Auto-tuning a Matrix Routine for High Performance

Rune E. Jensen Ian Karlin Anne C. Elster
(runeerle,iank,elster)@idi.ntnu.no

Abstract

Well-written scientific simulations typically get tremendous performance gains by using highly optimized library routines. Some of the most fundamental of these routines perform matrix-matrix multiplications and related routines, known as BLAS (Basic Linear Algebra Subprograms). Optimizing these library routines for efficiency is therefore of tremendous importance for many scientific simulations. In fact, some of them are often hand-optimized in assembly language for a given processor, in order to get the best possible performance. In this paper, we present a new tuning approach, combining a small snippet of assembly code with an auto-tuner. For our preliminary test-case, the symmetric rank-2 update, the resulting routine outperforms the best auto-tuner and vendor supplied code on our target machine, an Intel quad-core processor. It also performs less than 1.2% slower than the best hand coded library. Our novel approach shows a lot of promise for further performance gains on modern multi-core and many-core processors.

1 Introduction

Since many scientific applications use the same linear algebra calculations, libraries and application programming interfaces (APIs) are designed to allow code reuse and increase portability. One API, the Basic Linear Algebra Subprograms (BLAS) [4, 6, 14], provides linear algebra operations to application developers through a standardized interface. Using the interface programmers insert calls to the proper BLAS routine when developing their code and link to a tuned implementation for their target machine at compile time, resulting in a portable high performing program. Other packages, such as the Linear Algebra PACKage (LAPACK) [1], also built upon them.

The widespread use of the BLAS led to the development of tuned implementations for various hardware platforms. Vendors optimized BLAS kernels auto-tuned implementations that adapt themselves at install time [19] and a single programmer's efforts at a research laboratory writing hand-tuned assembly implementations [9] represent the state of the art.

In this paper, we present an alternative approach that uses a small hand tuned assembly kernel along with an auto-tuner to produce a tuned version of $C = AB^T + BA^T$, a symmetric rank 2 update (The BLAS `syr2k` routine). The resulting kernel, which is used in eigonsolvers, outperforms the vendor provided code and ATLAS [19], which is the best auto-tuned BLAS implementation, by over 5%. We also perform about 1% worse than the GotoBLAS [9] implementation of this kernel, which is the best available hand-tuned

This paper was presented at the NIK-2011 conference; see http://www.nik.no/.

implementation, for a large range of matrix sizes. This work shows that near-optimal performance can be generated from an assembly snippet combined with an auto-tuner.

The rest of the paper is organized as follows: Section 2, we describe state of the art of BLAS implementations and their uses. Section 3 presents the major component of our `syr2k` kernel. We include data access patterns and parameters that are tunable in the discussion. Section 4 describes our target architecture, hand optimized code snippet and auto-tuner. Section 5 contains a description of our test environment and methodology along with a comparison of our implementation to state of the art BLAS routines. Finally, Section 6 summarizes our contributions and describes planned improvements to this work.

2 Related Work

The BLAS (Basic Linear Algebra Subprograms) started as a set of vector operations, now called the Level 1 BLAS [14] that shared a common interface [14]. The subprograms design allows application developers to design portable code that links to efficient implementations on a target machine. The BLAS standard expanded to include matrix-vector functions, the Level 2 BLAS, and matrix-matrix functions, the Level 3 BLAS [6] to take advantage of cache based systems. An updated standard introduces new routines, extends the functionality of some routines, adds routines that combine multiple routines from the Level 1 and Level 2 BLAS and standardizes the inclusion of sparse matrix kernels [4]. Parallel versions for distributed memory machines have also been developed [7].

With a standard interface to vector operations EisPACK [18] and LinPACK [5] built off the Level 1 BLAS. These packages provide eigensolver and matrix solving functionality. LAPACK [1] combines the functionality of EisPACK and LinPACK into one package and takes advantage of the matrix-vector and matrix-matrix routines introduced in the Level 2 and Level 3 BLAS. Even more complex solver packages, such as Trilinos [11] and PETSc [2] often wrap the BLAS and LAPACK to enable their use.

The importance of BLAS routines to scientific program performance has resulted in many efforts to tune these kernels. Work by Lam *et al.* [13] shows that the performance of matrix-matrix multiply routines is susceptible to the block size chosen and that small changes in block size often result in large performance differences. Bilmes *et al.* [3] highlight important tuning techniques of high performing routines.

Tuned BLAS implementations released by hardware vendors take advantage of these tuning techniques. The cost of constantly updating routines for new architectures led to efforts to auto-tune BLAS routines. The Portable High Performance Ansi C (PHiPAC) [3] project demonstrated important machine factors to tune for and that the idea was feasible. Automatically Tuned Linear Algebra Subprograms (ATLAS) [19] produces portable auto-tuned performance that is nearly as good or better than vendor BLAS using install time benchmarks and auto-tuning of C code.

Since the advent of auto-tuned software further improvements have occurred. Goto developed GotoBLAS, which outperforms most vendor-released BLAS implementations and auto-tuned BLAS packages by combining hand-coded assembly and new tuning techniques [9], such as tiling for the TLB when performing matrix-matrix multiplication. Yotov *et al.* demonstrated that by using a model one can produce code as efficient as ATLAS in about half the time [20]. In a later work, they combined modeling and search to achieve better performance than their model alone and ATLAS [8]. More recent work has focused on auto-tuning small size kernels [17], fast GPU implementations [16] and using specially designed processors [15], to speed the performance of matrix-matrix

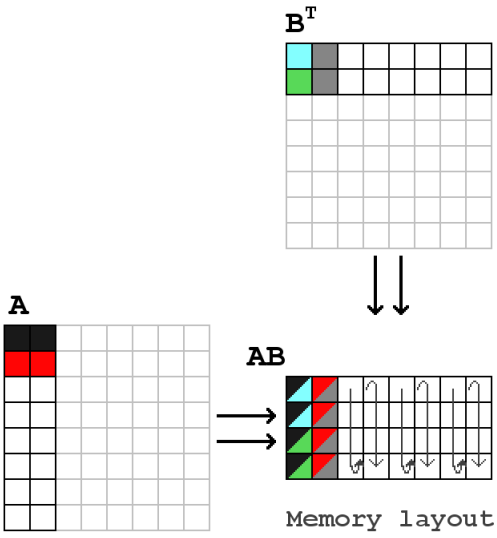

Figure 1: Pattern Used to Copy and Interleave Data

multiplication.

3 SYR2K Design

This section discusses the design of our `syr2k` routine. It starts by describing the three high-level optimizations used. We then explain the tunable components of our implementation starting from the low level code snippet and working up to high level details. We also describe the C support code and an OpenMP parallel version of the code. Low level details, are left to the next section.

High-Level Optimizations

There are three high-level optimizations used in our implementation: copying data, interleaving data and taking advantage of symmetry.

Copy Optimizations and Data Interleaving

To achieve good performance, matrix-matrix multiplication routines use loop tiling to reuse data stored in cache. Loop tiling breaks the iteration of one loop into two loops with the inner loop accessing only an amount of data that can fit within cache. However, loop tiling leads to non-consecutive data accesses since only part of each matrix row is read. Also, when two matrices are read from different memory addresses conflict misses can occur. To remove non-consecutive reads and reduce conflict misses we reorganize the internal storage of each tile in the A and B matrices as shown in Figure 1. The rows of the A matrix and the columns of the B^T matrix are interleaved with elements of A and B alternating in the new storage format. The resulting AB matrix has a serial data access pattern and reduced conflict misses.

Symmetry

Exploiting the symmetry of the resultant matrix is essential to good performance of `syr2k`. The most common way to exploit the symmetry is to use a general matrix multiplication to calculate $C = AB^T$, where B^T represents the transpose of the B matrix.

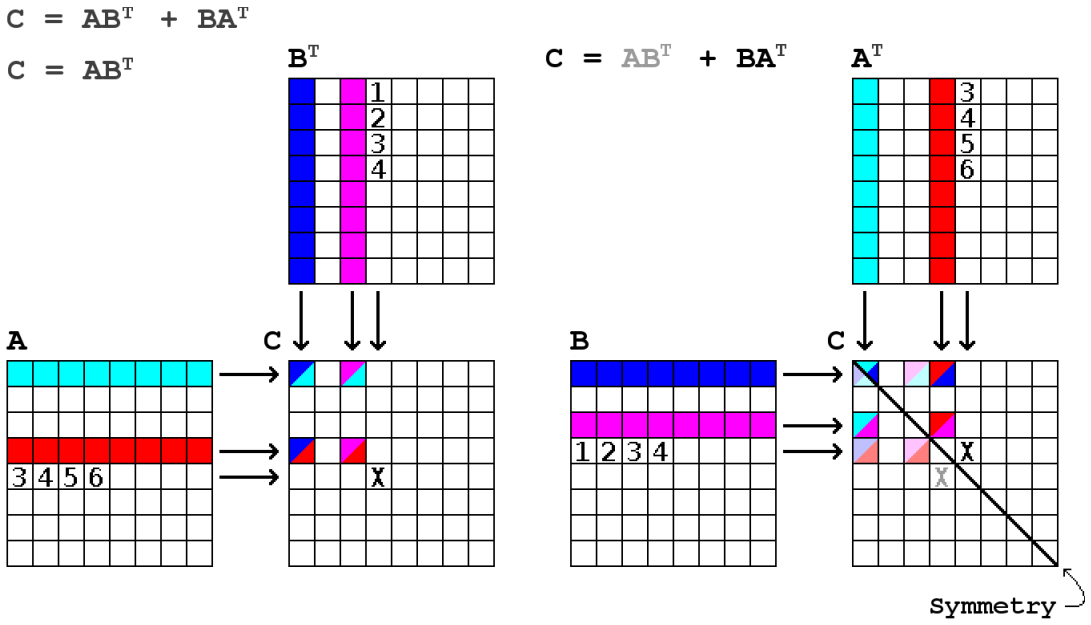

Figure 2: Symmetrical Properties

Then $C = AB^T + BA^T$ is calculated by summing the lower left and the upper right triangular portions of the C matrix. However, storing both triangles is redundant.

Our method calculates one triangle in C, performing the calculation $C = AB^T + BA^T$ directly, as shown in Figure 2. The left side of the figure illustrates calculating $C = AB^T$. The C matrix is colored to indicate the rows and columns from the A and B^T matrices needed to calculate that value. The right side of the figure shows the calculation of $C = BA^T$ in solid colors, $C = AB^T$ in faded colors and the axis of symmetry. Note the same entries are used to calculate the value of x in both triangles, while the location it is written to differs. Therefore, by calculating the rows of AB^T and columns of BA^T simultaneously, and then immediately summing them, we only have to store half of the resultant matrix reducing reads and writes to C by half.

Tunable Code Components

In this section, we present the code components that are either hand-tuned or auto-tuned to generate a high performing `syr2k` routine. The reasons we tune for these parameters are detailed in [10] and are omitted due to space constraints. The following describes our assembly snippets and further optimization through unrolling and cache tiling.

Hand Coded Assembly

To avoid compilers producing non-optimal assembly code from generated C code, we use a hand-coded and automatically generated assembly snippets. The snippets form the basis of the routine as the rest of the implementation involves the auto-tuner copying it, adjusting loop structures and testing various tuning parameters. The instructions structure is hand-coded whereas register names, operands and offsets are assigned dynamically. Since the snippet affects the routine's performance severally it is designed as optimally and small as possible to allow for more options in the tuning process.

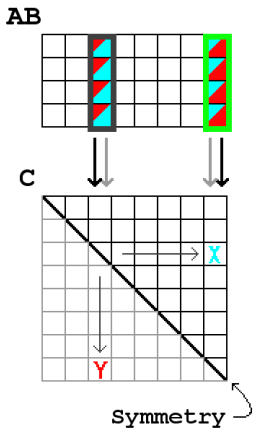

Figure 3: Data Read Pattern

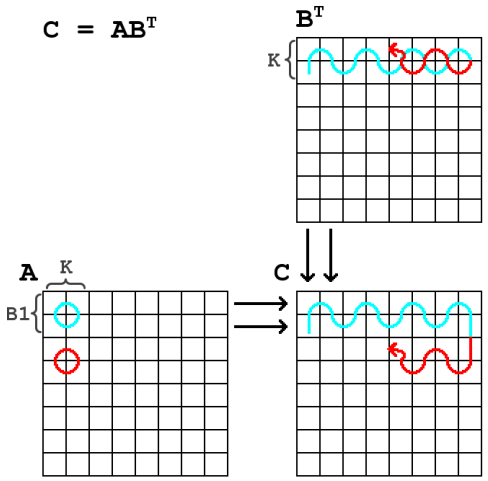

Figure 4: Level 1 Cache Tile Read Pattern

Loop Unroll Factor

The inner-most loop unroll factor (K) is the number of times the code snippet is duplicated in assembly without loop logic. The parameter also matches the length of each line of the interleaved AB matrix, which has K values from each matrix on each line. The value of K is auto-tunable with larger values decreasing loop overhead and allows the out of order execution unit of the processor more scheduling options, but also increases the size of level 1 cache tiles and executable size. At the end of the unrolled code segment a software prefetch instruction is inserted.

Level 1 Cache Tile Size

During execution of `syr2k` data are read from two columns of the AB matrix, as shown in Figure 3. A read pattern that exploits several levels of the memory hierarchy simultaneously, including the bandwidths of the Level 1 and 2 caches and main memory, increases the locality of memory reads, and balances the ratio of reads from memory structures of varying speeds, was used. Good locality and balance is achieved when the data in one column are read many times and used once per read from the Level 1 cache. The data in the other column are read once from a larger cache or memory and used many times.

The Level 1 cache tile size $B1$ is an auto-tunable parameter that controls block size in two directions. The tile is auto-generated in assembly from the unrolled code with optimal tile sizes keeping data read multiple times from the Level 1 cache within it as long as possible. Figure 4 illustrates how the optimized read pattern, used to reduce data movement in the tiles, traverses the original data layout.

Level 2 Cache or TLB Tile Size

Our C support code also uses a second level of tiling. The selection of tile size $B2$ is auto-tuned and targeted at either the TLB or Level 2 cache. The size of the blocks and the structure the blocking targets is determined by the auto-tuner. The access pattern of Level 1 cache blocks within the level 2 blocks is shown in Figure 5. The pattern ensures that some data used by the next Level 1 block is already stored within cache. Each level 2 block row is divided into three sectors with the last one extending to the end of the matrix.

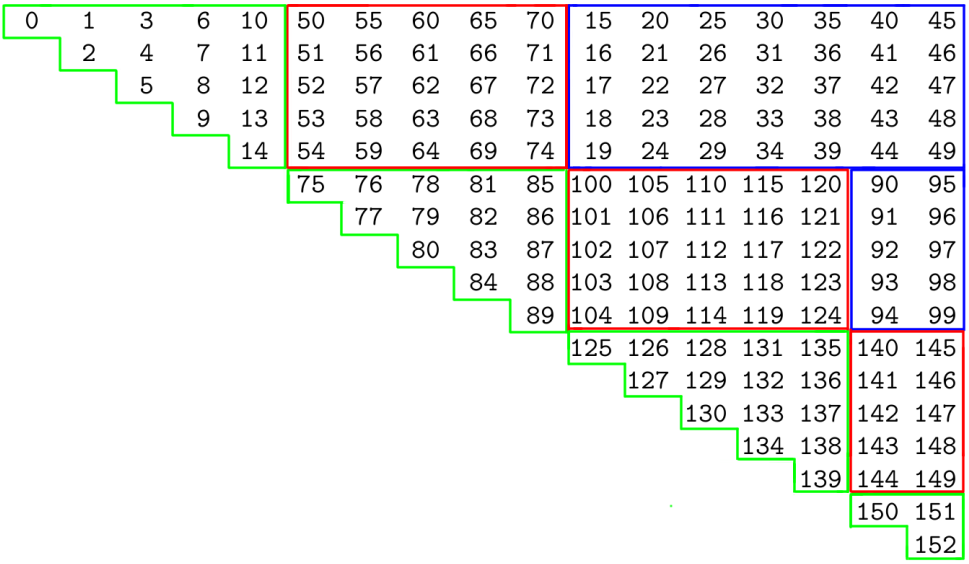

Figure 5: Level 2 loop tiling access pattern.

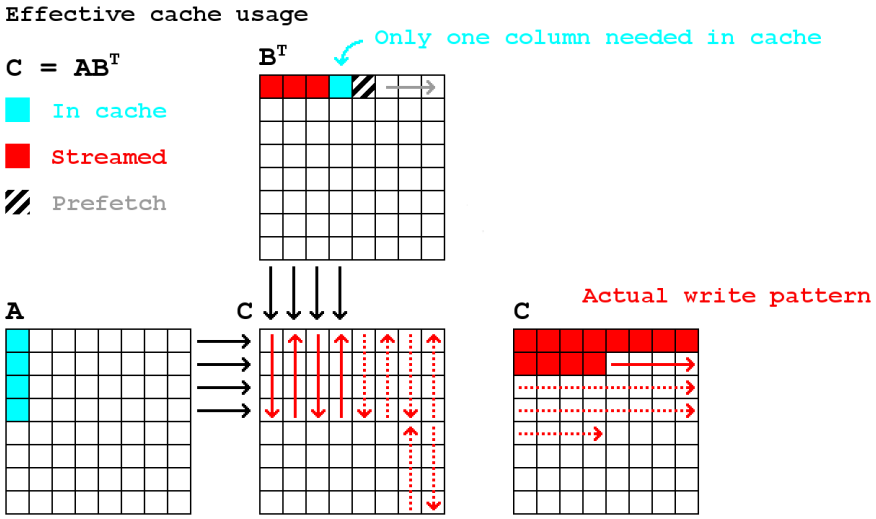

Figure 6: Effective and actual write pattern

High Level C Support Code

The high-level C support code works in two phases and calculates the addresses for software prefetching. The first phase performs the computation in two steps that are applied sequentially to each K element long chunk of A and B^T. In the first step, the copy optimization and interleaving of A and B^T is performed. Then in the second step the computation is performed for the same chunk.

The second phase of the support code reorders the result of the computation. Loop tiling changes the order in which the data values are written to the C matrix. This results in a non-consecutive write pattern. To efficiently use memory and hardware pre-fetching, the write pattern is linearized for every blocking level. Figure 6 illustrates both the mathematically correct index storage as well as our implemented write patterns. After the calculation completes, we restore the original (correct) pattern by relocating the data in C.

Parallel Version

Most modern machines contain two or more multi-core processors in a shared memory environment. To test the scalability of our `syr2k` routine on these systems we implemented a parallel version using OpenMP compiler directives. The version required a few changes to the setup and has issues that result in suboptimal performance.

Since OpenMP requires a single loop iterator as input, and the iteration sequence in the serial version works over two loops, a support function calculates a conversion of the two iteration components into a single loop iterator. This function introduces overhead and a $O(\log n)$ component to the runtime, decreasing performance. Also, software prefetching address calculation is not always correct in the parallel version, resulting in reduced performance for large matrices.

4 Tuning SYR2K

In this section, we describe the tuning of our code for the Intel Enhanced Core 2 processor. We start by presenting the relevant features of the this architecture. Then we explain how we tuned our assembly code snippets. Finally, how our auto-tuner works and the values it chooses for tunable parameters, is presented.

Target Architecture

As mentioned, the target hardware for our optimized code is an Intel Enhanced Core 2 processor. This processor has a 32KB Level 1 cache for each core and separate 6 MB caches shared by two of the cores. It has 16 SSE registers that each stores 16 bytes of data (2 doubles or 4 floats). Each clock cycle the processor can decode four SSE instructions that are no larger than 16 bytes in aggregate. It can execute three SSE instructions per cycle, but no more than one of each arithmetic operation, from the following categories: addition, multiplication and register copy. For double precision this results in 4 floating point operations per cycle (Flops) as a performance bound per core. During each cycle it can also perform one memory read and one memory write, either of which can be fused with an arithmetic operation. Also constraining the implementation is the two operand instructions x86 uses. One of the values used in computations is overwritten and must be copied before it is overwritten. To our advantage, the Core 2 processor is capable of out-of-order execution. Therefore, for high performance the instructions which are issued each cycle, do not need to fall within these maximums. However, they but must be maintained as an average over a short execution window.

Tuning the Core Assembly

In designing the assembly code snippet, various layouts were implemented and performance tested. However, we only include the final layout here, as shown in Figure 7. The details of some of the routines tried, and a theoretical evaluation of them are presented in [12].

The final implementation uses a $2x4$ block containing eight additions, eight multiplications, seven memory loads, two of which are fused with multiplications, and three register to register copy instructions. Seven memory loads are used, although only six are needed, to reduce the number of register to register copy instructions. The extra memory load is performed on the 4th location in the I column of AB, reading the same data from memory twice. The pattern contains 24 instructions that can be written as 128 bytes as shown in [12]. Therefore, it is possible for the code snippet to be run in eight

			AB Column *J*	
			1	2
			$R_{j1} \leftarrow \text{Mem}[AB_{j1}]$	$R_{j2} \leftarrow \text{Mem}[AB_{j2}]$
AB Column *I*	1	$R_{i1} \leftarrow \text{Mem}[AB_{i1}]$	$R_{tmp} \leftarrow R_{i1}$ $R_{tmp} \leftarrow R_{tmp} * R_{j1}$ $R_{acc1,1} \leftarrow R_{acc1,1} + R_{tmp}$	$R_{i1} \leftarrow R_{i1} * R_{j2}$ $R_{acc1,2} \leftarrow R_{acc1,2} + R_{i1}$
	2	$R_{i2} \leftarrow \text{Mem}[AB_{i2}]$	$R_{tmp} \leftarrow R_{i2}$ $R_{tmp} \leftarrow R_{tmp} * R_{j1}$ $R_{acc2,1} \leftarrow R_{acc2,1} + R_{tmp}$	$R_{i2} \leftarrow R_{i2} * R_{j2}$ $R_{acc2,2} \leftarrow R_{acc2,2} + R_{i2}$
	3	$R_{i3} \leftarrow \text{Mem}[AB_{i3}]$	$R_{tmp} \leftarrow R_{i3}$ $R_{tmp} \leftarrow R_{tmp} * R_{j1}$ $R_{acc3,1} \leftarrow R_{acc3,1} + R_{tmp}$	$R_{i3} \leftarrow R_{i3} * R_{j2}$ $R_{acc3,2} \leftarrow R_{acc3,2} + R_{i3}$
	4		$R_{j1} \leftarrow R_{j1} * \text{Mem}[AB_{i4}]$ $R_{acc4,1} \leftarrow R_{acc4,1} + R_{j1}$	$R_{j2} \leftarrow R_{j2} * \text{Mem}[AB_{i4}]$ $R_{acc4,2} \leftarrow R_{acc4,2} + R_{j2}$

Figure 7: Final Core 2 instruction layout.

cycles at 100% efficiency. Of note using SSE registers requires two elements of each matrix to be stored consecutively. Therefore, the interleaving described in Section 3 is performed with two elements of A followed by two elements of B^T.

Auto-tuner

The auto-tuner consists of three perl scripts. One of these scripts drives the process and takes in a set of parameters along with start, stop and step size to search over. First, it exhaustively tests all combinations of all level 1 cache block sizes ($B1$) and loop unroll factors (K). Then it takes the optimized level 1 code and searches over level 2 block sizes. Once block sizes are decided it tests other parameters that have a small impact on the produced routine's speed.

The other two scripts are called from the main script. The first script takes in values for $B1$ and K. It then produces an assembly implementation of the level 1 cache block by duplicating the assembly snippet K times and adding in loop support to tile the code. The output from this script is a .asm file that contains as comments the internal build selections and parameter choices.

Then the second perl script takes the output file and reformats it to match the gcc in-line specifications creating a C function containing only assembly code. The code is inlined into the C support code allowing gcc to first allocate registers for the support code from those not used by the inline assembly. The resulting code is then performance tested.

Currently the auto-tuner generates code optimized for a single input size and does not generate cleanup code. Cleanup code is important because the inner loop unroll factor and level 1 block are in assembly and must be fully executed. So unnecessary work is performed for matrix sizes not divisible by K and the level 1 block size.

5 Results

In this section, we present the experimental setup and methodology used to test our `syr2k` implementation. We then present serial results comparing our implementation to state-of-the-art BLAS routines on a single processor of our test machine. We conclude the section with parallel results comparing our OpenMP implementation of parallel `syr2k` to other parallel implementations.

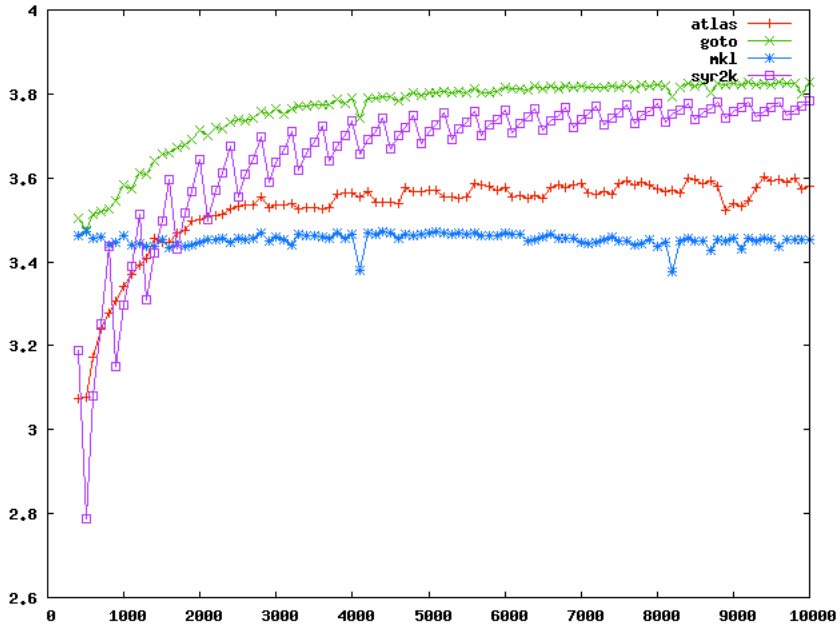

Figure 8: Serial Performance Comparison

Experimental Machines and Test Methodology

All experiments were run on the Clustis3 cluster at NTNU. The machine contains identical nodes with two Intel Xeon E5405 quad-core processors. Each node contains 8 GB of memory connected to the processor via a 1333 MHz system bus.

Our implementation of `syr2k` was compared to three state-of-the-art BLAS libraries; 1) version 10.3.4.191 of the Intel BLAS library MKL [1], 2) GotoBLAS2 version 1.1.13 [2] and 3) ATLAS version 3.9.45 [3]. All of these were the newest publicly available versions as of June 2011. They were compiled using `gcc` version 4.6.0 and installed using the default options provided by the software packages. For all tests we ran 5 repetitions and used the median value.

Serial Results

Our auto-tuner was run to tune for a $2000x2000$ matrix. The tuning was performed by running tests exhaustively for K values of 8 to 152 at steps of 8, $B1$ from 4 to 32 at steps of 4 and $B2$ values from 16 to 128 at steps of 16. The search resulted in a K of 80, $B1$ of 16 and a $B2$ of 64. Figure 8 compares our produced routine's performance to the other BLAS libraries on square matrix sizes from 300 to 10,000 at 100 steps.

Since no cleanup code is generated, and code must be executed in 80 long unrolled blocks, performance for matrix sizes not divisible by 80 suffers. A jagged performance pattern that peaks at every 400 size results. Despite these limitations, our code outperforms ATLAS and MKL for all sizes larger than 2000 and some smaller sizes. Performance for sizes that are multiples of 400 perform within 1.2% of GotoBLAS.

Other experiments suggest that the jagged performance can be eliminated. For example, performing a search for $1000x1000$ results in 3.48 flops/cycle for $K = 112$ and $B1 = 8$. Using the code optimized for $2000x2000$ resulted in 3.31 flops/cycle. A

[1] http://software.intel.com/en-us/articles/intel-mkl/

[2] http://cms.tacc.utexas.edu/tacc-projects/gotoblas2/downloads/

[3] http://sourceforge.net/projects/math-atlas/files/

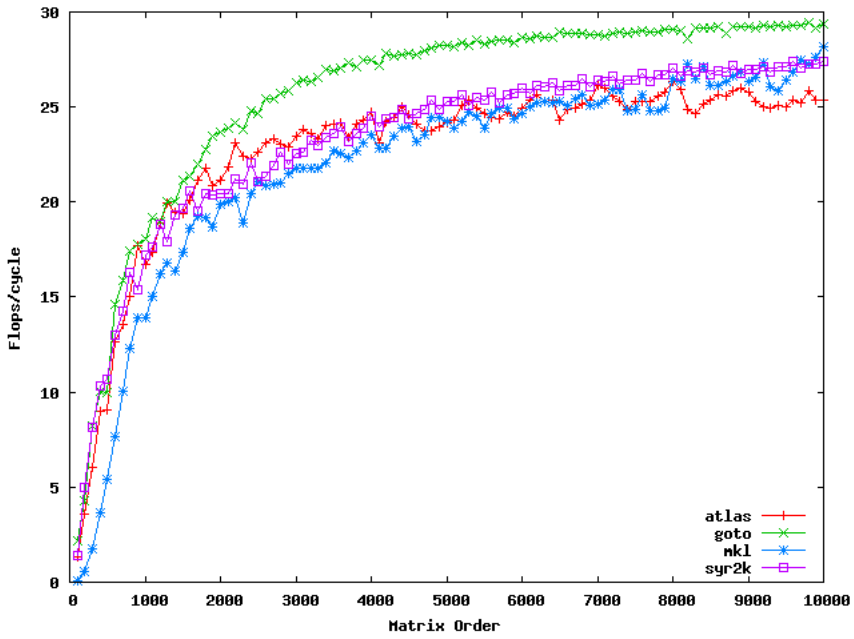

Figure 9: Parallel Performance Comparison

search using 3000x3000 results in 3.68 flops/cycle and parameters of $K = 120$ and $B1 = 8$. Using the code optimized for 2000x2000 resulted in 3.62 flops/cycle. Running the auto-tuner on a 4000x4000 matrix resulted in the same parameters compared to the 2000x2000 optimized code. This occurs since both of these cases have little wasted computation and are multiples of each other. In theory, we should have optimized for all sizes, but instead looked at other issues.

Parallel Results

Figure 9 shows the performance of our parallel implementation. The code used in our tests was a parallel version of our kernel for 2000x2000 matrix. Our implementation outperforms MKL for all, but five large sizes, and ATLAS for all sizes larger than 4000. Performance on smaller matrices is about 20% slower than GotoBLAS with the gap closing to 10% for larger matrices.

The performance of our implementation is affected by us currently not accounting for shared caches or any other parallel optimizations. Not considering shared cache effects can reduce performance significantly, since the cores that share the same cache can cause capacity or conflict misses to the other core's data.

6 Conclusions and Future Work

This paper shows that by using a small amount of hand-optimized structures together with assembly code and an auto-tuner one can generate a near-optimal BLAS routine. Our generated serial test routine, `syr2k`, outperforms ATLAS and MKL on our test machine. This routine performed just over 1% slower than the best known hand-tuned serial implementation. Our parallel implementation outperforms the auto-tuned libraries and vendor libraries for most matrix orders, despite not taking into consideration multi-core issues, such as shared caches. From these results we show that auto-tuning using a small amount of assembly code is a viable alternative to C-based auto-tuners and hand optimized routines.

To further our work, we plan to improve the auto-tuner to generate kernels for various size matrices and produce cleanup code to avoid padding issues. Implementations of other Level 3 BLAS routines on newer processor architectures, such as Intel's Sandy Bridge, AMD's Fusion and NVIDIA's Fermi will also be considered. Investigating multi-core specific tuning techniques, such as shared cache optimizations, will also be considered. In addition, techniques to reduce overhead, for example in the copy optimization phase, will be explored.

Acknowledgements

The authors would like to thank Dr. Thorvald Natvig and Jan Christian Meyer for use of their test environment and conversations that helped shape the paper.

References

[1] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. DuCroz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorenson. LAPACK: A portable linear algebra library for high performance computers. In *Proceedings of Supercomputing '90*, pages 2–11, New York, NY, November 1990.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[3] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of 11th International Conference on Supercomputing*, pages 340–347, New York, NY, July 1997. ACM Press.

[4] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002.

[5] J. J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, Pa., 1979.

[6] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[7] A. Elster. Basic matrix subprograms for distributed memory systems. *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 311–316, 1990.

[8] A. Epshteyn, M. Garzaran, G. DeJong, D. Padua, G. Ren, X. Li, K. Yotov, and K. Pingali. Analytic models and empirical search: A hybrid approach to code optimization. In *Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, pages 259–273. Springer Berlin / Heidelberg, 2006.

[9] K. Goto and R. Van De Geijn. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, 35:4:1–4:14, July 2008.

[10] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):25, May 2008.

[11] M. A. Heroux, R. A. Barlett, V. E. Howell, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, September 2005.

[12] R. E. Jensen. Techniques and tools for optimizing codes on modern architectures: A low-level approach. Master's thesis, Norwegian University of Science and Technology, Trondheim, Norway, May 2009.

[13] M. S. Lam, E. E. Rothber, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Palo Alto, CA, Apr. 1991.

[14] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[15] J. Makino, H. Daisaka, T. Fukushige, Y. Sugawara, M. Inaba, and K. Hiraki. The performance of grape-dr for dense matrix operations. *Procedia CS*, 4:888–897, 2011.

[16] N. Nakasato. A fast gemm implementation on the cypress gpu. *SIGMETRICS Perform. Eval. Rev.*, 38:50–55, March 2011.

[17] J. Shin, M. W. Hall, J. Chame, C. Chen, and P. D. Hovland. Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology. In *In The Fourth International Workshop on Automatic Performance Tuning*, 2009.

[18] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Kelma, and C. B. Moler. *Matrix Eigensystem Routines EISPACK Guide*, volume 6. Springer-Verlag, New York, 2nd edition, 1976.

[19] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of 1998 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–27, Washington DC, November 1998. IEEE Computer Society.

[20] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358 –386, feb. 2005.