CSE Master Program in
DMATH/D-PHYS

Semester Thesis

Extensions to Statically Accelerated Loop Templates

Optimizing Cache Usage in Matrix-Vector Multiplication

Author:
Andrew Foster

Supervisors:
Dr. Andreas Adelmann
Yves Ineichen

March 1, 2012

1 Introduction

Matrix-vector multiplication lies at the core of a whole range of computational tasks in disciplines from simulation and modeling to signal processing and graphics. We present an extension to the SALT (Statically Accelerated Loop Templates) library that improves matrix-vector arithmetic throughput over exisitng systems for large problems.

2 Previous Work

The SALT library provides a general framework for arbitrarily complex arithmetic on vectors of various C++ floating point data-types. It combines generously unrolled loop structures with expression templates [**?**] over an opaque interface to low level vectorization intrinsics to obtain high numerical throughput (approaching theoretical peak performance) for vector (BLAS Level 1) operations [**?**].

This is achieved by supplying the compiler with the information needed to optimize the evaluation of vector expressions. Firstly, unrolling simple loops (by as many as 48 operations) removes false dependencies and allows the compiler to pack the CPU registers for faster operand access. Likewise, reordering evaluation instructions, by clustering bursts of data reads followed by the operations to be performed, capitalizes on modern CPUs out-of-order execution capabilities and facilitates optimizing memory access. On a higher level, the expression template machinery allows compiler analysis to combine static calculations on multiple vectors into a single loop, often reducing the number of costly memory accesses by integer multiples. Finally, SALT abstracts the interface to the Streaming SIMD (Single Instruction Multiple Data) Extensions to the x86 architecture (specifically SSE3) for floating-point and double-precision data-types. The end result achieves the performance of hand coded SSE operations while conveniently hiding the complex memory management, alignment, and access and verbose operation execution from the programmer.

SALT had seen impressive benchmarks on an Intel Core i5-580M [**?**], often outperforming well-established and reviewed open-source expression template libraries, such as Eigen3, and highly tuned proprietary systems for vector arithmetic, like Intels Math Kernel Library (MKL).

3 AMD Benchmarks

The first task was to repeat the performance benchmarks for the AMD x86 architecture using the Opteron 8380 equipped nodes on the Brutus cluster. As before, the performance was compared for four different single-precision operations (simple dot product, in-place vector scaling, vector update [sAXPY], and out-of-place vector scaling) against Eigen3 and MKL using the Gnu and Intel compilers. See figures **??** - **??** in the appendix for performance plots. SALT

shows an impressive performance lead on almost all operations using both compilers, the only exception being medium sized dot products under ICC where MKL dominates. Unsurprisingly, the greatest performance gap occurs during the out-of-place vector scaling test; since this cannot be accomplished with a single BLAS call, MKL must use two separate vector operations (sCopy and sScale) to achieve the desired result, while expression templates allow SALT to combine these into one loop for a 2x performance gain. Most interestingly, for the dot product operation (figure **??**) we reach 90% of the theoretical peak performance of the processor (10 GFlop/s single precision)

The same test suite was repeated to examine double-precision performance yielding lower overall throughput in all cases but similar qualitative results, as can be seen in figures figures **??** - **??** in the appendix.

4 Matrix-Vector Multiply

The next task was to implement matrix-vector multiplication in the context of the SALT framework and benchmark the performance against Eigen and MKL. This task saw a number of challenges, namely optimizing cache usage by recursively subdividing the problem into appropriately sized pieces and handling the reduction of these intermediate results.

4.1 Nave Implementation

The first method for implementing matrix-vector multiplication using SALT relies on the standard dot-product machinery supplied in the vector portion of the library. In this naive implementation, the matrix class is little but a wrapper around a collection of vectors (each representing a row of the matrix) with appropriate access methods. Matrix-vector multiplication is then implemented by simply taking the dot product of a vector with each of the matrix rows and storing the results in an output vector. Figure **??** shows the results of this operation benchmarked against the corresponding functions in Eigen3 and MKL (using the sgemv BLAS call).

As an additional improvement to reduce memory consumption and promote data locality, the matrix constructor was updated so as to allocate the row-vector storage as one contiguous block using a single call to _mm_alloc; however, this resulted in no significant performance improvement.

Given that the SALT dot product operator vastly outperforms the alternatives for almost all vector sizes, it is surprising that this does not extend to the case of matrix-vector multiplication (by simply leveraging the dot product infrastructure).

4.2 Cache Oblivious Algorithm

In an effort to improve the performance for large matrix sizes, we attempted to optimize the memory access using a cache-oblivious (CO) algorithm for matrix-

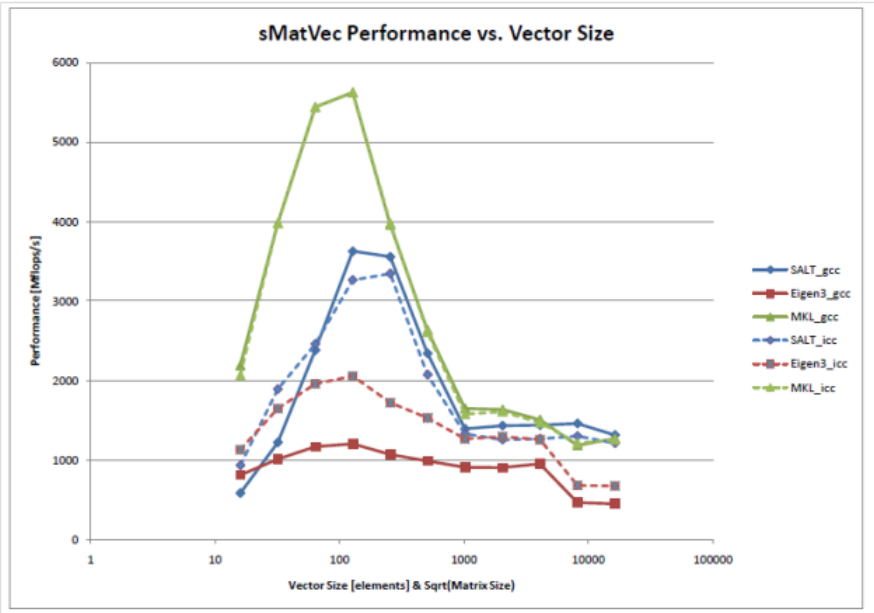

Figure 1: Performance metrics for the naive implementation benchmarked against the MKL and Eigen3 implementations.

multiplication. Generally speaking, CO algorithms use a divide-and-conquer approach to break a large problem into a series of smaller components, thereby increasing the granularity of the overall task to more effectively exploit the memory hierarchy [**?**]. In this case, we exploit the associativity of the problem by subdividing the matrix-vector product evenly along the row and column axes (i.e. into four approximately equal pieces), recursively applying this process until the sub-problems meet a certain base-case terminating criteria (such as, being able to fit into the processors L1 cache). These smaller cache-contained matrix-vector products are then evaluated in a depth-first fashion, and the results are combined by intermediate reduction steps, using the vector addition operations already defined by SALT. Figure **??** shows the results of benchmarking this method against the Eigen3 and MKL implementations.

Unfortunately, this method demonstrates significantly reduced performance, as compared with the naive implementation, for a number of reasons. Most severely, the overhead of recursive calls to the subdivision routine, which allocates a series of temporary objects and performs intermediate logical and reduction operations, adds a nontrivial computational burden to the multiplication process. Secondly, while cache usage may be somewhat improved, the underlying matrix representation in memory (row-major storage) does not accommodate the heavily non-sequential access patterns required for the base-case of the recursion.

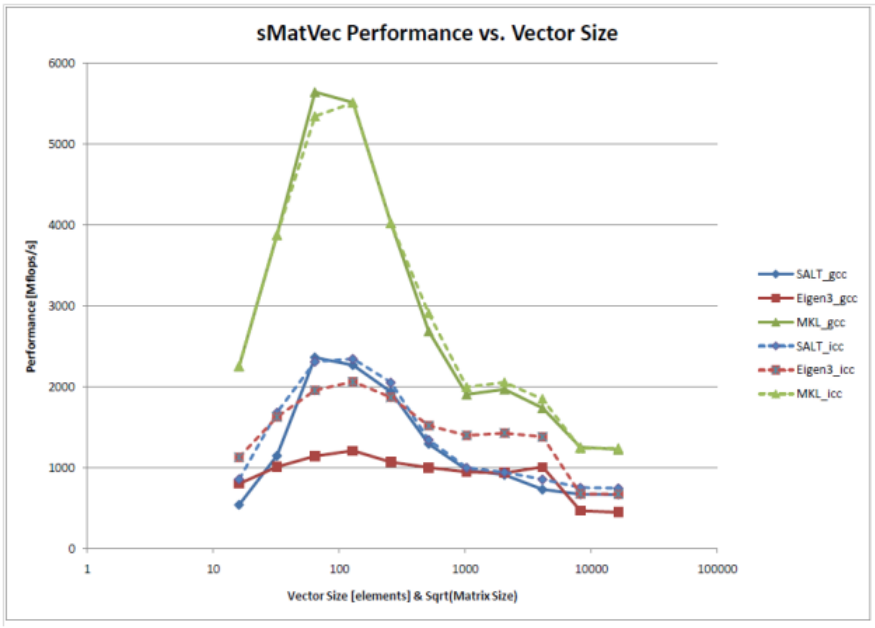

Figure 2: Performance metrics for the cache oblivious matrix-vector multiply benchmarked against the MKL and Eigen3 implementations.

5 Recursive Block Storage

5.1 Theory & Implementation

In an effort to eliminate the recursion and non-sequential access overhead, we implemented Recursive Block Storage (RBS) for the matrix memory representation. This method is nearly equivalent to reordering the row-major memory representation to match the access patterns dictated by the CO algorithm. This way, we effectively move the recursion overhead to the matrix initialization and promote data locality by storing the elements in the order they are accessed. The RBS scheme begins with a base-case of a size determined at compile time and generally chosen, like in the CO method, to fit into the processors L1 cache. Any matrix-vector multiplication method can be implemented for the base case (we use a row-major naive multiplication, to simply demonstrate the benefits of the RBS scheme) as the intermediate results are combined with an additive reduction step. We then partition the full matrix into smaller sub-problems by tiling the structure with base-cased sized elements. For an NxN matrix and a base-case of size bxb, we obtain a two-dimensional set of $(N/b)^2$ sub-matricies that are then stored linearly in memory using a space-filling curve. Although there are a number of different options for this mapping, we used a Morton Z-order curve to balance data locality with simplicity and ease of implementation. This allows simple constant-time translation of the full matrixs row-column in-

dex into a memory location using efficient bit-interleaving algorithms. Figure **??** demonstrates an example for a matrix where N/b equals 7.

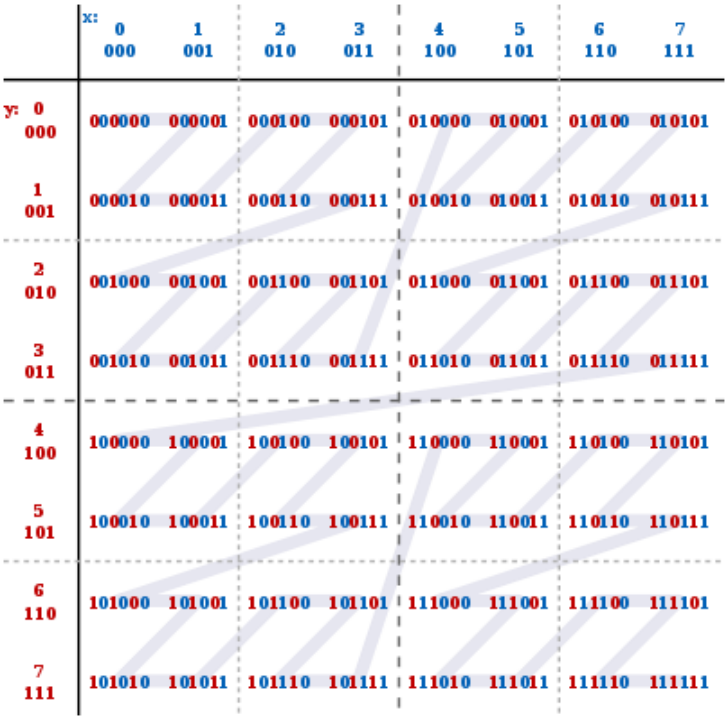

Figure 3: This demonstrates the mapping from two-dimensional matrix layout to a linear memory model.

In summary, the RBS multiplication scheme walks along the Z-order curve successively applying the base-case multiplication to each $(N/b)^2$ block of memory it encounters, and reducing these with additions along the row axis. With this method (which actually obtains optimal cache complexity when implemented for matrix-matrix multiplication), we have removed the non-sequential data access penalty by more effectively mapping the two-dimensional matrix layout to linear memory, as well as eliminated the overhead of the recursive subdivision of the multiplication.

This scheme was implemented within the SALT framework, again using the supplied dot product operators to carry out the base-case multiplication. Since the RBS method is parameterized by the size of the base-case, we catalogued the performance as a function of this parameter on the AMD hardware using the GNU and Intel compilers. Using this information, we chose a base-case size, b, of 64x64 to compare the performance of the RBS scheme against the MKL and Eigen3 implementations as before.

From figure **??**, we can see that this implementation performs approxi-

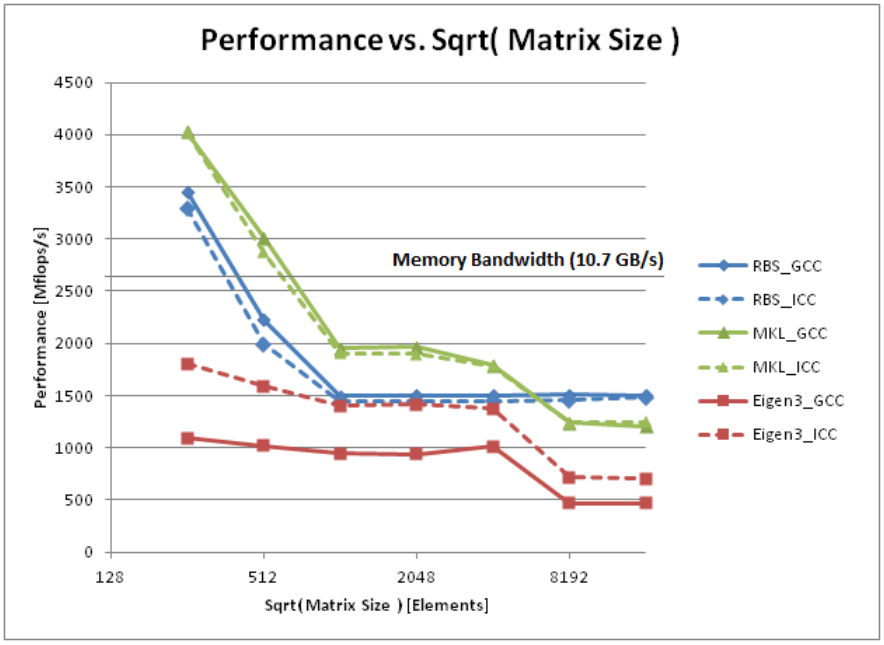

Figure 4: Performance metrics for the recursive block storage (RBS) method benchmarked against the MKL and Eigen3 implementations.

mately as well as the naive implementation for smaller matrix sizes (less than 1024x1024), but proves more efficient beyond that point by hitting a stable throughput plateau at 1,500 MFlop/s (over half of the processor's stated memory bandwidth).

Unfortunately, we were only able to take the measurements out to matrix sizes of 1GB before the Brutus job scheduling system disallowed larger allocations. This benchmark was repeated using the Merlin cluster at PSI and the performance plateau was qualitatively stable out to matrix sizes of 64GB.

To validate that the matrix reordering was more effectively exploiting the systems memory hierarchy, we turned to cache simulation using the Valgrind-based tool, Cachegrind. It works by running the executable inside a virtual machine that allows for every individual memory event, like L1 and L2 read/write cache-misses, to be counted and correlated with a machine instruction. As summarized in Figure **??**, we counted the cache misses at the lowest level of the multiplication (those resulting from calls to the SSE functions) for the RBS method (with a 64x64 element base case) and compared them to those caused by the naive scheme.

We can, in figure **??**, see that the RBS scheme greatly improves upon the naive multiplication by scaling L1 cache read misses almost directly with the matrix size. Additionally, we found that the RBS scheme results in zero L1 cache write misses for any of the tested matrix sizes, while this figure grows

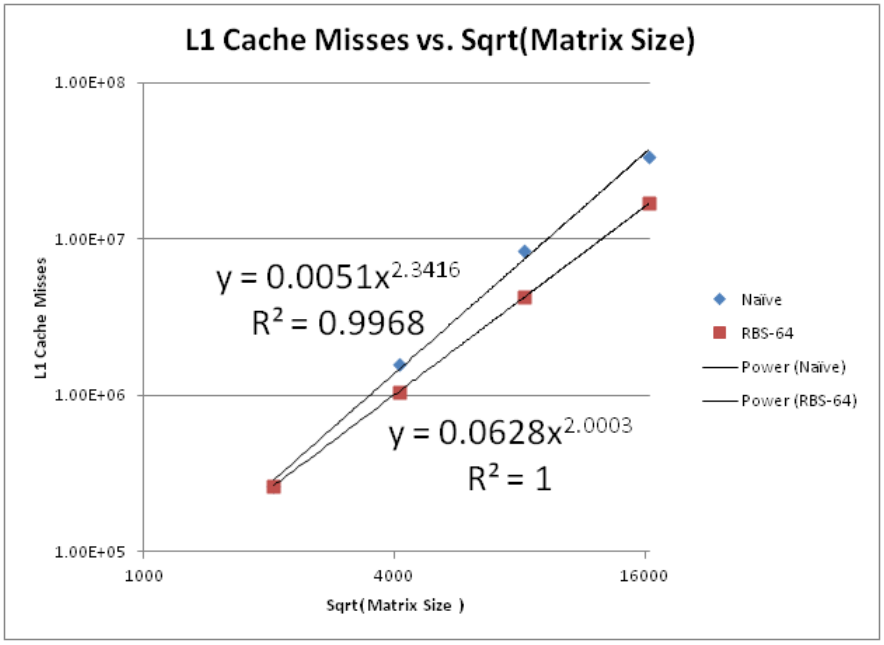

Figure 5: This plot shows the number of L1 cache misses as a function of the square root of the matrix size. The number of misses scales directly with the size of the matrix indicating near optimal cache efficiency.

directly with the matrix size for the naive implementation.

Having optimized the cache usage and improved throughput for large problem sizes, we turned our attention to the lower-than-expected performance on small and medium sized matricies. Firstly, it is obvious to see that the throughput for any matrix size is capped at the performance level of the dot product corresponding to the side-length of the RBS base-case. This is simply because the matrix multiplication, in its current form, consists of a series of smaller dot products. We observe this effect in both the naive and RBS cases where the performance ceiling is reached for matricies of sizes 128x128 and 256x256; although the simple vector dot product has much better throughput beyond these sizes, the additional data required for the matrix computation causes computations for larger systems to be bound by the L1-L2 memory bandwidth. Secondly, and perhaps less important, the RBS scheme requires $N(N/b-1)$ additional intermediate reductions (as compared to the naive method). While not particularly significant, this does represent an extra computational load.

5.2 Possible Improvements

To further improve performance for smaller problem sizes, we propose refining the base-case computation kernel for the RBS scheme from a naive matrix-

vector multiplication. Often, for cache-oblivious divide-and-conquer methods, implementers use column-major matrix storage in the base case, ostensibly, to reduce the number of reads required during the multiplication. To better leverage the structure of the SSE arguments, we propose a hybrid memory layout where SSE vector-width blocks of rows are stored column-major. This will result in one scalar-vector multiplication and one vector-vector addition per step. Instead of the current base case implementing N N-element dot products, the proposed method is analogous to performing (in the single precision case) N/4 dot products of 4N-element vectors, thus eliminating 75% of unnecessary reductions by performing them in parallel. Therefore, for a 128x128 base case, we could expect perhaps a 50% improvement in throughput based solely on the difference in dot product performance between 128-element and 512-element vectors.

Beyond this, one could amend the base-case by taking the base-case size down such that it fits entirely within the allotted registers. Since cache-oblivious algorithms will make optimal use of the memory hierarchy at any level, using named register variables, one could develop a matrix-vector kernel that operates entirely within the SSE registers and take the principle of maximizing data-locality to the extreme. It would be interesting to see if this could be done within the current SALT framework as it may require generalizing the loop templates.

6 Conclusions

Here we have presented a simple cache oblivious implementation of matrix-vector multiplication within the SALT framework. The result is an near-optimally cache efficient method that improves over Intel's MKL implementation by up to 20% for large linear systems (over 256 MB). We also note some possible improvements to this algorithm: firstly, by ammending the base-case to for more efficient register usage in the inner loop of the multiplication, and secondly, by restructuring the loop templates and using named register variables so as to extend the cache oblivious approach to the lowest level of the memory heirarchy.

References

[1] T. Veldhuizen. 1995. *Expression Templates*. C++ Report 7.

[2] J. Progsch, Y. Ineichen, and A. Adelmann. 2012. *A New Vectorization Technique for Expression Templates in C++*. AJUR, June 2012. arxiv: 1109.1264v1.

[3] H. Prokop. 1999. *Cache-Oblivious Algorithms*. Massachusetts Institute of Technology, (Cambridge, MA).

A Addtional Performance Results

A.1 Single Precision Benchmarks

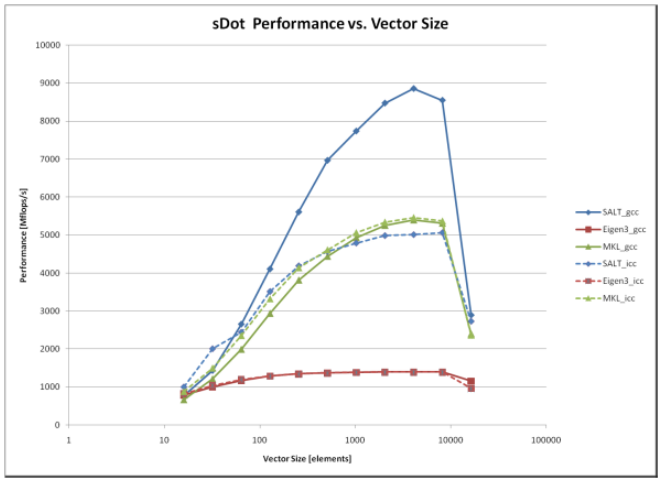

Figure 6: Performance metrics for the single precision vector dot product on the AMD Opteron 8380 benchmarked against the MKL and Eigen3 implementations.

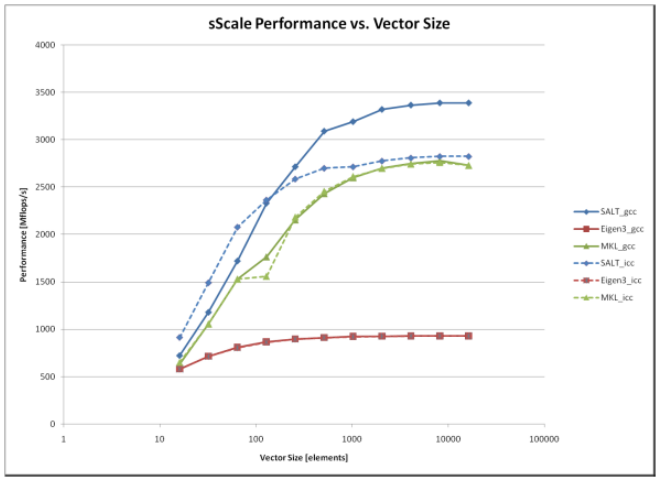

Figure 7: Performance metrics for the single precision vector in-place scaling on the AMD Opteron 8380 benchmarked against the MKL and Eigen3 implementations.

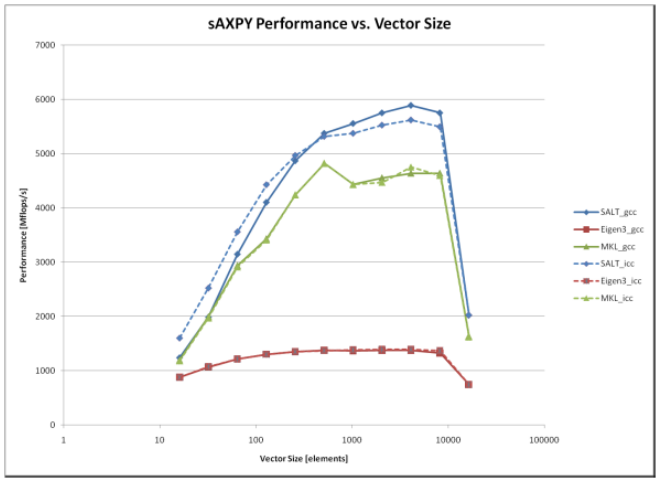

Figure 8: Performance metrics for the single precision vector AXPY operation on the AMD Opteron 8380 benchmarked against the MKL and Eigen3 implementations.

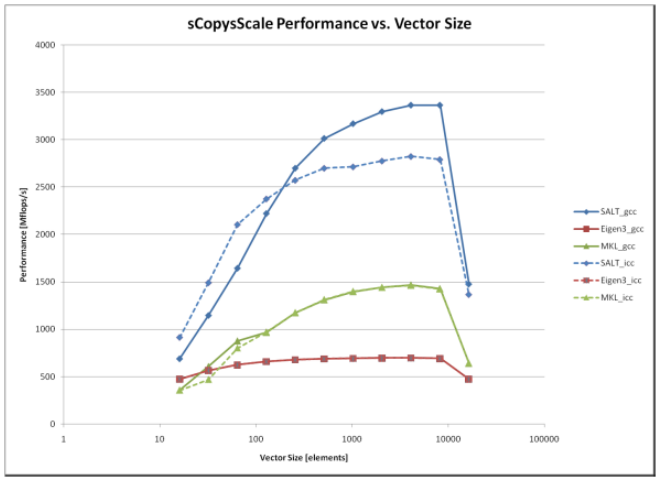

Figure 9: Performance metrics for the single precision vector out-of-place scaling on the AMD Opteron 8380 benchmarked against the MKL and Eigen3 implementations.

A.2 Double Precision Benchmarks

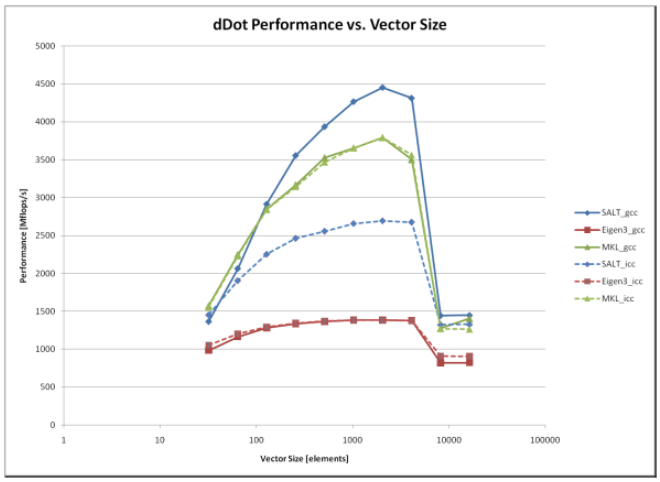

Figure 10: Performance metrics for the double precision vector dot product on the AMD Opteron 8380 benchmarked against the MKL and Eigen3 implementations.

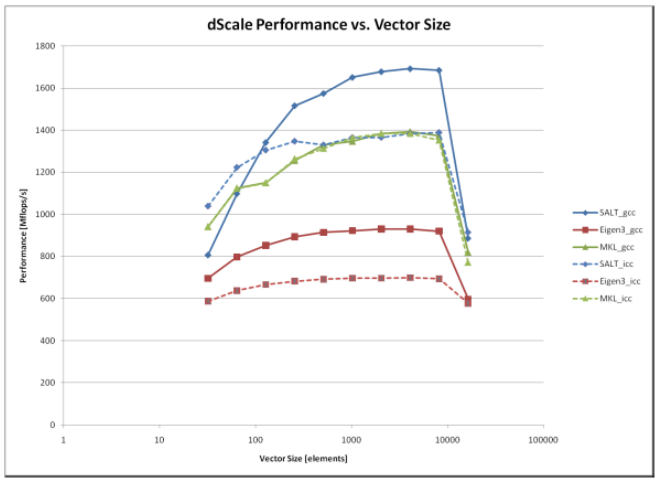

Figure 11: Performance metrics for the double precision vector in-place scaling on the AMD Opteron 8380 benchmarked against the MKL and Eigen3 implementations.

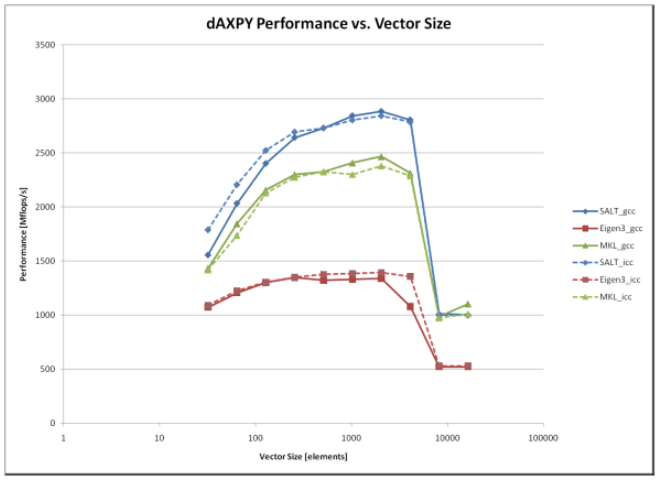

Figure 12: Performance metrics for the double precision vector AXPY operation on the AMD Opteron 8380 benchmarked against the MKL and Eigen3 implementations.

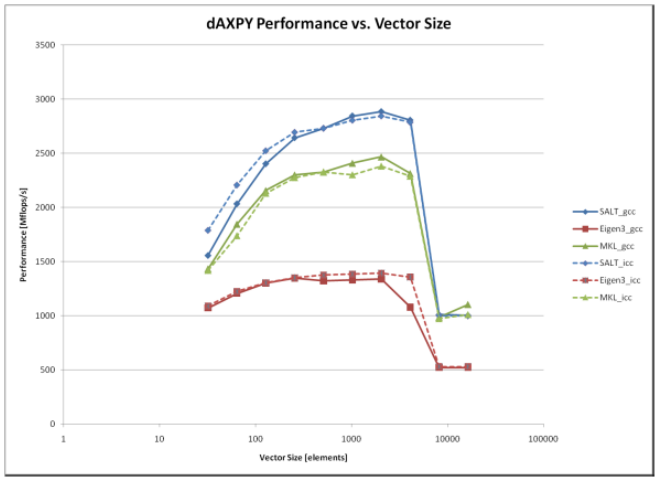

Figure 13: Performance metrics for the double precision vector out-of-place scaling on the AMD Opteron 8380 benchmarked against the MKL and Eigen3 implementations.