Numerical Algorithms Group

Title: **High-Performance Math Libraries**

Who says you can't get performance and accuracy for free?

Summary: This article examines how to use high-performance math libraries to speed-up application code, and presents tricks used to achieve excellent performance, while still maintaining accuracy. Although this article concentrates on AMD's ACML, the benefits discussed apply to libraries from other hardware and software vendors.

The AMD Core Math Library (ACML) is a freely available toolset that provides core math functionality for Advanced Micro Devices' AMD64 64-bit processor (http://www.amd.com/amd64/). Developed by AMD and the Numerical Algorithms Group (http://www.nag.com/), the highly optimized ACML is supported on both Linux and Windows and incorporates BLAS, LAPACK, and FFT routines for use in mathematical, engineering, scientific, and financial applications.

In this article, I examine how you can use high-performance math libraries to speed-up application code, and present tricks used to achieve excellent performance, while still maintaining accuracy. Although I concentrate on ACML, the benefits apply just as much to libraries from other hardware and software vendors.

BLAS

The ACML contains the full range of Basic Linear Algebra Subprograms (BLAS) to deal with basic matrix and vector operations. Level 1 BLAS deal with vector operations, level 2 with matrix/vector operations, and level 3 BLAS with matrix/matrix operations. Since a surprising amount of mathematics and statistics rely at some level on these operations, NAG in the 1980s became part of an international team that designed a standard set of interfaces for these operations. The result was the netlib suite of BLAS reference source code (http://www.netlib.org/blas/).

Although you can compile the netlib code, that is probably not the best way to get optimum performance. Many key BLAS routines—double-precision generalized matrix-matrix multiply (DGEMM), for instance—can benefit massively from tuning to a specific hardware platform, as is the case with the ACML.

For instance, Figure 1 illustrates the difference in performance between using the DGEMM code from netlib compiled with full optimization, and the ACML code tuned for the AMD64—both running on a single processor 2000MHz AMD Athlon64 machine. (The netlib DGEMM was compiled using the GNU Fortran compiler g77 with -O3 optimization level). Performance is measured in megaflops—millions of double-precision floating-point operations per second. The theoretical peak speed of the processor used for this graph is 4000 megaflops.

Speedups such as this were not gained by using purely high-level languages like Fortran. In fact, for the ACML version of DGEMM, the performance was gained by a heavy dose of assembly language using blocked algorithms designed to take advantage of the processor's cache memory. Fortran wrapper code was then used to set up blocking and handle "cleanup" cases.

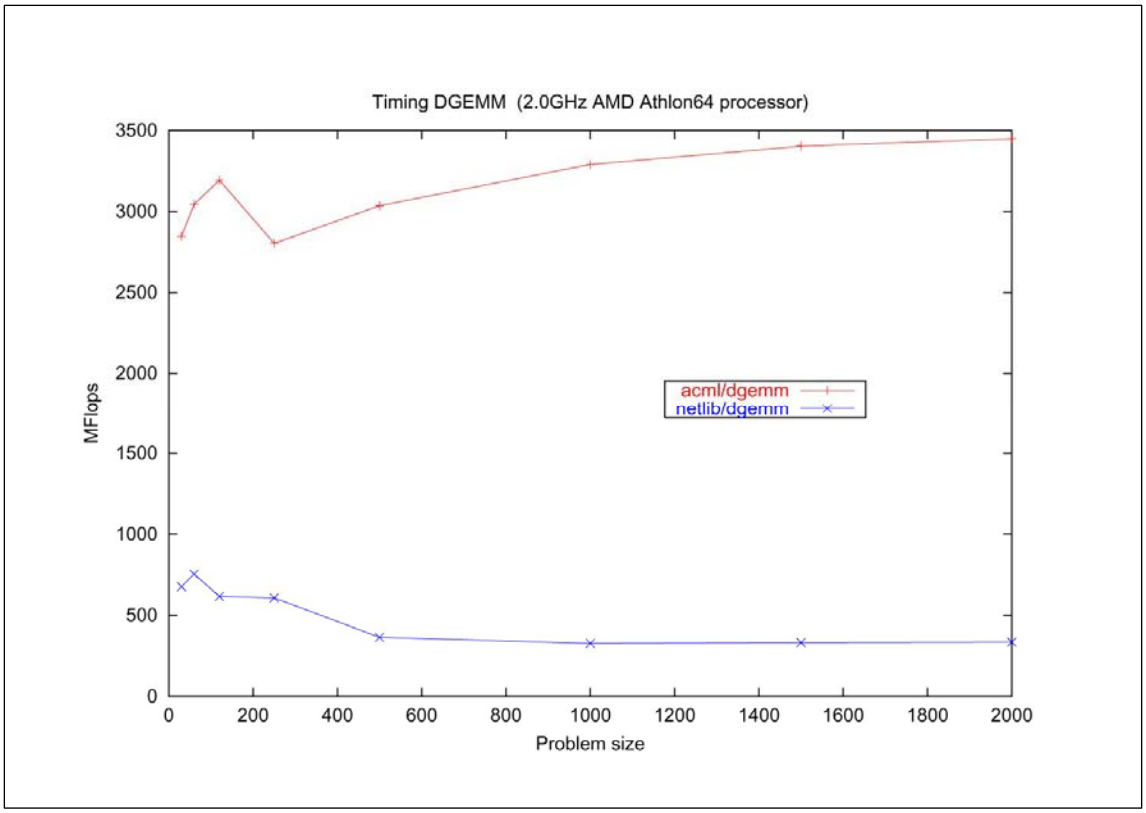

***Figure 1**: Timing DGEMM (2.0GHz AMD Athlon64 processor).*

The ACML assembly kernels use (and for best performance you or your compiler will want to use) Streaming SIMD Extension (SSE) instructions. Single Instruction Multiple Data (SIMD) lets the processor work on several floating-point numbers, packed into a long 128-bit register, at the same time.

In Figure 2, the 128-bit registers *xmm1* and *xmm2* each contain four packed 32-bit floating-point numbers. Multiplying *xmm1* by *xmm2* returns the four products, again in packed format. This operation can be performed significantly faster than four separate 32-bit products. (It's worth noting that SSE instructions don't just appear on 64-bit processors. Newer AMD and Intel 32-bit chips also have them, so the aforementioned comments don't apply just to the 64-bit world.)

xmm1	a1	a2	a3	a4
xmm2	b1	b2	b3	b4
xmm1*xmm2	a1*b1	a2*b2	a3*b3	a4*b4

***Figure 2**: The 128-bit registers* xmm1 *and* xmm2 *each contain four packed 32-bit floating-point numbers.*

In general, programming in assembly language is not recommended because the frustration and general maintenance overheads tend to outweigh performance gains. However, in this case, significant performance gains can be achieved in a truly core routine, which can be used by a great deal of other code. The good news is that hardware vendors take on all the pain, so that we high-level developers don't have to—so long as we remember to take advantage of it!

LAPACK

The Linear Algebra Package (LAPACK) is a standard set of routines for solving simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, singular value problems, and the like. LAPACK 3.0 (http:// www.netlib.org/lapack/) is available across almost all of the vendor math libraries. LAPACK is designed using blocked algorithms, in which large problems are broken down into smaller blocks wherever possible, to take advantage of level 3 BLAS operations. LAPACK routines cover higher level linear algebra tasks, such as factorizing a matrix or solving a set of linear equations. You can tune many LAPACK routines by choosing a block size that fits in well with the machine architecture. A good choice of block size can lead to many times better performance than a poor choice.

Within the ACML, the LAPACK routines gain their speed by proper choice of blocking factor and calls of BLAS assembly kernels, rather than by using their own dedicated assembly. Many key ACML LAPACK routines, such as the matrix factorization routine DGETRF, have also been redesigned internally to take best advantage of the hierarchically cached memory of modern systems like the AMD64. (This matrix factorization is the most important mathematical ingredient of solving a set of simultaneous linear equations.)

Although they maintain identical user interfaces to standard LAPACK, ACML routines have been improved by identifying two kinds of bottleneck:

- Code that must be executed serially, even on a parallel machine.
- Memory access bottlenecks. Sometimes if computations are performed in the natural order, they are required to use data not held in cache, which is slow to operate on.

By rewriting the algorithm used, bottlenecks can often either be removed altogether (perhaps by doing some work earlier than the intermediate results are actually required) or hidden (by postponing work until such time that more can be done with memory-cached data, thus reducing access times).

In addition, ACML LAPACK routines make use of multithreading to achieve extra performance on multiprocessor shared memory systems. ACML uses OpenMP (http:// www.openmp.org/) as a portable way for application programmers to control such shared memory parallelism. (For more information on OpenMP, see "Faster Image Processing with OpenMP," by Henry A. Gabb and Bill Magro; *DDJ*, March 2004.)

Figure 3 shows the performance gain on a multiprocessor system using the LU factorization DGETRF compared to the same code from netlib. The machine used to generate the timing results was a 64-bit 1800-MHz four-processor AMD Opteron system. All Fortran code was compiled with the Portland Group

(http://www.pgroup.com/) Fortran compiler pgf77, using high-level optimization flags. (Although Opteron is a 64-bit processor, it also runs 32-bit code natively.)

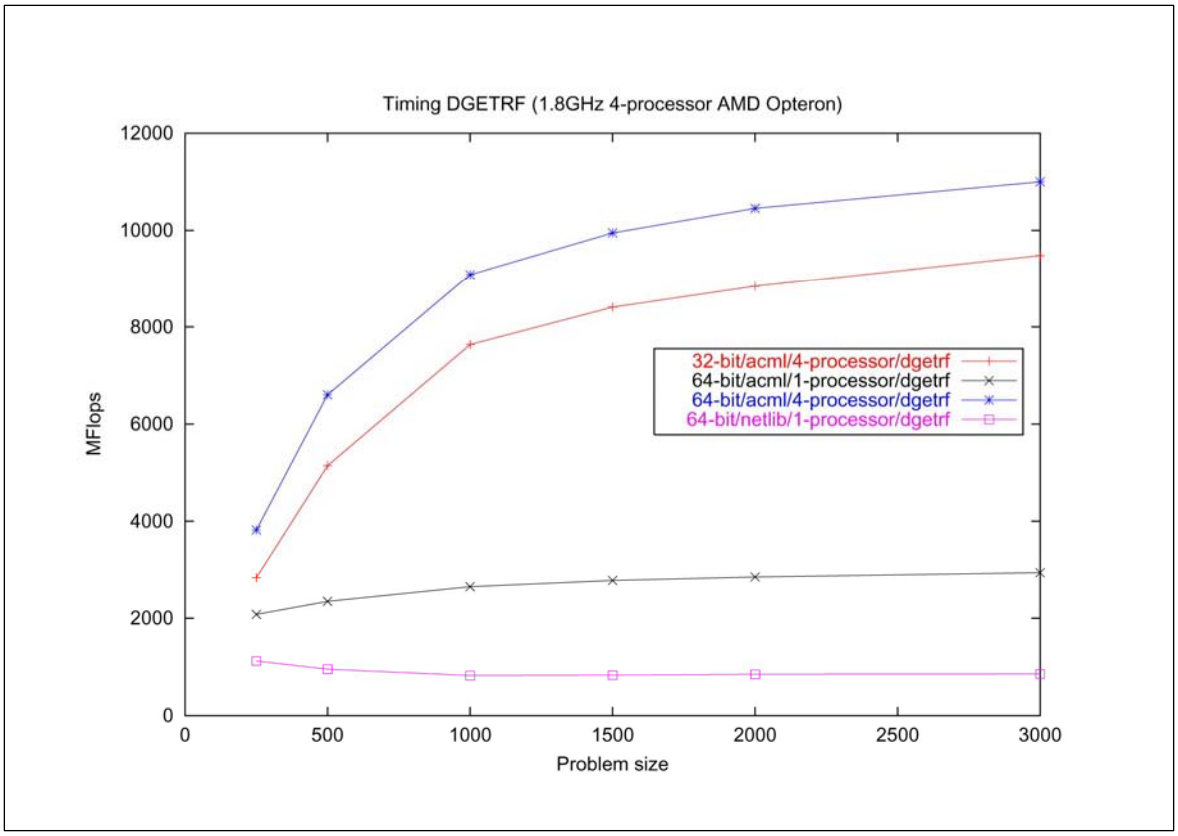

***Figure 3**: Timing DGETRF (1.8GHz four-processor AMD Opteron.*

The version of DGETRF compiled from netlib code was built using the vanilla Fortran BLAS code that comes with netlib LAPACK, whereas the ACML version of DGETRF takes advantage of highly tuned ACML BLAS assembly kernels. No additional tuning of netlib DGETRF was performed (knowledgeable users might improve the performance of DGETRF by changing blocking sizes supplied by LAPACK routine ILAENV). The netlib DGETRF is not parallelized, so running on a one-processor or a four-processor machine makes no difference.

In Figure 3, notice that:

- On a single processor, ACML runs over three times as fast as the vanilla netlib Fortran code.
- ACML scales well. Using four processors, DGETRF performs almost four times as fast as using one processor.
- 64-bit code shows a significant advantage. Running both 64-bit and 32-bit versions of DGETRF on four processors, the 64-bit DGETRF runs about 15 percent faster than the 32-bit DGETRF.

In fact, performance gains of the order of 15 percent for 64-bit code over 32-bit code can be seen in a variety of applications. This includes when running on multiple processors or just a single one, despite the fact that AMD64s run 32-bit code at least as fast as 32-bit AMD chips of the same clock speed. This is a very good reason to upgrade to 64-bit code if you can!

Fast Fourier Transforms

Fast Fourier Transforms (FFTs) are the third common suite of routines in vendor math libraries. FFTs can be used in analysis of unsteady signal measurements—the frequency spectrum computed by the FFT tells you something about the frequency content of the signal. For example, an FFT can help you remove periodic background noise from signals without damaging the part of the signal (such as a music recording) that you are interested in.

Unfortunately, FFT routines do not have a common interface, so extra work is necessary when moving between various vendor math libraries. However, given the high-performance gains possible with FFTs, that exercise should be well worth the effort. This is another area where vendors tend to aggressively tune their code. As with the BLAS, ACML makes extensive use of assembly kernels to achieve high performance.

Because of the way FFT algorithms work, much of the performance that can be got from an FFT implementation depends on the size of the FFT to be performed. For any FFT suite, a data sequence with a length that is an exact power of 2 will likely be transformed faster than a sequence of a slightly different size—sometimes many times faster. This is so much the case that some FFT implementations only work on such sequences, and force you to adapt your data to the FFT routine. With ACML, however, FFTs work on any problem size, though it is always best to have a size that is a product of small prime numbers if possible. That is because, in ACML, the code that deals with these small prime factors has been aggressively tuned with assembly code to take advantage of the 64-bit chip, ensuring that data streams into cache memory in the right order to maximize performance.

ACML comes with FFT routines to handle single- and multidimensional data. The multidimensional routines also benefit from the use of OpenMP for good scalability on SMP machines.

Conclusion

When developing code, it’s important to build on the work of others and not try to build everything from scratch. Of course, we do it all the time. We do not work in machine code, we program in higher level languages or use packages to insulate us from the boring and mundane!

Sometimes though, we forget to extend those principles of reusing others’ tried and tested work as far as possible to gain maximum benefit.

With all this in mind, if in your work you already make use of routines that feature in ACML, why not try linking to ACML? It won’t cost you anything, and you just might be pleasantly surprised at the speedup you see in your application. And while this article has concentrated on 64-bit AMD processor, the same concepts apply to 64-bit chips from Intel and others, and high-performance math libraries should be available for all of them.

Finally, if you're keen to work in assembly language, a good resource is the *AMD64 Software Optimization Guide* (http://www.amd.com/us-en/assets/ content_type/white_papers_and_tech_docs/ 25112.PDF).

By Mick Pont. Mick is a senior technical consultant for the Numerical Algorithms Group. He can be contacted at mick@nag.co.uk.

Courtesy of Dr. Dobb's Journal, www.ddj.com. Article originally published March 2005.

Numerical Algorithms Group

www.nag.com / info@nag.com (North America)
www.nag.co.uk / info@nag.co.uk (All Others)