Optimizing SciMark* 2.0 Using Intel® Software Products

By Henry Gabb and Chirag G. Shah,
Intel Corporation

November, 2005

Introduction

SciMark* 2.0 is a floating-point benchmark from the National Institute of Standards and Technology. It consists of five computational kernels:

1. FFT – performs a complex 1D fast Fourier transform
2. SOR – solves the Laplace equation in 2D by successive over-relaxation
3. MC – computes π by Monte Carlo integration
4. MV – performs sparse matrix-vector multiplication
5. LU – computes the LU factorization of a dense N x N matrix

These kernels represent the types of calculations that commonly occur in numerically-intensive scientific applications.

SciMark 2.0 is widely-used to measure CPU performance. As such, this benchmark is important to Intel. Each kernel except MC has small and large problem sizes (Table 1). The small problems are designed to test raw CPU performance and the effectiveness of the cache hierarchy. The large problems stress the memory subsystem because they do not fit in cache. The MC kernel only uses scalars so there is no distinction between the small and large problems.

Benchmark	Problem Size	
	Small	Large
FFT	N = 1024	N = 1048576
SOR	100 x 100	1000 x 1000
MV	N = 1000, NZ = 5000	N = 100000, NZ = 1000000
LU	100 x 100	1000 x 1000

Table 1. Small and large problem sizes for the SciMark 2.0 kernels. Note that there is no distinction between small and large problem sizes for the MC kernel.

In order to measure the success of a software tuning project, it is necessary to generate baseline performance data. Both Java* and ANSI C versions of SciMark 2.0 are available. The ANSI C version was used for the present study. Unless otherwise noted, the following dual-processor server was used for all performance measurements:

Hardware	
CPU	Intel® Xeon® Processor (3.6 GHz Xeon 2 MB L2 cache) with Intel® EM64T
Motherboard	Intel Server Board SE7520AF2
Memory	512 MB DDR2
BIOS	
Version	P06
Adjacent Cache Line Prefetch	ON
Hardware Prefetch	ON
Hyper-Threading Technology	OFF
Software	
Operating system	Red Hat Enterprise Linux* AS 3
Linux Kernel	2.4.21-20.EL #1 SMP
Intel® C++ Compiler for Linux	8.1 (l_cce_pc_8.1.024)
Intel® Cluster Math Kernel Library	7.2 (l_cluster_mkl_7.2.008)
GNU* C compiler	gcc* 3.2.3

As the GNU C compiler is readily available on Linux, `gcc` was used to set the performance baseline. Results are reported in millions of floating-point operations per second (MFLOPS). Thus, higher values indicate better performance. Although exact performance is reported, the SciMark 2.0 FAQ states that actual performance may vary by ±5%. However, the observed performance variations were generally much smaller than 5%.

Benchmark	Performance (MFLOPS)	
	Small	Large
FFT	149	36
SOR	401	393
MC	47	47
MV	213	211
LU	297	292
Composite score	**221**	**196**

Table 2. The GNU C compiler was used to generate preliminary performance data for SciMark 2.0 compiled at the default optimization level.

Table 2 displays SciMark 2.0 performance using the GNU C compiler at default optimization. However, the GNU C compiler is capable of more advanced optimization so additional compiler options (-O3 -march=nocona -ffast-math -mfpmath=sse) were used to set a more realistic performance baseline. Table 3 shows the best SciMark 2.0 performance achieved with gcc during this study. These results will be used as the performance baseline to measure tuning effectiveness.

Benchmark	Performance (MFLOPS)	
	Small	Large
FFT	510	45
SOR	524	495
MC	206	206
MV	857	453
LU	884	392
Composite score	**596**	**318**

Table 3. The GNU C compiler with aggressive optimization was used to generate baseline performance data. SciMark 2.0 was built with the following compiler options: `-O3 -march=nocona -ffast-math -mfpmath=sse`.

Using the Intel C++ Compiler for Linux to Improve Performance

It is standard practice to submit results for unmodified benchmarks in order to make direct comparisons between results and to keep an even playing field between competitors. Therefore, before examining source code modifications that could potentially improve performance, SciMark 2.0 was simply recompiled with the Intel C++ Compiler for Linux. The Intel compilers are designed to take maximum advantage of Intel processor architecture so some performance improvement is expected over the GNU compiler, which is designed for generality rather than maximum performance. As such, recompiling SciMark 2.0 with the Intel compiler at default optimization improves the benchmark composite score relative to the GNU baseline as shown in Table 4. Similar results were obtained for the large problem sizes, as shown in Table 5.

Benchmark	Performance (MFLOPS) for Small Problems		
	GNU baseline	Intel baseline	Speedup
FFT	510	512	1.0
SOR	524	759	1.4
MC	206	153	0.7
MV	857	883	1.0
LU	884	1282	1.4
Composite score	**596**	**718**	**1.2**

Table 4. The Intel C++ Compiler for Linux at the default optimization level improves the benchmark composite score relative to the GNU baseline for the small problem sizes.

Benchmark	Performance (MFLOPS) for Large Problems		
	GNU baseline	Intel baseline	Speedup
FFT	45	45	1.0
SOR	495	720	1.5
MC	206	153	0.7
MV	453	455	1.0
LU	392	402	1.0
Composite score	**318**	**355**	**1.1**

Table 5. The Intel C++ Compiler for Linux at the default optimization level improves the benchmark composite score relative to the GNU baseline for the large problem sizes.

Though default optimization yields good results, more aggressive optimization can improve performance even further. The -fast option encompasses several common compiler optimizations: `-O3, -xP, -ipo. The -O3` option enables the default optimizations plus other optimizations designed to improve the performance of floating-point-intensive loops. The Intel compilers also have options to optimize for specific Intel processor families. The `-xP` flag tells the compiler to generate a binary tuned specifically for Intel processors that support Streaming SIMD Extensions 3 (SSE3) instructions. The `-ipo` option enables interprocedural optimizations such as inline function expansion.

By default, the Intel compilers conservatively assume that some memory locations are aliased and are referenced by more than one variable. This dependency prevents the compiler from performing some optimizations. The SciMark 2.0 source code does not contain aliased memory locations so the `-fno-alias` flag was also used. (It is worth noting that improper use of the `-fno-alias` flag can result in incorrect calculations.)

Aggressive optimization with the Intel C++ compiler resulted in significantly greater performance relative to the GNU baseline for both the small, as shown in Table 6, and the large problem sizes, as shown in Table 7. The Intel compiler doubles the performance of some kernels.

Benchmark	Performance (MFLOPS) for Small Problems		
	GNU baseline	Intel optimized	Speedup
FFT	510	521	1.0
SOR	524	1092	2.1
MC	206	447	2.2
MV	857	832	1.0
LU	884	1827	2.1
Composite score	**596**	**943**	**1.6**

Table 6. The Intel C++ Compiler for Linux using the "`-fast -fno-alias`" optimization flags significantly improves the benchmark composite score relative to the GNU baseline for the small problem sizes. Performance is also significantly improved for three of the five kernels.

Benchmark	Performance (MFLOPS) for Large Problems		
	GNU baseline	Intel compiler	Speedup
FFT	45	45	1.0
SOR	495	1015	2.1
MC	206	447	2.2
MV	453	457	1.0
LU	392	389	1.0
Composite score	**318**	**389**	**1.2**

Table 7. The Intel C++ Compiler for Linux using the "`-fast -fno-alias`" optimization flags further improves the benchmark composite score relative to the GNU baseline for the large problem sizes.

Performance is also significantly improved for two of the five kernels.

Using the Intel Math Kernel Library (MKL) to Improve Performance

Most benchmark code is designed for portability rather than maximum performance on a particular platform. However, optimized numerical libraries are readily available for most platforms. It is standard practice to use numerical libraries for common mathematical operations. For example, the Basic Linear Algebra Subprograms (BLAS) and the Linear Algebra Package (LAPACK) provide standard interfaces to many linear algebra functions. Fourier transform libraries are also available.

Intel® Math Kernel Library (Intel® MKL) contains highly optimized BLAS, LAPACK, and Fourier transform implementations. It also contains two vector libraries for random number generation and transcendental functions. These capabilities are directly applicable to the SciMark 2.0 FFT, MC, MV, and LU kernels. Many Intel MKL functions are also threaded to take advantage of multiprocessor systems like the one used in this study. The compiler-level tuning described in the previous section did not modify the original benchmark source code. That constraint is now removed to accommodate MKL.

Tuning the FFT Kernel with the Intel MKL Discrete Fourier Transform API

The fast Fourier transform is a well-studied algorithm that is used in a wide variety of applications. Developers can often choose from several off-the-shelf discrete Fourier transform (DFT) libraries. Unlike linear algebra, however, there are no standard calling conventions for DFT functions. Replacing the hand-coded transform in the SciMark 2.0 FFT kernel therefore requires some code modification in order to use Intel MKL.

Rather than attempting to provide a unique function for every DFT permutation, Intel MKL uses a general-purpose API. Developers describe the desired transform to Intel MKL before initiating the computation. For example, the SciMark 2.0 FFT kernel computes a complex 1D forward-backward transform on an array of 1024 numbers. With Intel MKL, the developer creates a descriptor of this transform as shown in the following code.

```
#include <mkl.h>

int N = 1024;
long status;
double *x = RandomVector ((2 * N), R);
double scale = 1.0 / (double)N;
DFTI_DESCRIPTOR *dftiHandle;

status = DftiCreateDescriptor (&dftiHandle, DFTI_DOUBLE, DFTI_COMPLEX, 1, N);
status = DftiSetValue (dftiHandle, DFTI_BACKWARD_SCALE, scale);
status = DftiCommitDescriptor (dftiHandle);

status = DftiComputeForward (dftiHandle, x);
status = DftiComputeBackward (dftiHandle, x);

status = DftiFreeDescriptor (&dftiHandle);
```

The call to `DftiCreateDescriptor` allocates and initializes a descriptor for a double precision, complex 1D DFT of size N = 1024. (Note that the transform array is dimensioned to 2048 to accommodate the real and imaginary parts of each complex number.) `DftiSetValue` applies the specified scaling factor to the backward transform. `DftiCommitDescriptor` does what its name implies. `dftiHandle` can now be used to transform any array that is consistent with the descriptor. `DftiComputeForward` and `DftiComputeBackward` perform the actual forward and backward transforms on the array using the supplied descriptor. Descriptors can be reused, but when they are no longer needed, `DftiFreeDescriptor` frees the memory allocated to hold the descriptor. This versatile API allows developers to describe and compute a wide variety of DFT's. It is also much easier than trying to remember the name and prototype of numerous functions that compute a specific type of DFT.

Replacing the hand-coded DFT in the SciMark 2.0 FFT kernel with MKL gives a significant speedup over baseline performance (Table 8). It is worth noting that on multiprocessor systems, Intel MKL automatically computes multidimensional DFT's in parallel.

FFT Benchmark	Performance (MFLOPS)		
	GNU baseline	MKL	Speedup
Small problem (N = 1024)	510	1817	3.6
Large problem (N = 1048576)	45	600	13.3

Table 8. Replacing the hand-coded transform in the SciMark 2.0 FFT kernel with Intel MKL significantly improves performance.

Tuning the LU Kernel with the Intel MKL LAPACK Implementation

Like the Fourier transform, LU factorization is another common mathematical operation. The MKL LAPACK implementation contains a suitable function for the SciMark 2.0 LU kernel. Specifically, the `dgetrf` function computes the LU factorization of a general, double precision M x N matrix. Replacing the hand-coded LU factorization function in SciMark 2.0 with `dgetrf` requires attention to some important details because SciMark 2.0 is written in C whereas LAPACK only defines a Fortran interface.

```
Syntax
call dgetrf (m, n, A, lda, ipiv, info)

Input Parameters
m         INTEGER                     Number of rows in matrix A (m >= 0)
n         INTEGER                     Number of columns in matrix A (n >= 0)
A         DOUBLE PRECISION            Input matrix
          DIMENSION (lda,*)
lda       INTEGER                     First dimension of A

Output Parameters
A                                     Overwritten by L and U
ipiv      INTEGER                     Pivot indices
          DIMENSION MAX(1, MIN(m,n))
info      INTEGER                     Error code
```

Fortran is call-by-reference while C is call-by-value. The following function call adheres to Fortran calling conventions:

```
dgetrf (&N, &N, A, &N, pivot, &error);
```

Arrays in C are often allocated as pointers-to-pointers, which are not necessarily contiguous in memory. Much better performance is possible with contiguous data so the C interface to `dgetrf` expects a vector. Finally, Fortran uses column-major ordering for arrays but C uses row-major ordering. The data supplied to `dgetrf` must be in column-major order to get correct results.

This attention to detail pays off because the Intel MKL LU factorization significantly improved performance for the large problem, as shown in Table 9. Parallelism is an added bonus that improves performance even further. Though Intel MKL performance for the small problem is good, it is slightly worse than the best performance achieved by the Intel compiler. A problem this small does not merit the LAPACK overhead. Similarly, a 100 x 100 LU factorization is too small to benefit from multithreading.

LU Benchmark	Performance (MFLOPS)				
	GNU baseline	MKL		Speedup	
		1-thread	2-threads	1-thread	2-threads
Small problem	884	1680	N/A	1.9	N/A
Large problem	392	3837	6646	9.8	16.9

Table 9. Replacing the hand-coded LU factorization in the SciMark 2.0 LU kernel with Intel MKL significantly improves performance for the large problem.

Note: multithreading does not help the small problem because there is not enough work in a 100 x 100 LU factorization to merit thread creation.

Parallelizing the LU Kernel for a Cluster

In addition to thread-level parallelism, there is yet another advantage to using the Intel MKL LAPACK implementation. Namely, Intel Cluster MKL supports the distributed-memory parallel version of LAPACK called ScaLAPACK (Scalable LAPACK). Even though the large LU problem in SciMark 2.0 is far too small to merit solution on a cluster, a ScaLAPACK implementation is presented here to show that with a few adjustments, the Intel MKL version of the LU kernel can solve much larger problems. These adjustments can be summarized in four steps:

1. Initialize the process grid
2. Create a descriptor for each matrix that will be distributed across the process grid
3. Replace the call to `dgetrf` with `pdgetrf` (the 'p' is for parallel)
4. Release the process grid.

The ScaLAPACK SL_INIT routine creates a virtual process grid on which the computation will be performed. This also allows each process to find its location in the process grid and determine its sub-domain of the global computation. ScaLAPACK scales well because both data and work are divided across multiple processes, which run in parallel on different processors.

The developer creates a descriptor for each matrix involved in the computation. The descriptor is simply an array that contains such information as the matrix type (e.g., dense or banded), the handle for the process grid, the total number of rows and columns in the global matrix, the blocking factor (discussed below), the process holding the first row and column of the global matrix, and the leading dimension of the local sub-matrix. These descriptors determine how Intel Cluster MKL distributes across the process grid. Only one matrix descriptor is needed for LU factorization.

In ScaLAPACK, matrices are distributed in block-cyclic fashion (Figure 1). The block size specified in the matrix descriptor greatly affects overall parallel performance. There are no hard-and-fast rules for setting block size but, in general, a large block size minimizes communication overhead at the expense of load balance. Conversely, a small block size improves load balance but increases communication between the processes. This tradeoff is illustrated in Figure 1 for a lower-triangular matrix. In the leftmost diagram in Figure 1, notice that process-0 has significantly more of the matrix than process-3. The rightmost 2D block-cyclic distribution is a definite improvement over a simple 1D block distribution. However, communication is required along each process boundary. Comparing the two 2D block-cyclic distributions shown in Figure 1, it is clear that the rightmost diagram has better load balance but higher communication overhead.

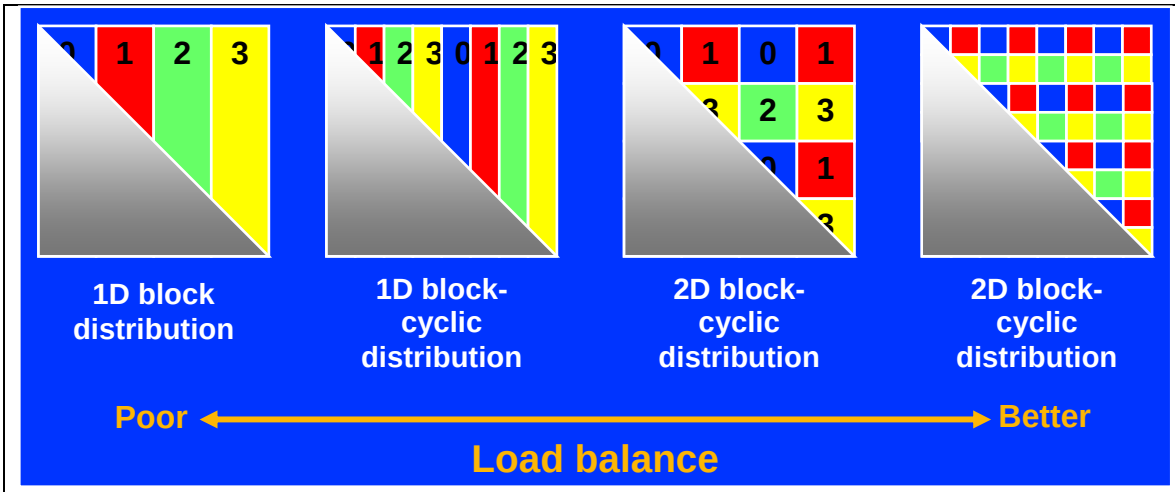

Figure 1: ScaLAPACK uses block-cyclic data distribution to improve load balance.

Once the global matrix is properly distributed, the work can be distributed by replacing the call to `dgetrf` (described in the previous section) with a call to its parallel counterpart:

```
pdgetrf_ (&global_rows,    // Number of rows in the global matrix
          &global_cols,    // Number of columns in the global matrix
          A,               // Local sub-domain of global matrix
          &one,
          &one,
          descA,           // Descriptor for global matrix
          pivot,
          &error);
```

Comments highlight the key arguments that differentiate `pdgetrf` from `dgetrf`. It is important to note that the LU matrix is now spread across multiple processes. The matrix argument supplied to `pdgetrf` is actually a sub-matrix local to the calling process. Instead of allocating memory for the entire LU matrix, the processes need only allocate enough memory for their respective sub-domains. ScaLAPACK can harness the memory of an entire cluster to solve linear algebra problems too large for a single computer.

Intel Cluster MKL makes it possible to perform the LU factorization on a 40000 x 40000 matrix using a small cluster [8 dual-3.0 GHz Intel Xeon processors (512 KB L2 cache nodes and 2 GB memory per node) connected via Gigabit Ethernet and InfiniBand*] in a few minutes (approximately 42000 MFLOPS using Gigabit Ethernet and 46000 MFLOPS using InfiniBand). The Intel MPI Library 1.0 was used as the underlying communication layer because it is fabric-independent. A double precision array of this size requires approximately 12 GB memory. Few computers have this much memory so a slow, out-of-core solution would normally be required to solve a problem this large. If more compute nodes, and hence, more memory, are added to the cluster, it is possible to solve even larger problems.

The previous discussion omits many details for the sake of brevity and only hints at the power and flexibility of ScaLAPACK. The reader is referred to the ScaLAPACK User's Guide and the Intel Math Kernel Library – Reference Manual for more detailed examples of solving large linear algebra problems.

Tuning the MC Kernel with the MKL Vector Statistical Library and OpenMP

Simulation methods and stochastic algorithms require a set of randomly determined initial conditions or a continuous stream of random numbers. Because random number generation is such a common feature in computational methods, the Intel MKL Vector Statistical Library (VSL) provides optimized random number generators for common probability distributions: uniform, Gaussian, exponential, Poisson, etc. The basic random number generators can be used to generate non-uniform distributions. Users can also register their own random number generators with VSL.

VSL functions return vectors of random numbers because algorithms that use random number generation usually need many random numbers instead of just one. Also, vector functions give better performance than scalar random number generators.

The SciMark 2.0 MC kernel calculates π by randomly sampling points in a unit square and determining whether these points fall within the upper-right quadrant of a unit circle inscribing the square:

```
double MonteCarlo_integrate (int Num_samples)
{
   int under_curve = 0;
   int count;

   Random R = new_Random_seed (SEED);
   for (count = 0; count < Num_samples; count++)
   {
       double x = Random_nextDouble ®;
       double y = Random_nextDouble ®;

       if (x*x + y*y <= 1.0) under_curve++;
   }
   Random_delete ®;

   return ((double) under_curve / Num_samples) * 4.0;
}
```

The following steps are used to implement VSL in the SciMark 2.0 MC kernel:

1. For efficiency, VSL provides a vector of random numbers. Creating a vector for the entire stream is impractical because `Num_samples` may be too large. Therefore, a static array of defined size is used to hold blocks of the random number stream.
2. A random number stream of type `VSLStreamStatePtr` is initialized with a call to `vslNewStream`, specifying which basic random number generator and seed to use.
3. Next, the call to `vdRngUniform` puts a uniform distribution of `(2 * BLOCK_SIZE)` double precision random numbers of range [0.0 to 1.0] into the array `rnBuf`.
4. The random numbers are used in the π calculation.
5. The call to `vslDeleteStream` deletes the random number stream when it is no longer needed.

```
#include <mkl.h>

double MonteCarlo_integrate (int Num_samples)
{
   int under_curve = 0;

   int i, j, blocks, tail;
   static double rnBuf[2 * BLOCK_SIZE];
   double rnX, rnY;
   VSLStreamStatePtr stream;

   blocks = Num_samples / BLOCK_SIZE;
   tail = Num_samples - blocks * BLOCK_SIZE;

   vslNewStream (&stream, VSL_BRNG_MCG31, SEED);

   for (i = 0; i < blocks; i++)
   {
      vdRngUniform (VSL_METHOD_DUNIFORM_STD, stream,
                    (2 * BLOCK_SIZE), rnBuf, 0.0, 1.0);

      for (j = 0; j < BLOCK_SIZE; j++)
      {
         rnX = rnBuf[2*j];
         rnY = rnBuf[2*j+1];
         if (rnX*rnX + rnY*rnY <= 1.0) under_curve++;
      }
   }

   vdRngUniform (VSL_METHOD_DUNIFORM_STD, stream, (2 * tail), rnBuf, 0.0, 1.0);

   for (j = 0; j < tail; j++)
   {
      rnX = rnBuf[2*j];
      rnY = rnBuf[2*j+1];
      if (rnX*rnX + rnY*rnY <= 1.0) under_curve++;
   }

   vslDeleteStream (&stream);
   return ((double) under_curve / Num_samples) * 4.0;
}
```

The underlying MC algorithm has not been changed in the VSL implementation. Scalar random number generation has simply been replaced by a vector approach, resulting in a significant performance improvement, as shown in Table 10.

The MC algorithm is naturally parallel. Each random sample is independent of every other sample. VSL functions are threadsafe so they can be used for parallel random number generation. After the initial VSL implementation, OpenMP is used to parallelize the MC kernel. OpenMP is a portable standard that it is easy-to-use and supported by the Intel compilers. The VSL/OpenMP version of the SciMark 2.0 MC kernel is shown below:

```
nThreads = maxThreads = omp_get_max_threads ();
omp_set_num_threads (nThreads);

vslNewStream (&streamX, VSL_BRNG_MCG31, SEED);
vslCopyStream (&streamY, streamX);
vslLeapfrogStream (streamX, 0, 2);
vslLeapfrogStream (streamY, 1, 2);

streamXThread = (VSLStreamStatePtr) malloc (nThreads *
                                            sizeof (VSLStreamStatePtr));
streamYThread = (VSLStreamStatePtr) malloc (nThreads *
                                            sizeof (VSLStreamStatePtr));

for (i = 0; i < nThreads; i++)
{
   vslCopyStream (&(streamXThread[i]), streamX);
   vslCopyStream (&(streamYThread[i]), streamY);
   vslSkipAheadStream (streamX, BLOCK_SIZE);
   vslSkipAheadStream (streamY, BLOCK_SIZE);
}

#pragma omp parallel for \
     reduction (+:under_curve) \
     private (i, j, rnX, rnY, threadID, thread_under_curve, rnBufX, rnBufY)
for (i = 0; i < blocks; i++)
{
   threadID = omp_get_thread_num ();

   vdRngUniform (VSL_METHOD_DUNIFORM_STD, streamXThread[threadID],
                 BLOCK_SIZE, rnBufX, 0.0, 1.0);
   vdRngUniform (VSL_METHOD_DUNIFORM_STD, streamYThread[threadID],
                 BLOCK_SIZE, rnBufY, 0.0, 1.0);

   thread_under_curve = 0.0;

   for (j = 0; j < BLOCK_SIZE; j++)
   {
      rnX = rnBufX[j];
      rnY = rnBufY[j];
      if (rnX*rnX + rnY*rnY <= 1.0) thread_under_curve++;
   }
   under_curve += thread_under_curve;

   #pragma omp critical
   {
      vslCopyStreamState (streamXThread[threadID], streamX);
      vslCopyStreamState (streamYThread[threadID], streamY);
      vslSkipAheadStream (streamX, BLOCK_SIZE);
      vslSkipAheadStream (streamY, BLOCK_SIZE);
   }   // End OpenMP critical section
}   // End OpenMP parallel loop
```

Additional code modifications and VSL functions are needed to generate a reproducible random number stream. First, each thread creates its own copy of the random number stream with `vslCopyStream`. The call to `vslSkipAheadStream` ensures that their random number sequences do no overlap. The OpenMP "`parallel for`" pragma creates threads and executes the next for-loop in parallel. The OpenMP `private` clause creates thread-private copies of the specified variables. The OpenMP "`reduction (+:under_curve)`" clause creates a private copy of `under_curve` for each thread and then sums the values from each thread at the end of the parallel computation.

There is a certain amount of system overhead associated with multithreading. Before creating threads, the programmer must ask, “Does the amount of computation merit thread creation?” For example, the trailing loop is too small to bother parallelizing. Most of the work is in the leading loop. The MC kernel steadily increases the number of random samples used to calculate π until the timer resolution threshold is reached. To avoid tripping this threshold when the number of samples is too small to effectively use multiple threads, the number of samples is set to 268,435,456 for the OpenMP tests. This is the value at which the serial VSL implementation reaches the timer resolution threshold on the test system. Taking advantage of both processors in the test system improves MC performance, as shown below in Table 10.

MC Benchmark	Performance (MFLOPS)	Speedup
GNU baseline	206	
Intel compiler only	447	2.2
Intel compiler + VSL	699	3.4
Intel compiler + VSL + OpenMP	1003	4.9

Table 10. Replacing the scalar random number generator with a vector random number generator in the Intel MKL VSL and employing multiple threads improves performance of the SciMark 2.0 MC kernel relative to the GNU baseline.

Tuning the MV Kernel Using the Intel MKL Sparse BLAS Capability

Matrix-vector multiplication is such a common operation that it is defined in BLAS. For example, the `dgemv` function performs a double precision, matrix-vector product. Standard BLAS functions are designed with dense matrices in mind but the SciMark 2.0 MV kernel uses a sparse matrix, in which the majority of elements are zeros. One could treat a sparse matrix as a dense matrix and simply use a standard BLAS routine, but this is wasteful in terms of storage and computation. The large MV problem uses a 100,000 x 100,000 double precision matrix with only 1,000,000 nonzero elements. A dense matrix representation would require approximately 75 GB of memory.

Sparse matrices are common in technical computing so many compressed storage schemes have been devised. These schemes typically store only the nonzero matrix elements and their locations in the original 2D context. The SciMark 2.0 MV kernel uses a compressed sparse row (CSR) format consisting of three arrays: one to store the nonzero values, one to store the column index of each nonzero value, and one containing the index into the values array for the first nonzero element of each row. The following example from the MKL documentation illustrates the CSR format.

$$\begin{vmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{5}{8} & 0 \\ 0 & 0 & 0 & 0 & 16 \end{vmatrix}$$

$$values = \left(9 \quad \frac{3}{2} \quad 6 \quad \frac{3}{4} \quad 3 \quad \frac{1}{2} \quad \frac{1}{2} \quad \frac{5}{8} \quad 16 \right)$$

$$columns = (1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 2 \quad 3 \quad 4 \quad 5)$$

$$rowIndex = (1 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10)$$

The size of the values and column index arrays is equal to the number of nonzero elements. The size of the row index array is equal to the number of rows plus one. Therefore, the sparse representation of the MV large matrix requires only 16 MB of memory compared to 75 GB for the dense representation.

Fortunately, Intel MKL supports the CSR format so very little code modification is necessary to use the corresponding Intel MKL sparse BLAS routine. First, it is necessary to include the `mkl_spblas.h` header. Second, the Intel MKL sparse storage scheme starts array indices at one rather than zero so the values in the MV row and column index arrays must be incremented by one. Finally, the Intel MKL sparse BLAS function `mkl_dcsrgemv` is called to compute a double precision, sparse matrix-vector multiplication in CSR format.

In general, BLAS is used for large computations because of the overhead incurred. The MV small and large problems are too small to merit the overhead of BLAS. For such small problems, the `mkl_dcsrgemv` function actually degrades performance relative to the GNU and Intel compiler baselines (Tables 6 and 7). However, even the large MV problem, which only requires 16 MB of memory, could be considered small when most modern workstations measure memory in gigabytes.

A problem size of N = 1,600,000 and NZ = 250,000,000 takes advantage of a 64-bit address space and amortizes the normal BLAS startup costs while keeping the same degree of sparseness as the original large MV problem. The `mkl_dcsrgemv` function performs about as well as the GNU and Intel compilers but it has an important advantage – it is already threaded, as shown in Table 11.

Extra-large MV problem	Performance (MFLOPS)	Speedup
GNU compiler	183	
Intel compiler	181	1.0
Intel compiler + MKL	177	1.0
Intel compiler + MKL + OpenMP	362	2.0

Table 11. Replacing the hand-coded sparse matrix-vector multiplication in the SciMark 2.0 MV kernel with Intel MKL and enabling multithreading improves performance.

Note: a different system was used to generate this performance data. The differences are as follows: dual Intel Xeon processor with Intel EM64T (3.6 GHz, 1 MB L2 cache, 4 GB memory), Intel Cluster MKL 8.0.

Conclusion

The SciMark 2.0 benchmark was used to demonstrate that Intel programming tools can dramatically improve application performance on Intel-based platforms with a minimum of effort. The Intel C++ compiler for Linux improved the benchmark scores with no modification of the original source code. Through relatively small source code modifications, the Intel Math Kernel Library improved performance even further. The best single-node SciMark 2.0 performance for the small and large problem sizes is shown in Tables 12 and 13, respectively. In nearly every instance the individual benchmark kernels also improve. Consequently, composite scores improve for both problem sizes. The composite score for the large problems improves dramatically because the full power of MKL can be brought to bear on the FFT and LU kernels.

Benchmark	Performance (MFLOPS) for Small Problems		
	GNU baseline	Intel best	Speedup
FFT	510	1817	3.6
SOR	524	1092	2.1
MC	206	1003	4.9
MV	857	832	1.0
LU	884	1827	2.1
Composite score	**596**	**1314**	**2.2**

Table 12. The Intel C++ Compiler for Linux and Intel MKL significantly improve SciMark 2.0 performance relative to the GNU baseline for the small problem sizes.

Benchmark	Performance (MFLOPS) for Large Problems		
	GNU baseline	Intel best	Speedup
FFT	45	600	13.3
SOR	495	1015	2.1
MC	206	1003	4.9
MV	453	457	1.0
LU	392	6646	16.9
Composite score	**318**	**1944**	**6.1**

Table 13. The Intel C++ Compiler for Linux and Intel MKL significantly improve SciMark 2.0 performance relative to the GNU baseline for the large problem sizes.

It is worth noting that many MKL functions are multithreaded so calling these functions allows an application to take advantage of parallel computing. For example, multidimensional DFT's and most BLAS and LAPACK functions can use threads to solve large problems on multiprocessor systems. In the case of the SciMark 2.0 LU and MV kernels, MKL creates threads to take advantage of shared-memory parallelism. By modifying the MKL version of the LU kernel to use Cluster MKL, it is possible to factor very large matrices on a cluster.

With a little more tuning effort it is likely that the performance of the SciMark 2.0 SOR kernel can be improved even further. Perhaps restructuring some loops could improve the data

layout in memory and thus improve performance. The SOR algorithm is readily parallelizable with OpenMP or MPI. However, this article illustrates that with minimal effort, significant performance gains are possible using Intel software products.

References and Additional Resources

Intel® Software Network – This site contains a wealth of information for developers. Numerous technical articles are available, e.g.:

- "Making the Monte Carlo Approach Even Easier and Faster"
- "Monte Carlo European Options Pricing Implementation Using Various Industry Library Solutions"
- "Monte Carlo Simulations with MKL/VSL Random Number Generators" (soon to be published)

Intel Software Network also has interactive forums that are hosted by Intel experts, e.g.:

- Intel Math Kernel Library
- Intel C++ Compiler
- Intel VTune Performance Analyzer for Linux
- HPC and Intel Cluster Tools
- Threading on Intel Parallel Architectures

Intel Software Development Products – Extensive product information and documentation is available online or in the product packages, e.g.:

- Intel Math Kernel Library – Reference Manual
- Vector Statistical Library Notes
- Intel C++ Compiler for Linux User's Guide
- Getting Started with the Intel MPI Library
- Quick-Reference Guide to Optimization with Intel Compilers: A Step-by-Step Approach to Application Tuning with the Intel Compilers

SciMark 2.0 Home Page – The benchmark source code plus additional information and published results are available here.

LAPACK Users' Guide (2nd Edition), E. Anderson *et al.*, Society for Industrial and Applied Mathematics, 1995

ScaLAPACK Users' Guide, L.S. Blackford *et al.*, Society for Industrial and Applied Mathematics, 1997

OpenMP C and C++ Application Program Interface (version 2.0)

About the Authors

Henry Gabb is a Senior Staff Software Engineer in the Intel Parallel Applications Center (Champaign, IL). He has been working on parallel applications and parallel performance issues since he joined Intel in 2000. Henry holds a PhD in biochemistry and molecular genetics from the University of Alabama at Birmingham School of Medicine. Prior to joining Intel, Henry was Director of Scientific Computing at the U.S. Army Engineer Research and Development Center MSRC, a DoD high-performance computing site.

Chirag Shah is a Software Engineer in the Application Design-In Center. He joined Intel in 2000 and has been involved in software optimizations and High Performance Computing since 2003. Chirag holds a Masters in Electrical and Computer Engineering from Carnegie Mellon University in Pittsburgh, PA.