Performance Analysis and Optimization of the Trilinos Epetra Package on the Quad-Core AMD Opteron Processor

Brent Leback
The Portland Group (PGI) – a wholly-owned subsidiary of STMicroelectronics
Douglas Doerfler *and* **Michael Heroux**
Sandia National Laboratories

ABSTRACT: *At CUG 2007, we presented a paper describing optimizations applied to the Alegra C++ application which included allowing packed, consecutive-element storage of vectors, some restructuring of loops containing neighborhood operations, and adding type qualifiers to some C++ pointer declarations to improve performance. Additionally, improvements to the PGI 7.1 C++ compiler were implemented to enable better C++ vectorization. In this paper we investigate the performance of Sandia's Trilinos Solver, and specifically its performance on the latest Quad-Core processors from AMD. By implementing a different sparse storage format and tuning a key kernel we are able to demonstrate a 44% to 91% improvement in performance.*

KEYWORDS: Compiler, C, C++, Optimization, Vectorization, Performance Analysis, AMD Opteron™, Intel Core™ 2

1. Introduction

The emergence of multi-core microprocessor architectures is forcing application library developers to re-evaluate coding techniques, data structures and the impact of compiler optimizations to ensure efficient performance.

For current and future massively parallel high performance computing systems, a key factor limiting overall performance and scalability is the ability to move data between subsystems. System performance is dictated by the ability to move data between nodes and the I/O subsystem. Data movement between the memory subsystem and its associated CPU dictates node performance, which is the focus of this paper. We have entered an era in which for many CPUs, FLOPS are free, and extracting performance from the CPU is a function of optimizing the data movement to best utilize the computational resources. The application writer must be cognizant of optimal algorithms and data structure storage techniques. As a result, given proper data layout in main memory, an optimizing compiler can then be uninhibited to extract high performance for a given CPU.

However, these two necessary conditions cannot be performed independently and it is necessary to involve application knowledge and compiler expertise simultaneously to maximize results. The PGI compiler suite has been the mainstream compilation environment at Sandia National Laboratories for several years in its high performance computing initiatives. PGI compiler suites were deployed on the 32-bit Intel x86 processor-based ASCI Red supercomputer in 1997, in addition to Sandia's 2nd generation capability class ASC Red Storm, a 64-bit AMD Opteron processor-based platform deployed in 2004 and still operational. PGI compilers are also deployed on Sandia's many computational clusters, the most recent being the Opteron based ASC Tri-Lab Capacity Computing (TLCC) clusters, being deployed in 2008. Throughout this period, there has been a continual effort to ensure that optimal performance is extracted and utilized on these platforms.

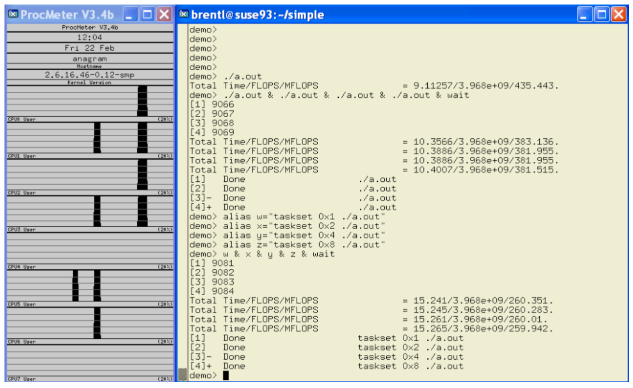
Sandia's Scalable Architectures Group is continually evaluating new technologies and technological trends to determine the impact on its application base. One such trend is the increasing width of SIMD units to increase peak floating-point rates in general purpose processors. At CUG 2007, a paper was presented on optimizing the performance for a kernel from Sandia's Alegra application [1]. This paper is a follow-on effort investigating performance for sparse matrix-vector kernels from the Trilinos solver package.

Packages employed in the Trilinos framework, and in particular the Epetra package, must support the transition to multi-core timely and efficiently. This paper evaluates the performance of Epetra on the AMD Barcelona processor and investigates optimizations to exploit the performance potential of multiple cores and the double-wide SIMD unit.

2. Architecture Considerations on x64 Processors

Recent X64 processors from AMD and Intel have followed a similar, well-documented track. While the speed of the processors seems to have reached a plateau, the number of cores on a socket has increased from one, to two, to four, over the last several years.

It is important when running on newer systems to be clear in the presentation of performance data. Performance obtained running on 1 core of a quad-core system may or may not be sustainable when running the same or similar code on all four cores at the same time. Even running multiple instances of the code, on multiple socket systems, can be misleading. As Figure 1 shows, it is important to lock threads or processes onto all cores of the same socket. A term used for this principle is running "fully subscribed". Taskset is a Linux utility that can set a processes's CPU affinity and the ampersand and wait command are shell job control utilities.



```
demo>
demo>
demo>
demo>
demo>
demo> ./a.out
Total Time/FLOPS/MFLOPS                = 9.11257/3.968e+09/435.443.
demo> ./a.out & ./a.out & ./a.out & ./a.out & wait
[1] 9066
[2] 9067
[3] 9068
[4] 9069
Total Time/FLOPS/MFLOPS                = 10.3566/3.968e+09/383.136.
Total Time/FLOPS/MFLOPS                = 10.3886/3.968e+09/381.955.
Total Time/FLOPS/MFLOPS                = 10.3886/3.968e+09/381.955.
Total Time/FLOPS/MFLOPS                = 10.4007/3.968e+09/381.515.
[1]   Done                    ./a.out
[2]   Done                    ./a.out
[3]-  Done                    ./a.out
[4]+  Done                    ./a.out
demo> alias w="taskset 0x1 ./a.out"
demo> alias x="taskset 0x2 ./a.out"
demo> alias y="taskset 0x4 ./a.out"
demo> alias z="taskset 0x8 ./a.out"
demo> w & x & y & z & wait
[1] 9081
[2] 9082
[3] 9083
[4] 9084
Total Time/FLOPS/MFLOPS                = 15.241/3.968e+09/260.351.
Total Time/FLOPS/MFLOPS                = 15.245/3.968e+09/260.283.
Total Time/FLOPS/MFLOPS                = 15.261/3.968e+09/260.01.
Total Time/FLOPS/MFLOPS                = 15.265/3.968e+09/259.942.
[1]   Done                    taskset 0x1 ./a.out
[2]   Done                    taskset 0x2 ./a.out
[3]-  Done                    taskset 0x4 ./a.out
[4]+  Done                    taskset 0x8 ./a.out
demo> █
```

Diagram 1. Performance of the same binary can vary from 435 MFlops to 260 MFlops depending on where and how many.

As we documented in the CUG 2007 paper, the width of the SSE computational units, for obtaining peak floating point performance, has increased from 64 bits wide to 128 bits wide in the last year. A recent announcement from Intel states they are extending the SSE units to 256 bits wide in a future design[2] . AMD has plans to use fused-multiply-add (FMA) ops in a future product[3]. Both of these architectural changes have the potential to again double the peak floating point performance.

One performance feature which has not seen this "doubling phenomenon" is the memory hierarchy. In fact, given the increase in the number of cores and the SSE width, memory bandwidth as a percentage of peak performance has actually decreased over the last 5 years.

Diagram 2 shows the peak data transfer bandwidth as a percentage of the peak floating point performance for a number of X64 processors which have been available over the last 5 years. These numbers were measured on a number of different types of machines, from laptops to servers, and from single-core-per-socket processors to quad-core. All measurements are with fully-subscribed cores.

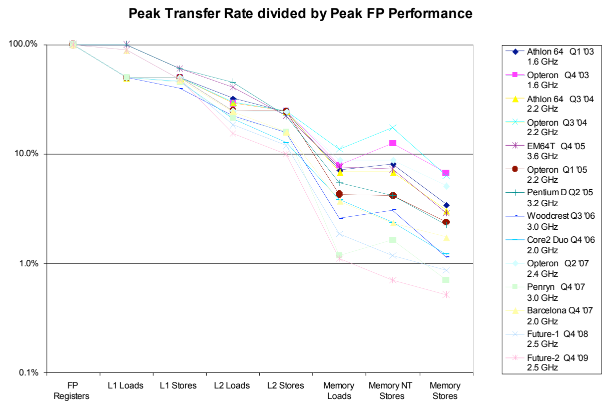


Diagram 2. The decline of the peak rate of data transfer compared to peak floating point performance. Note the logarithmic scale.

The Future-1 and Future-2 processors are hypothetical. The Future-1 processor from Diagram 2 assumes hardware updates which doubles the peak floating point performance (either through FMA or wider SSE units), runs at 2.5 GHz, and has a 25% memory bandwidth improvement. The Future-2 processor also has the Future-1 SSE units and speed, but with eight cores per socket, and an additional 25% memory bandwidth improvement over Future-1.

Diagram 3 shows the sustainable compute intensity for each level of the memory hierarchy. Computational intensity is defined as the number of arithmetic operations (floating point operations, normally) performed per memory transfer. [4]

$$\textit{Compute Intensity} = \frac{\textbf{Total Number of Operations}}{\textbf{Number of Input/Output Data Points}}$$

Today's X64 architectures can perform 2x the number of single precision computations as double precision in their

SSE units, where each data operand is 1/2 the size. Therefore compute intensity is independent of data type.

So, for example, a simple dot product:

```
    for (i = 0; i<N; i++)
       sum = sum + a[i] * b[i];
```

requires 2*N loads and performs 2*N arithmetic operations, for a compute intensity of 1. For simplicity, it is usually assumed the N factors cancel out, but this can become more complicated in nested loops. Also, it is assumed in the intensity calculation that reuse is handled perfectly by the compiler, i.e. for each data element, one load or store counts, the rest are for free.

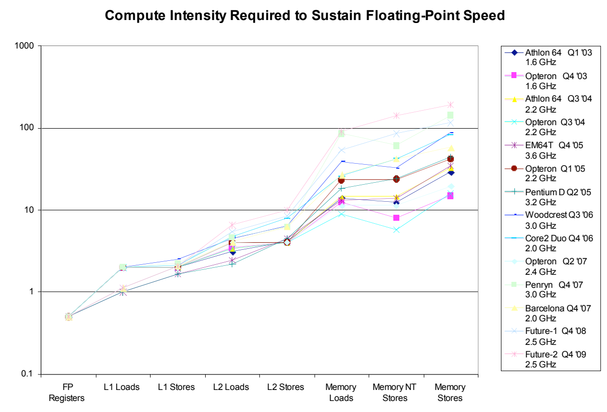


Diagram 3. Inverse of Diagram 2 gives a rough estimate of the compute intensity that can be sustained from each memory level and operation.

Starting with the 7.2 release of the PGI compilers, you can get –Minfo output of the computational intensity of innermost loops (nested loops will follow in PGI 8.0). Here are some examples from simple implementations of vector add, daxpy, and ddot:

```
dvadd:
     3, Intensity = 0.333
        double add:1 ld:2 st:1
daxpy:
    11, Intensity = 0.667
        double add:1 mul:1 ld:2 st:1
ddot:
    21, Intensity = 1.000
        double add:1 mul:1 ld:2
```

As can be seen by Diagram 3, to obtain near peak floating point performance when the data set is too large to reside in the data caches, the compute intensity must be very large (~100). In some cases this can be achieved due to the structure of the calculations. There may be lots of time being spent in math functions such as log(), pow(), exp() or extended precision arithmetic. Perhaps the calculations are an order of N or log(N) higher in magnitude than the data transfers required, as is the case with matrix multiply or FFTs. To find the intensity in these cases, it is necessary to look at the reuse over the non-innermost loops, and again assume the 2nd-through-nth memory accesses are free or nearly free.

A technique employed by compilers and programmers to improve data reuse is termed tiling, blocking (2-d) and/or strip mining (1-d). The goal of this technique is to restructure long nested loops to get finer reuse on data resident in the L1 or L2 caches. Often, by breaking one long loop into 2 blocked or strip-mined loops, the number of streams of data to or from memory in the innermost loop can be reduced from many to one. More complicated techniques, but having the same general idea, have been presented by Sandia authors at CUG [5] before and are termed "cache-oblivious" algorithms. [9][10]

In previous papers [11] we have discussed the benefits of vectorization. It should be noted that on newer processors, vectorization can provide a 2x speedup on double precision loops, and 4x speedup on single precision loops, assuming the data is cache-resident. This is hardly insignificant, but in this paper, we expand on the previous work and attempt to address issues which arise as we move our focus further from the processor core.

In the remainder of this paper we discuss results generated in a cooperative effort between The Portland Group and Sandia's Scalable Architectures Group.. We specifically looked at the Trilinos Epetra package of sparse solvers, and some similar Sandia sparse solver benchmarks. The Trilinos package is a critical component of many of the codes that run at Sandia and other DOE sites, and performance of this package can make a large impact on the overall job throughput of the DOE systems.

3. Targeting Trilinos Epetra Performance

Epetra [6] is a collection of distributed data objects for sparse and dense matrices, vectors and graphs. It is the most heavily used package in Trilinos because it provides matrix and vector services for all other Trilinos packages. Epetra is written for real-valued double-precision scalar field data only, and restricts itself to a stable core of the C++ language standard. As such, Epetra is very portable and stable, and is accessible to Fortran and C users. Epetra combines in a single package (i) support for generic parallel machine descriptions, (ii) extensive use of standard numerical libraries including BLAS and LAPACK, (iii) use of object-oriented C++ programming and (iv) parallel data redistribution. The availability of Epetra has facilitated rapid development of numerous applications and solvers because many of the complicated issues of working on a parallel distributed memory machine are handled by Epetra.

Application developers can use Epetra to construct and manipulate matrices and vectors, and then pass these objects to other Trilinos solver packages. Furthermore, solver developers can develop new algorithms using Epetra classes to handle the intricacies of parallel execution. Epetra also has extensive parallel data redistribution capabilities, including an interface to the Zoltan load-balancing library [7]. Epetra linear operators and matrices are written as concrete implementations of abstract linear operator and matrix bases classes. The most heavily used matrix class is Epetra_CrsMatrix, which stores the user matrix in a compressed sparse row format.

Other Trilinos packages access Epetra services via the base class interfaces and do not rely on the specific way in which matrix data is stored. Therefore, users and Epetra developers can develop new matrix classes, such as classes that use different data structures, and seamlessly use these new classes with the rest of Trilinos, as long as the new classes are accessible via the base class interfaces. We use this extensibility of Epetra in this paper to test the performance of sparse diagonal data structures for matrices that have sparsity structures amenable to this representation.

Some of the results generated in this paper, specifically those related to the sparse diagonal format, are from the Mantevo [8] benchmark code pHPCCG, which captures the performance-impacting features of Epetra sparse matrix kernels, but is much smaller in size than Epetra and easier to use for focused performance studies.

Sparse matrices come in many varieties, and numerous data formats are used to store sparse matrix data for the purpose of computing basic operations. The most common sparse matrix operation by far is sparse matrix times a dense vector, sometimes referred to as SpMV, and usually formulated as y = A*x. In this paper we considered three data formats: compressed row storage (CRS), jagged diagonal storage (JDS) and sparse diagonal storage (SDS). Each format is appropriate in important situations.

To illustrate these data structures, consider the following matrix:

$$A = \begin{bmatrix} 11 & 12 & 0 & 14 & 15 \\ 21 & 22 & 0 & 0 & 25 \\ 0 & 32 & 33 & 34 & 0 \\ 41 & 0 & 0 & 44 & 0 \\ 0 & 52 & 0 & 0 & 55 \end{bmatrix}.$$

Using the CRS format, the matrix is stored in three arrays as follows (the colons are not part of the format):

$$\begin{aligned} values &= [11,12,14,15:21,22,25:32,33,34:41,44:52,55] \\ indices &= [0,1,3,4:0,1,4:1,2,3:0,3:1,4] \\ offsets &= [0,4,7,10,12,14] \end{aligned}.$$

Nonzero elements of each row in the matrix A are "compressed" or packed, row-by-row in a single array. The indices array is the same length as values and contains the corresponding column indices (using zero based indexing). The offsets array contains offsets into the values and indices arrays such that offsets[i] is the first matrix entry of row i. The last value of offsets is set to the number of nonzero entries in the matrix. The CRS format is an excellent general-purpose data structure and is the most popular format in many linear algebra libraries.

Given this default format, a simplified version of the SpMV operation y = A * x is then coded like this:

```
for (i=0; i< nrow; i++) {
  double sum = 0.0;
  double * A_vals = A->ptr_to_vals_in_row[i];
  int    * inds   = A->ptr_to_inds_in_row[i];
  int cur_nnz     = A->nnz_in_row[i];
  for (j=0; j< cur_nnz; j++)
    sum += A_vals[j]*x[inds[j]];
  y[i] = sum;
 }
```

When compiled with PGI 7.2, we get this information output:

```
67, Intensity = 1.000
          double add:1 mul:1 ld:2
          integer add:2 ld:1
Generated vector sse code for inner loop
Generated 1 prefetch instructions for this loop
```

For other operations in the solver process, the loop may be transposed and appear something like this, and get this –Minfo output:

```
xtmp = x[i];
for (j=0; j< cur_nnz; j++)
  y[inds[j]] -= A_vals[j]*xtmp;
```

```
93, Intensity = 0.667
          double add:1 mul:1 ld:2 st:1
          integer add:2 ld:1
Loop not vectorized: data dependency
Loop unrolled 2 times
```

Although it is not apparent in this code, a problem on X64 architectures with the compressed row structure is that generally the inner loop counts are small: for certain benchmark datasets, they range from two to twelve, and for typical problems only between 10-30. Adding overhead to vectorize the loop may be counter-productive. Since there is no software-pipeline via register renaming or other techniques on the X64 architecture, the only method to hide the latency involved in indirect addressing is to unroll the loop an abnormally high number of iterations, which can lead to other register allocation issues within a compiler, and may additionally lead to code bloat when the loop count is less than the unroll factor and the vectorized or unrolled loops are never taken.

As an experiment, we added directives or modified the code to assume that the values in the inds array were safe, i.e. unique in the innermost loop, and that the code could vectorize. We found that had no significant affect on performance. Since the indices and A matrix are very large, with no reuse within these nested loops, the performance of the function was bounded by the memory bandwidth required to load 12 bytes of data (8 from A_vals and 4 from inds) per each multiply-add.

For instance, from Diagram 2, if we find that our peak transfer rate of memory loads for a given architecture is 5% of peak FP performance, which is 8 GFlops, we can expect memory-bound loops with a compute intensity near 1 to run at roughly 8 GFlops * 5% = 400 MFlops. Note how much of a difference in FP performance can be gained or lost with a 1% increase or decrease in the values in Diagram 2!

Similar estimates can be made on other loops with proper assumptions about the location of the data being operated on. These assumptions can be based on either programmer knowledge, based on the size of the data, or obtained from real profile-generated hardware counter information.

To further fine-tune our estimates, consider the two loops from the CRS format above. We know from years of experience, and from the size of the data, that the performance is dominated by the time to load the A matrix and the index array. If the data is double precision, and the indices are ints, we can state that the compute intensity "from memory" is 2 FP ops per 1.5 memory loads, which is a value of 1.33. A peak transfer rate of %5 of peak fp performance corresponds to a compute intensity of 20 that is sustainable. So our peak FP performance is bounded by 8 Gflops * (1.33 / 20.0) or 533 MFlops. Again, we can see how much of a difference in FP performance can be gained or lost with a small increase or decrease in compute intensity.

The next storage class we looked at was the jagged diagonal or JDS. Using the JDS format, the matrix is stored in three arrays as follows:

$$\begin{aligned} values &= \left[11,21,32,41,52:12,22,33,44,55:14,25,34:15\right] \\ indices &= \left[0,0,1,0,1:1,1,2,3,4:3,4,3:4\right] \\ offsets &= \left[0,5,10,13,14\right] \end{aligned}$$

.

The values array again contains all nonzero matrix values, but now they are ordered so that the first non-zero value from each row is listed in order (a so-called jagged diagonal), followed by the second non-zero value from each row, and so on. Since the nonzero count per row is not uniform, the length of the jagged diagonals will become smaller, assuming that the rows are first permuted from most dense to least dense. The matrix shown here is already ordered this way. The indices array is similar to the CRS format, containing the column indices corresponding to the values array. The JDS format is certainly more complicated than CRS but has the important property that the jagged diagonals for realistic problems, and therefore the for-loop lengths in matrix operations such as SpMV, are typically of length proportional to the dimension of the matrix which will be of dimension thousands to millions. This is in stark contrast to CRS where the for-loop length is the number of non-zero values per row, which as mentioned above is typically a very small value and independent of matrix dimension.

The JadMatrix code for SpMV originally looked something like this:

```
if (!TransA) {
  for (int i=0; i<jaggedDiagonalLength; i++) {
    int ix = curIndices[i];
    int iy = RowPerm[i];
    double val = curValues[i];
    y[iy] += val*x[ix];
   }
} else {
  for (int i=0; i<jaggedDiagonalLength; i++) {
    int iy = curIndices[i];
    int ix = RowPerm[i];
    double val = curValues[i];
    y[iy] += val*x[ix];
   }
}
```

Note that the loop now runs the length of jaggedDiagonalLength, which can be up to a million points or more. But, also note that in the general case, it now requires two index arrays, which are also long, hence the actual total data requirements from memory has somewhat increased.

The epetra code contains optimized code for this format which computes on multiple right-hand-sides (RHS). The most-used loop is unrolled by 5 and looks something like this:

```
for (int i=0; i<jaggedDiagonalLength; i++) {
  int ix = curIndices[i];
  int iy = RowPerm[i];
  double val = curValues[i];
  y[iy] += val*x[ix];
  iy+=ldy; ix+=ldx;
  y[iy] += val*x[ix];
  iy+=ldy; ix+=ldx;
  y[iy] += val*x[ix];
  iy+=ldy; ix+=ldx;
  y[iy] += val*x[ix];
  iy+=ldy; ix+=ldx;
  y[iy] += val*x[ix];
}
```

The idea behind this was to get reuse on the matrix A and also on the index arrays. Which is a good idea and may help on many architectures. It turns out that it did not seem to help on the X64 architectures we tried. Probably this is due to the fact that ldx and ldy are larger than the L2 cache size. We are just adding multiple streams to and from memory, so the hardware and software prefetch mechanism is also less effective. And any possible reuse from a single column of y or x in the 1 RHS case is probably lost with 5 RHS.

One trick which became apparent with the JDS format was that for typical data sets, both the curIndices and RowPerm arrays contain long sequences of sequential (+1) accesses. We implemented an experiment to test the performance if we took advantage of this knowledge, which looked like this:

```
if (icnt > 0) {
  ilen = icnt;
  int ir = irun;
  int ix = curIndices[ir];
  int iy = RowPerm[ir];
  double * __restrict xx1 = x+ix;
  double * __restrict yy1 = y+iy;
  for (int i=0; i<ilen; i++) {
     double val = curValues[ir];
     yy1[i] += val*xx1[i];
     ir++;
   }
} else {
  ilen = -icnt;
  int ir = irun;
  for (int i=0; i<ilen; i++) {
     int ix = curIndices[ir];
     int iy = RowPerm[ir];
     double val = curValues[ir];
     y[iy] += val*x[ix];
     ir++;
   }
  }
```

So, when we were in a run, we reset the starting addresses for the x and y columns, and the innermost loop turns into a vector-multiply-add without any indirect addressing. Over the course of all processing in this routine, it turns out that almost 99% of the elements are handled in the first loop for a typical benchmark dataset.

While the results were overall favorable, the speedup achieved on AMD and Intel platforms was somewhat inconsistent. The largest speedup we saw was on a single threaded run on a Woodcrest with both this technique, and by reducing the default number of RHS to 1, it netted us a 1.76x speedup. More typical was around 1.2x.

While these were interesting, it is clear that instead of changing the solver code looking for patterns, a better approach is to choose a format that fits the opportunity. Thus we moved on to SDS.

Using the SDS format, the matrix is assumed to have a strongly diagonal pattern, where nonzero entries occur in only a few of the diagonals of the matrix. Many sparse matrices have this property at least locally, even if the underlying problem is globally unstructured. If a matrix does not have this property, then zeros may be inserted to allow diagonal storage, or diagonals that have just a few nonzero entries can be stored separately in a more general-purpose data format such as CRS. In our example, we will store all nonzero entries in SDS format. The arrays are as follows:

$$\begin{aligned}
&values[0] = [41,52] \\
&values[1] = [21,32,0,0] \\
&values[2] = [11,22,33,44,55] \\
&values[3] = [12,0,34,0] \\
&values[4] = [14,25] \\
&values[5] = [15] \\
&offsets \quad = [-3,-1,0,1,3,4]
\end{aligned}$$

The values arrays are used to store matrix entries diagonal-by-diagonal, with some zero fill for this example. The value of offsets[i] indicates which diagonal values[i] contains, where negative, zero and positive offset values refers to diagonals below, on and above the main diagonal, respectively. The advantage of the SDS format, if a matrix has strongly diagonal patterns is that there is no indices array and no indirect addressing is required for operations such as SpMV. This reduces memory bandwidth requirements, and memory references are done with unit stride, something that compilers and processors can often optimize via pre-fetching techniques. Also, as with JDS, for-loop lengths are proportional to matrix dimension.

Here is the loop structure to perform the SpMV operation given the SDS format:

```
for (int i=0; i<numDiags; i++) {
  curValues = ptr_to_diags[i];
  curDiagOffset = diagonal_offsets[i];
  if (curDiagOffset < 0)
     y = rvector-curDiagOffset;
  else
     y = rvector;
  if (curDiagOffset < 0)
     x = dvector;
  else
     x = dvector+curDiagOffset;
  diagLength = diagonal_lengths[i];
  for (int j=0; j<diagLength; j++) {
     y[j] += curValues[j] * x[j];
  }
}
```

As the code was originally written, the performance of this routine was poor. A complete CG solver, on a 100 x 100 x 100 problem, running fully subscribed on 4 cores, was performing at only 128 MFlops per core, compared to 271 MFlops for the same problem running on CRS. The first thing we noticed was that the code, although now simplified, was not vectorizing. The first step in tuning C/C++ code should always be to make appropriate use of the restrict type qualifiers. In C++, they take this form:

```
double * __restrict curValues = 0;
double * __restrict y = 0;
double * __restrict x = 0;
```

You can verify that code vectorizes by using the –Minfo switch on PGI compilers:

```
148, Generated 3 alternate loops for the inner loop
     Generated vector sse code for inner loop
     Generated 3 prefetch instructions for this loop
     Generated vector sse code for inner loop
     Generated 3 prefetch instructions for this loop
     Generated vector sse code for inner loop
     Generated 3 prefetch instructions for this loop
     Generated vector sse code for inner loop
     Generated 3 prefetch instructions for this loop
```

Alternate loops, or "altcode", is generated by the PGI compiler, based on array alignment and loop lengths which are executed at runtime. It is controllable with the -Mvect=altcode switch. Vectorizing this loop gave us a tiny bump in performance, but not what was expected.

A benefit of removing the indirect addressing in the SDS format was it became clear what the next step in optimization should be. The inner loop count is long, longer than the size of the cache, yet in the outer loop, the y and x vectors basically start over again at or near the same point each time. This lends itself to strip-mining. We added an extra loop to the processing, to work in smaller, less-than-L2-cache-sized strips of y and x. It looks something like this:

```
#define STRIPVAL 16384
  for (int k=0; k<maxDiagLength; k+=STRIPVAL) {
    for (int i=0; i<numDiags; i++) {
      curValues = ptr_to_diags[i];
      curDiagOffset = diagonal_offsets[i];
      y = …;
      x = …;
      diagLength = diagonal_lengths[i];
      curValues += k;
      y += k;
      x += k;
      diagLength -= k;
      if (diagLength > STRIPVAL)
        diagLength = STRIPVAL;
      for (int j=0; j<diagLength; j++) {
         y[j] += curValues[j] * x[j];
      }
    }
  }
```

After this change, we started to see some movement in the performance results. The GNU g++ performance was up to 349 MFlops per core. However, the PGI performance was still only 146 MFlops per core. Why?

There is a hint about the PGI performance in the Minfo output: prefetching. With the strip-mining modifications, the y and x vectors are resident in cache. Prefetching data that is in cache can be costly (sometimes, as in this case, extremely costly). When we added the Mnoprefetch flag to the PGI compile line, the performance was recovered, at 352 MFlops, roughly equivalent to the g++ performance.

The memory tuning was still not optimal. We would actually like to prefetch the A matrix values, but not prefetch the y and x vectors. This has been possible with previous versions of PGI compilers, but we are making it easier in version 7.2. This is now the recommended pragma form:

```
      for (int j=0; j<diagLength; j++) {
   #pragma mem prefetch curValues[j+8]
         y[j] += curValues[j] * x[j];
      }
```

which specifies a target and a distance. The prefetch pragma will override the default prefetch rules on any target processor. With this final change, the performance increased to 395 MFlops total, per core, and 460 MFlops in this SpMV kernel (Table 1b). When run on a single core of a quad core socket, the performance was 607 and 675 MFlops, respectively (Table 1a). Since 395 * 4, for each of the four cores, is 1.58 GFlops, this certainly makes a case for running even "memory bandwidth limited" codes fully subscribed, rather than on one or two cores of the processor, as long as they have been tuned.

Finally, getting back to our transfer rate vs. peak floating point performance graphs, for the fully subscribed Barcelona, we measured a memory load transfer rate of 3.7% of peak, and could sustain a compute intensity of 27. The compute intensity from memory of the SDS loop, assuming y and x are in cache, is 2. So, 8 GFlops * (2.0 / 27.0) = 593 MFlops. We've attained 77.6% (460/593) of our memory limiting peak performance, and the rest is attributed to the L2 load and store transfer rates which must be added at the compute intensity level of (2 FP Ops) / (2 FP loads from L2 + 1 FP Store to L2) = 0.667. The actual combinatorial method is left as an exercise for the reader.

4. Results

Table 1a and 1b contain performance data on a 2.0 GHz AMD Barcelona, running on 1 core of a Quad-Core, and also running fully subscribed. Four datasets are shown: the largest results in a matrix dimension of one million. The smallest shows signs of fitting into the L2 cache on some systems. Both the CRS and SDS data formats are used, and values are in MFlops for the total solve, not just SpMV.

Compared to the original CRS implementation, by implementing and tuning the SDS approach, we were able to improve performance by between 44% and 91% on a fully subscribed Quad-Core.

Problem Size	24**3	48**3	72**3	100**3
g++ original code, CRS	512	486	466	469
g++ original code, SDS	512	350	267	285
pgCC original code, CRS	487	461	451	430
pgCC original code, SDS	541	360	279	292
pgCC SDS, + restrict	389	343	334	326
g++, SDS, restrict + Strip-mine	540	486	482	493
pgCC SDS, restrict + Strip-mine	389	341	336	346
pgCC same, +Mnoprefetch	572	459	465	481
pgCC, same + prefetch pragma	695	632	563	607

Table 1a. Performance of a single core running on an otherwise idle socket of an AMD Quad-Core processor. Values are in MFlops for the entire solver.

Problem Size	24**3	48**3	72**3	100**3
g++ original code, CRS	314	275	270	271
g++ original code, SDS	442	125	125	128
pgCC original code, CRS	291	271	263	264
pgCC original code, SDS	438	125	126	129
pgCC SDS, + restrict	177	149	135	133
g++, SDS, restrict + Strip-mine	469	353	349	349
pgCC SDS, restrict + Strip-mine	172	148	145	146
pgCC same, +Mnoprefetch	476	351	351	352
pgCC, same + prefetch pragma	599	422	388	395

Table 1b. Performance of each core of a fully subscribed AMD Quad-Core processor. Values are in MFlops for the entire solver.

5. Conclusions

Although the Quad-Core processors from AMD and Intel have been out for some time, we are still learning how best to take advantage of the characteristics of hardware resources available. Working on Trilinos and other codes, a few general guidelines have become apparent:

Every fully-subscribed code is likely memory bandwidth limited. The available memory bandwidth, as a percentage of peak floating point performance, has been steadily declining since the X64 architecture was introduced.

Tuning for data movement optimizations can be far more important than tuning for vectorization, though many optimizations may only kick in most effectively when the code is vectorized. Also, vectorization, as we have stated in previous papers, is the key to obtaining a large fraction of peak performance when there is good reuse of data in cache.

PGI offers a number of switches and pragmas for fine-grained control of data movement optimizations. Note that the compiler can't always determine, by itself, whether data is likely resident in the caches. There is a performance penalty for prefetching data already in cache, which was discussed, and there is also a performance penalty for using non-temporal stores to data that is in cache (not discussed in this paper).

Limit the memory bandwidth requirements if possible. This probably requires code restructuring, such as the strip-mining method we applied. Take advantage of the caches. It seems that if you can limit the number of "streams" of data to and from memory in each loop, the hardware and software prefetching mechanisms run more efficiently, especially when the socket is fully subscribed.

6. Future Work

PGI will continue to explore and improve the tracking and presentation of compute intensity and a related metric, transferability. Whether to provide hints to the developer for memory tuning, as we have shown here, for auto-parallelization, or for farming work off to an attached GPU, we are working to carry this technology forward.

We will continue to examine the application programming and compiler techniques required to support larger vector operations that are most likely to appear in future sequential and parallel architectures deployed by the commercial processor vendors.

Results derived from this work will benefit users of Trilinos on current and future DOE and DOD computing resources. The coding practices and compiler enhancements that we outline will also help other programmers and users in applying the same techniques and tools to their codes. The improved performance we demonstrate may also be beneficial in scheduling resource utilization and in planning future compute platform acquisitions. Finally, the cooperation between Sandia's Scalable Architectures Group and the Portland Group product development and support teams is shown to be a model for future joint endeavours.

About the Authors

Brent Leback is the Applications Engineering manager for PGI. He has worked in various positions over the last 20 years in HPC customer support, math library development, applications engineering and consulting at QTC, Axian, PGI and STMicroelectronics. He can be reached by e-mail at brent.leback@pgroup.com.

Douglas Doerfler is a Principal Member of Technical Staff at Sandia National Laboratories. He started his career at Sandia 22 years ago in the field of analog instrumentation, then migrated to embedded computing, then to embedded high performance computing, and that lead to an interest in supercomputing. His current job interests include scalable computational architectures research and supercomputing performance analysis. He can be reached by e-mail at *dwdoerf@sandia.gov*.

Michael Heroux is a Distinguished Member of Technical Staff at Sandia national Laboratories. He leads the Trilinos libraries project and the Mantevo performance modelling project. Prior to joining Sandia, he worked for 10 years at Cray Research and Silicon Graphics in the math libraries and engineering applications groups. He can be reached by email at maherou@sandia.gov.

References

1. Doerfler, Hensinger, Miles, and Leback, *Tuning C++ Applications for the Latest Generation x64 Processors with PGI Compilers and Tools*, CUG 2007 Proceedings
2. *Intel News Fact Sheet*, http://www.intel.com/pressroom/archive/reference/IntelMulticore_factsheet.pdf
3. *AMD Developer Central*, http://developer.amd.com/cpu/SSE5/Pages/default.aspx
4. Roger W. Hockney, *The Science of Computer Benchmarking*, Society for Industrial and Applied Mathematics, 1996.
5. Hensinger, Luchini, Frigo and Strumpen, *Performance Characteristics of Cache Oblivious Implementation Strategies for Hyperbolic Equations on Opteron Based Super Computers*, CUG 2006 Proceedings.
6. M.A.Heroux, *Epetra Home Page*, http://trilinos.sandia.gov/packages/epetra, 2004
7. K.D. Devine, B.A.Hendrickson, E.G. Boman, M. M. St. John and C. Vaughn, *Zoltan: A dynamic load-balancing library for parallel applications – User's Guide,* Technical Report, Sandia National Laboratories, SAND99-1377, 1999
8. M. Heroux, *Mantevo Home Page*,http://software.sandia.gov/mantevo, April 2008.
9. Frigo, Leiserson, Prokop and Ramachandran, *Cache Oblivious Algorithms*, Proc. 40th Annual Symp. Foundations of Computer Science (FOCS'99).
10. Frigo and Strumpen, *Cache Oblivious Stencil Computations*, Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05).
11. D. Miles, B. Leback and D. Norton, *Optimizing Application Performance on Cray Systems with PGI Compilers and Tools*, CUG 2006 Proceedings.
12. Michael Wolfe, *High Performance Compilers for Parallel Computers,* Addison-Wesley, 1996.