 32 loops vectorized fully. Three loops were unrolled. Of the 20 loops that did not vectorize, 19 contained inner loops and one contained a function call. Speedups of 28.0, 345.1, 345.1, and 329.1 times were obtained for byte, integer, float, and double data, respectively. Running times for the optimized routines were 0.0655, 0.0144, 0.0144, and 0.0155 seconds for the four data types. Since the optimized run times decreased so drastically, *Perftrace* could not gather reliable statistics for MFLOPS.

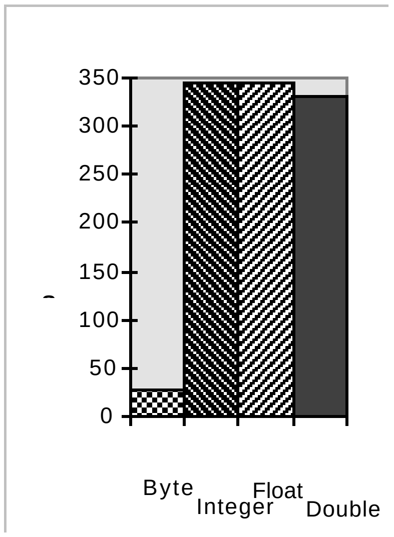


Figure 15: Optimized Downsize Module

Because the speedups were so dramatic, additional analysis was performed on the float test case to verify the results. Code changes were made incrementally with run times checked after each additional modification. It was determined that simply removing the modulo operations and nested *if* statements caused a speedup of 1.4 from the baseline. Processing the data as floats instead of bytes yielded speedups of 11.3 from the previous case and 16.7 from the baseline. Moving the short vector loop to the outside caused speedups of 2.8 from the previous case and 46.4 from the baseline. Finally, the addition of the "*#pragma _CRI ivdep*" to the innermost loops allowed them to vectorize fully and yield speedups of 7.0 from the previous case and 324.5 from the baseline.

10 Read Field Module

The Read Field module is used to read an AVS field from a disk file or import data file into the AVS field format. Input data may be in ASCII, FORTRAN unformatted or pure binary form. The output field is in AVS field format. This module is input/output intensive.

The Read Field module contained no loops. It called another routine, *rf_read_field*, to perform the majority of its work.

The AVS Network used to test this module is shown in Figure 14. Downsize, Generate Colormap, Field to Mesh, and Geometry Viewer modules were run on an SGI Onyx workstation while the Read Field module was run remotely on *Denali*. The module required 159.0 seconds to read the Alaska terrain field as float data and obtained zero vector megaflops.

The only code optimization performed for the Read Field module was manually inlining the *rf_read_field*. The inlined *rf_read_field* function contained 42 loops. Nine of these vectorized fully and two vectorized with a short vector length. Four loops were unrolled. The optimized routine ran in 93.7 seconds, a speedup of 1.7 over the baseline case. Vector MFLOPS remained zero. The results were poor for this routine because it contained many function calls to routines that were not available for inlining and no arithmetic computations.

11 Summary

Figure 16 shows a summary of speedups and MFLOPS for all test cases. Blanks in the table indicate that the specified data type was not applicable for the test case or that results did not differ based on data type, except for the Downsize module, where optimized run times were too short to gather accurate MFLOP statistics. The speedups ranged from a high of 345 times for the Downsize module to a low of 1.1 for the Field to Mesh module.

12 Conclusions

This project demonstrated that minor code changes can yield significant performance improvements for AVS modules executing on CRI vector supercomputers. Speedups of up to 345 times were obtained for CPU time, and up to 187 MFLOPS were attained through the use of simple optimization techniques. Out of approximately 2200 lines of source code, only 350 lines (about 16%) were modified. Optimization methods used for each module are summarized in Figure 17 below.

Compiler directives, either in the form of command line options or pragmas, were used for all six modules. This type of optimization leaves the source code completely portable. Command line compiler directives do not affect the source code at all, and pragmas are treated as comments by non-CRI compilers. The "*#pragma _CRI ivdep*" directive is particularly useful in C code. Because of the prevalence of pointers in C programs, the compiler is unable to automatically vectorize many loops due to the danger of different pointers referencing common memory locations. This pragma can force the compiler to vectorize loops that are not optimized automatically because of perceived pointer dependencies. Forty-three loops

	Speedup				MFLOPS			
	Byte	Integer	Float	Double	Byte	Integer	Float	Doubl
Field Math								
field * const + const	8.1	10.7	12.9	6.3	41.3	94.4	67.0	44.3
field * field	9.4	22.5	29.8	8.8	45.3	112.1	76.4	61.1
field XOR field	12.0	25.9			32.6	103.3		
RMS(field)	11.0	11.9	10.9	4.1	105.0	187.4	149.5	135.2
Interpolate								
2D point	5.1	8.3	10.1	10.2	45.4	65.4	56.7	56.1
2D bilinear	10.0	6.0	6.5	5.8	64.6	71.7	62.0	62.0
3D point	6.6	12.2	15.3	16.0	50.1	74.3	63.9	63.8
3D trilinear	9.9	10.2	9.2	9.1	74.9	85.7	68.4	65.0
Compute Gradient								
2D	3.2				88.6			
3D	2.0				73.5			
Field to Mesh								
Regular field		1.1	1.1			0.3	0.2	
Rectilinear field		1.3	1.3			0.2	0.0	
Downsize	28.0	345.1	345.1	329.1				
Read Field			1.7				0.0	

Figure 16: Summary of Optimization Results

were vectorized as a result of using pragmas, each requiring only a single line modification.

Function inlining is another optimization technique that leaves the source program completely portable while providing performance improvements in both scalar and vector code. One function was manually inlined in the Read Field module. The command line compiler directive "*-h inline3*" caused 17 functions to be inlined automatically by the compiler. The resulting speedup was particularly significant in the Field Math module which attained the most MFLOPS of all the optimized routines. When functions cannot be inlined, performance increases can be greatly inhibited, as is demonstrated in poor results obtained in both the Field to Mesh and Read Field modules. Although many loops were vectorized in both cases, the majority of execution time was spent in functions outside the modules. The limitations imposed by Amdahl's law prevail.

Other simple code modifications can be made that enhance performance without hampering portability. In the Interpolate module, the conversion of switch statements to equivalent if-then-else statements allowed several loops to vectorize. The creation of temporary variables in Field Math, Interpolate, Downsize, and Compute Gradient modules allowed invariant calculations to be moved outside loops and also allowed conversion of character data to vectorizable types. The conversion of characters to another data type and then back to characters again might seem counterproductive to optimization because of the time involved. Speedups in the Interpolate, Downsize, and Compute Gradient modules demonstrated, however, that the overhead is more than compensated for by the performance improvement obtained from resulting vectorization.

Five of the six modules were modified to use vectorized CRI library routines. Although these modifications are not portable to non-CRI machines, they are worthwhile nonetheless. Because the library routines are written in assembly language and are designed specifically to take advantage of the CRI vector architecture, they can provide speedups that would not be possible using high level languages such as C or FORTRAN. Therefore, vectorized CRI libraries should be used whenever possible.

Even after the majority of loops in a routine are vectorized, performance will suffer if vector lengths are too short. A proper analysis of loop ordering to achieve optimal vector lengths requires knowledge about minimum, maximum, and typical input values. In the Field Math, Downsize, and Interpolate modules this information was available since these routines are typically used for image processing in which field vector lengths are limited to four or less to represent color data.

The performance improvements obtained in this project clearly demonstrate the value of AVS code optimization for vector processing. Future work should focus on the AVS kernel and on other AVS modules that are computationally intensive. Poor candidates for optimization are those that contain few loops or that are input/output intensive.

13 Acknowledgments

This project was funded by Cray Research, Inc. and by the Strategic Environmental Research and Development Program (SERDP) under the sponsorship of the Army Corps of Engineers Waterways Experiment Station.

14 References

[1] Advanced Visual Systems, AVS User's Guide, Release 4, Revision B, May 1992.

[2] Cray Research, Inc., SR-2074 8.0, Cray Standard C Programmer's Reference Manual, 1994.

[3] Cray Research, Inc., SR-2040 8.0, Performance Utilities Reference Manual, 1994.

						Creation of	Conversio
	Library	Compiler	Function	Loop	Data Type	Temporary	of Switc
	Routines	Directives	Inlining	Transposition	Conversion	Variables	Statemen
Field Math	X	X	X	X	X	X	
Interpolate	X	X		X	X	X	X
Compute Gradient	X	X			X	X	
Field to Mesh	X	X					
Downsize	X	X		X	X	X	
Read Field		X	X				

Figure 17: Summary of Vectorization Techniques Used for Each Module