Flexible High-Performance Matrix Multiply via a Self-Modifying Runtime Code

Greg Henry
Intel Corp.
Computational Software Laboratory
5350 NE Elam Young Pkwy, Bldg EY2-03
Hillsboro, OR 97124-6461
greg.henry@intel.com
503-696-3878

December 5, 2001

Abstract

With the advent of architectures with multiple levels of memory hierarchies, the quest for high performance implementation of commonly used linear algebra operations has become progressively more complex. Fortunately, recent research has shown that a large class of such operations can be implemented in a relatively portable fashion provided a high-performance matrix-matrix multiplication routine is available for a given architecture. More recently yet it has been shown that the matrix-matrix multiplication itself can be implemented in a portable fashion provided a high-performance "inner kernel" is available. This inner kernel performs the matrix-matrix multiplication of small submatrices in such a way that data movement between caches and registers is carefully amortized. In recent years, approaches like those pursued in the ATLAS and PHiPAC projects have used automatic generation, at build time, of matrix-matrix multiply inner kernels. In this paper, we present the idea of using an inner kernel that, at run-time, is self-modifying. The advantages of using a self-modifying inner kernel at run-time include performance gains, minimizing the amount of code to maintain, and a reduction in the memory footprint of the executable. In addition to a thorough discussion of the issues affecting the inner kernel we present performance results for an implementation on the Intel Pentium (R) III processor.

1 Introduction

Architectures with memory hierarchies help overcome the problem that often data movement is slow compared to computation. If data brought into the low end (registers, cache) is reused to a significant degree, then the data traffic between the layers of the hierarchy is decreased. In linear algebra, programmers hope to write portable and modular code that can exploit a memory hierarchy. The BLAS (Basic Linear Algebra Subroutines) [5] allow this efficiency and portability, since new machines are often released with an optimized BLAS library. The BLAS have different levels of data reuse, with the most significant being matrix-matrix multiplication. Often, linear algebra developers will try to cluster several matrix updates to apply a bunch of them simultaneously, in order to pick a BLAS routine with greater reuse. A modified "block" algorithm results. Linear Algebra libraries like LAPACK [2] and FLAME [7] that employ block algorithms have replaced older libraries such as LINPACK [4] and EISPACK [13] for the same dense linear algebra problems. On a machine with hierarchical memory, a blocked algorithm that calls the BLAS is often preferable, even if it involves marginally more arithmetic.

Arguably, the significant BLAS kernel is matrix-matrix multiplication [6] or `GEMM` (GEneral Matrix times Matrix). `GEMM` is usually the fastest and simplest of all the BLAS. Furthermore, all of the faster BLAS can be written in terms of `GEMM` ([1, 9, 12].) It has also been observed that `GEMM` itself can be efficiently implemented in terms of an inner kernel which performs a matrix-matrix multiplication on subblocks of the matrix [1, 3, 14]. Our examples are with the double precision (real*8) version of `GEMM` called `DGEMM` [5]. There have been numerous papers and techniques based on optimizing `DGEMM` (for example, [10, 1, 14, 3, 8].) All of these papers share a single thing in common. They all implement outer loops around a fast inner kernel. ATLAS [14] and PHiPAC [3] attempt to use the target computer and its compiler to generate a fast inner kernel at build time. Henry [10] tried to use alternate data structures to make the inner kernel run even faster. ITXGEMM [8] attempted to use mathematics to model the memory hierarchy in order to find the most efficient blocking strategy, and had the interesting feature of treating each level of the memory hierarchy as a separate problem. `DGEMM` itself is defined by the general matrix-matrix multiplication of

$$C = \alpha \mathtt{op}(A)\mathtt{op}(B) + \beta C$$

where $\mathtt{op}(X)$ equals X or X^T, $C \in \Re^{m \times n}$, $\mathtt{op}(A) \in \Re^{m \times k}$, and $\mathtt{op}(B) \in \Re^{k \times n}$.

In this work, we concentrate on the innermost kernel, so it is only necessary to consider a single loop order, and a single set of blocking parameters. We shall denote m_b, n_b, and k_b as the blocksizes for m, n, and k respectively. All of the above `GEMM` optimization approaches depend on somehow obtaining a fast inner kernel similar to $C \leftarrow A^T B + C$ with blocksizes m_b, n_b and k_b replacing m, n and k. Determining how to block `DGEMM` outside the innermost kernel helps to dictate what percentage of the innermost kernel performance is obtainable for a larger matrix. If the innermost kernel runs slowly, suffice to say that the resulting `DGEMM` will run slowly as well. We refer the reader to Gunnels et. al. [8] for one example of blocking outside the inner kernel.

Since this paper will detail how to make fast performing inner kernels, we end our introduction with a brief description of exactly what makes a fast performing inner kernel. All of our modifications will be based on the simple design illustrated here. Eventually, we will show how making the inner kernel self-modifying enhances performance and code reliability.

In practice, on the Intel Pentium ® III processor, ideal results were observed when the innermost loop was with $m_b = 4$, $n_b = 1$, and $k_b = 64$. This makes sense, because the innermost kernel is blocked for the registers, and there are only 8 available double precision registers available on the Intel Pentium III processor. Since the cache line is of size 32 bytes, it is convenient to block for at least $m_b = 4$ doubles. One could also write fast kernels with $m_b = 2$ and $n_b = 2$ (making certain to call it an even number of times).

A key observation is that the $m_b = 4$ and $n_b = 1$ innermost loop is best when unrolled in the k_b dimension. There are several reasons for this. The most obvious of which is avoiding loop overheads. But there are also subtle reasons involving reducing the number of instructions during the section of code near the stores and increasing the number of instructions to include some reordering instructions elsewhere. The only way to set up a loop on this architecture to have a different structure at different places is to unroll it and set the structure manually.

Final performance turns out to be based on some of the following fine details:

- Unrolling the loop
- Simplifying the instructions around the store
  - In-lining the β parameter
  - Minimizing data movement (if k_b is less than the ideal size, try to pick k_b as close to the ideal size as possible.)
- Choosing a row blocking (on m) to be at least 8 when k_b is small enough. (Choosing more than 8, depending on how large k_b is, may run into instruction cache problems since everything is unrolled. A general rule of them on the Intel Pentium III processor is a load is typically 6 bytes, and a multiply and add are 2 bytes each, so not exceeding the 16 Kbyte L1 instruction cache generally means making certain things are not unrolled past 8×64.)

All our inner kernels will be based with the above facts in mind.

We introduce the idea here to use an inner kernel that, at run-time, is self-modifying. We show in Section 2 how the number of variations of possible inner kernels otherwise could lead to a code explosion. We propose to address this problem with a self-modifying kernel. Our self-modifying kernel leads to advantages, including minimizing the amount of code to maintain, and reducing the memory footprint of the executable. It also leads to performance advantages, in both uniformity and peaks, and those are detailed in Section 3. We end the paper with some conclusions in Section 4.

2 Many Inner Kernel Cases

2.1 Code Explosion

Given the information in the previous section, it is possible to construct a single well optimized inner kernel for `DGEMM.` The problem arises when one realizes that to support all cases of `DGEMM`, there may arise a need for many such kernels.

For example, suppose we decide the innermost kernel for our optimal `DGEMM` is with $m_b = 8$ (explicitly unrolled twice such that really $m_b = 4$) $n_b = 1$ and $k_b = 64$. What should happen if someone should want to solve the problem with $m = 4$ or $k < 64$ or $\beta = 0$? We now detail what is typically done to resolve this problem.

A library will typically write kernels for $k = 64$, and then $k = 32$, $k = 16$, $k = 8$, $k = 4$, $k = 2$, $k = 1$, and then should a `DGEMM` call require say $k = 63$, solve the problem as follows: first do a $k_b = 32$, then $k_b = 16$, then $k_b = 8$, then $k_b = 4$, and then finish the problem by iterating over $k_b = 2$ or $k_b = 1$. Sometimes the bottom end (small k_b) of the spectrum will be replaced by high-level compiled code, or codes that are not unrolled. There are two strong shortcomings to this approach:

1. The Mflop rate performance for $k_b = 4$ is much worse than $k_b = 32$, and in general any time one uses a smaller k_b, the data reuse decreases, and the performance drops. Even though the $k_b = 4$ represents 1/8 the work, it may also run at 1/3 the speed, thus slowing down a $k = 36$ by ten to thirty percent or so. [1] In fact, with this approach, even though $k = 36$ has better data re-use one typically observes it to run ten to thirty percent slower (in terms of Mflop performance) than $k = 32$.

2. The border cases may not have the same level of optimization, either because it is compiled code, or not unrolled completely, or the developers did not have the time to pay too close attention to all the special cases.

Of course, from a library developer's viewpoint, this is bad news because one has just gone from one major inner kernel to $\log(k_b)$. Sometimes even more kernels arise depending on the performance curve over k. For example, if $k = 64$ runs twenty percent faster than $k = 32$, one may feel the need to develop even more inner kernels, say a $k = 48$, just to help fill in some gaps.

Granted, this is not a concern for when the optimal kernel does not need to be unrolled. However, there are other concerns besides unrolling.

One needs to consider values of β. As mentioned earlier, inserting a arbitrary multiplication of β may be in the critical path of the stores, and slow down the resulting code compared to the case where β is zero or one. Now we have three cases: β arbitrary, β zero, and β one. Note that since $\beta = 1$ is the most common case, it may be a performance penalty to simply use the β arbitrary case that runs slower. As an added complication, `DGEMM` specification requires that when β is zero, that the matrix be overwritten and not that it simply get multiplied by zero before the add. This may seem like the same thing, until one remembers that C may start off the computation filled with NaNs (not floating point numbers), and be sent into a `DGEMM` call with $\beta = 0$ and expect that the result be legitimate values. Simply multiplying a NaN times zero does not yield zero, so these are not the same thing.

Of course, β and k_b are not the only variables. The precision of the routine (real*8 as opposed to real*4) makes a difference, even if the inner kernel approach is identical between the two routines! The problem is exasperated when one considers picking m_b optimally, or perhaps choosing α may end up making a difference in performance (clearly this is architecturally dependent just as β sensitivity is). Another variable might be whether or not the data is aligned. Changing the instructions to deal with unaligned data is minor, but can easily double or triple the number of inner kernels one would want in a perfect world. Even if one implemented dozens of kernels

[1] In greater detail, if the $k = 32$ kernel solves a problem with a fixed $m, n = 1000$ in 8 seconds (which translates to a rate of 8 Mflops), there is no reason to believe that the $k = 4$ kernel will maintain the same Mflop rate and solve a problem with the same $m, n = 1000$ in one second. Because the data re-use for $k = 4$ is not as good, it might instead take 4 seconds (2 Mflops). Now the $k = 36$ kernel uses a $k = 32$ and $k = 4$ and therefore takes $8 + 4 = 12$ seconds total to do $m = n = 1000$, which represents 6 Mflops. This is 33 percent slower than the $k = 32$ kernel, even though theoretically we would expect $k = 36$ to run faster and be more efficient than $k = 32$.

to match the most commonly encountered cases, running a slightly different case may prove to run significantly slower than a library which was specifically optimized for that precise case. This is the reason most libraries have some performance hiccups. The only systematic way of eliminating most of these hiccups in the past has been to implement more special cases (code explosion).

Our goal is to present a self-modifying library that will automatically re-adjust for a large number of cases that would be too time consuming to implement by hand.

Consider the following list, which represents a number of kernels for just one Intel processor.

- The size of the k_b loop (possibly 64, 32, 16, 8, 4, 2, 1)
- The presence of α (possibly one, negative one, arbitrary)
- The presence of β (possibly one, negative one, zero, arbitrary)
- Whether the data is aligned or not (8 choices: yes/no for A, B, and C)
- The size of the m_b loop (possibly 4, 8, 12)
- The precision of the routine. (one of 4 possibilities)
- Conversion between A and A^T (one of 2 possibilities)
- Conversion between B and B^T (one of 2 possibilities)

Each of the above choices represents several different implementations. The total number of possible inner kernels grows geometrically with the number of interesting choices, which seems to increase with each processor release. The current observations could lead to

$$7 \times 3 \times 4 \times 8 \times 3 \times 4 \times 2 \times 2 = 32256$$

possibilities. Realistically, one might chose the 100 most likely possibilities, and simply expect some cases that run less optimally.

But the most important observation is that *each implementation of a given choice differs only marginally, in some predictable fashion, from other implementations of the same choice.*

To date there have been two possible solutions, although one solution is fairly recent.

1. The typical approach: Bite the performance bullet and simply implement some subset of cases
2. The let your computer go code crazy approach: Use the computer to generate the cases for you and enumerate some large subset of them [3, 14] [2]

[2]Even when it is the computer that is coding these routines, such as in ATLAS, tradeoffs are made so that only a large subset of the 32000 routines are used.

The second case, using an approach such as taken in ATLAS [14], is only adequate if you can depend on the compiler to generate nearly optimal code. Although they claim fantastic results, our observations on the Intel architecture is that the compiled performance of ATLAS is typically slower (usually by roughly 15% or more) than what is possible. The dependence on the compiler is yet another concern; even though compiler technology continues to improve, it does not improve as quickly as new processors come out. Obviously, the ATLAS authors must recognize this truth as well: The latest version of ATLAS (as of this writing, version 3.2) for the Intel Pentium III processor require user-contributed hand-tuned assembly language for the Intel SSE ® instructions, even though the Intel Pentium III processor has been in mass production for considerable time. The primary ATLAS author, Clint Whaley, distributed a note to the entire ATLAS community on September 13, 2001, in which he states:

> Gcc 3.0 and RedHat 7.1's 2.96-85 gcc are absolute disasters for ATLAS... chances look fairly good that from now on, ATLAS will simply have to require an old gcc in order to get good performance.

This only illustrates further our worry about having a library performance depend entirely upon a changing compiler. For the purposes of this paper, we used Mr. Whaley's binaries built from an earlier compiler which did not exhibit this problem.

Enumerating all the possibilities, however, leads to a tremendous code explosion, even when its the computer that is doing the coding. Nevertheless, this is an interesting idea: teach the computer how to write C code.

2.2 Code Implosion

Our approach is more fundamental: teach the computer how to modify its own object (machine language) code at run time to dynamically create the inner kernels for a math library that are specifically optimized for the given input. In other words, one gains the performance benefit of having literally hundreds if not thousands of specially tuned kernels at ones disposal, but the user only has to write a single routine!

The success of the self-modifying library is that instead of writing all the different combinations of implementations for all choices, one teaches the computer how to manipulate one kernel implementation into another. One programs the algorithm to change a $k_b = 17$ size loop into a $k_b = 15$ size loop. For each choice (size of the k_b loop, α, β, whether the data is aligned or not, etc..), a **default** option is implemented. Then each choice is addressed independentally, and the algorithm we suggest is taught the necessary manipulations to modify the default option into any of the other options.

We illustrate this with an example. Consider the choice of whether the C data is aligned or not. On many, but not all, modern architectures, the alignment of the data can have a serious impact on performance. For the Intel Pentium III processor, it suffices that the data be 8-byte aligned. For the Intel Pentium(R) 4 processor, if the data is 16-byte aligned, significant performance enhancements can be made. Because there are significant performance enhancements for aligned data on the Pentium 4, the inner kernels should be written to assume the data is aligned. The software developer then has an obvious choice: duplicate every kernel for the case where C is not aligned, or simply copy C into an aligned buffer, suffering a performance penalty (possibly) for that copy. For large matrices, the cost of the copy is immaterial, but for an outer product update where k is small and m and n are large, and C is unaligned, this cost becomes as expensive as the matrix multiply itself.

The self-modification runtime technique would address this example as follows. The default option for the choice of whether C is aligned or not would be aligned, because that is fastest. The difference in the object code between an aligned load and an unaligned load is only one byte (the same can be said of stores). The default object code can be accessed as a data array using standard C pointer manipulation. Once as a data array, it can be copied and manipulated. The correct location for the loads and stores of C can be found merely as a function of any inputs (m, k, etc..) that are unrolled. The byte(s) in question can be changed to the new values, and the resulting object code suddenly works with unaligned data.

In practice, the same modification techniques are applied to each and every choice, except when the inputs happen to match the default options. By treating each choice separately, the problem becomes tractable. The result is the way a human typically thinks about a problem: at a high level on how each problem differs. A human typically understands that there are many combinations of possibilities, but does not bother to pay learn or pay attention to every single one, and instead just learns how to manipulate one into another.

Instead of having 80, 100, or 32000 different kernels, or some subset and the possibility of never having all cases of interest being optimized, one develops a single routine and teaches the computer the same human understanding of what changes one code into another. The resulting manipulations are so fast they can be done at run time, when some of the information becomes available. Instead of requiring the user to have the space to include the possibly hundreds of routines just in case one of them might be useful, this is more a "just in time" type of self-modification that can be done a runtime.

Let us now consider one of the more interesting choices: the size of the unrolled k_b loop. We call our single routine `single_inner_kernel` for computing $A^T B$.

Suppose `single_inner_kernel` is optimized for $m_b = 8$ (two sets of $m_b = 4$) and $k_b = 64$. Suppose during run time a routine is passed in where $k = 5$, or perhaps where $k = 5$ is necessary. At the start of the `DGEMM` call, it should be possible to determine in advance what routines will be necessary for the duration of that `DGEMM` call. (One might want to create multiple copies of our modifiable kernel for this reason.) We recommend creating the routines at the start of the `DGEMM` call because an inner kernel may end up being called many times, and there is no reason to remold it into the same thing each time.

Suppose one wanted to stop the computation after the fifth unrolled block so that our becomes a kernel on $A^T B$ with the leading dimension of A still 64, but unrolled to $k = 5$. There are really two choices:

1. **JUMP Modification** Overwrite the next block with minor clean-up followed by a jump to the end of the loop.

2. **Rewrite Modification** Overwrite the next block with minor clean-up followed by the end of the loop (thus shortening the binary object array).

The second solution is fastest, but involves changing the most object code. The first solution shall be discussed in a bit more detail because of the wonderful observation that only a few bytes of data need to be modified.

Typically there may be some cleanup associated with either of these. It is our experience that this is extremely small.

Obviously, a loop that assumes $\alpha = 1$ and $\beta = 0$ is significantly more simple than a loop that assumes nothing about α and β. Because the store to C is usually a bottleneck, removing these extra instructions makes for a much faster routine. That is, the bottleneck of the memory access is more significant than just the extra instructions. Removing them then yields a double benefit, and the resulting performance is optimal.

To complete the picture, suppose $m = n = k = 300$ and $\alpha = 2.3$ and $\beta = 4.9$. One might need 3 inner kernels to pull this off effectively:

- One tuned for β arbitrary and $k_b = 64$
- One tuned for $\beta = 1$ and $k_b = 64$
- One tuned for $\beta = 1$ and $k_b = 44$ (300 mod 64).

Usually, one can resolve any one `DGEMM` call with up to 4 inner kernels. Using these self-modifying techniques, one can create all four of these inner kernels once at the start of the `DGEMM` call. This amortizes their already extremely inexpensive self-modification time with the time it takes to do the entire `DGEMM`. The jump modification necessary to change the $k_b = 64$ code into a $k_b = 44$ code was only the writing of 18-bytes. Over the cost of an entire `DGEMM`, the manipulation of those 18-bytes wasn't even observable (it turned out to be less than other noises between runs.)

It is for this reason that we recommend creating multiple copies of the same array for several different modifications at any one given time. In our Intel Pentium III implementation, we keep track of each modification that was done, and have a "least recently used" caching policy on the inner kernels. If a request is made for a routine that has already been created, the pointer to a function (really the appropriate array) gets passed back to the `DGEMM` level and no modifications are required at all. To prevent two different threads from overwriting the same array, a simple "test and set" mechanism is implemented, so only one thread can have modification rights to an array at a time, however several threads could go ahead and use the array. However, we have also experimented with rewrite modifications that create kernels as they go, and so never have to worry about a caching policy. Although this was more costly to do, it still was on the order of manipulating an extra 5K array next to the cost of manipulating several large matrices.

3 Experiments with the New Approach

In Figure 1, we see `DGEMM` performance as k varies on an Intel Pentium III processor running a 2.2-12 Linux kernel at 600 Mhz.. The problem size was fixed at $m = n = 960$ and this was the `'NT'` case of `DGEMM`: $C \leftarrow A_{960\times k}(B_{960\times k})^T$. Each data point was run five times and the best was taken. Before each run, all the data was reinitialized and a one megabyte array was read so that the caches would hopefully always be in the same state for scientific consistency. To further test this, each library was run against itself, and the variance between the timings done in this experiment

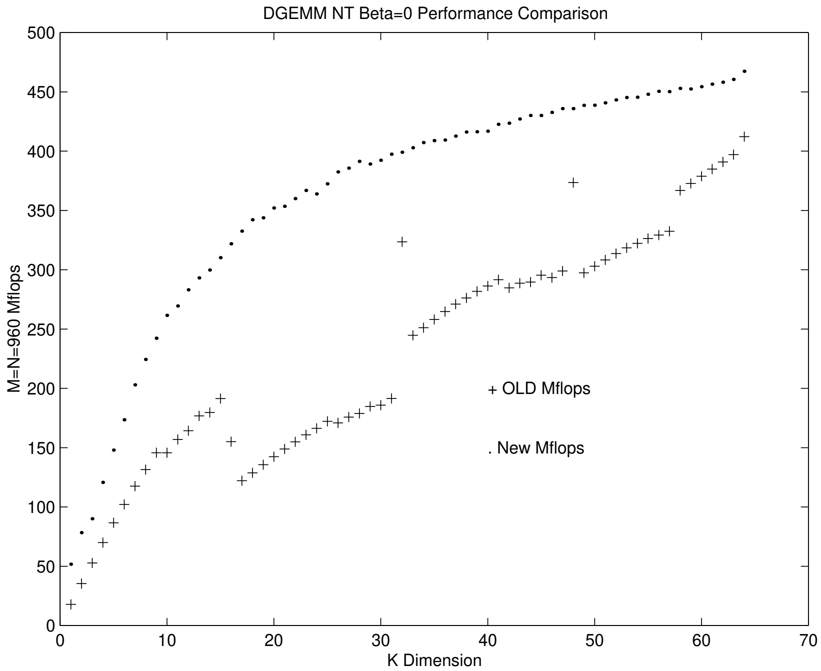

Figure 1: DGEMM $C \leftarrow AB^T, \beta = 0, m = n = 960$ Performance Comparison

was always less than half a percent. We see an older library used internally at Intel, which has fantastic peak performance, have some peaks and valleys typical of most DGEMM implementations. The older library was chosen because it was the most competitive library we could find. Later in the paper we do a more "apples to apples" comparison, as well as a contrast directly against ATLAS, but we strongly feel that a library approach is only justified when compared successfully against the state-of-the-art. The "new" library represents a single implementation of the idea represented in this paper. The same source code and path was used for the "new" library for all runs for the purpose of focusing the results on the impact of the self-modification. The leading dimensions of the matrix were fixed at 960 for all runs to circumvent the obvious optimization of avoiding a copy (which again illustrates the outer blocking techniques and not the inner kernels.) Notice the strange performance behaviors for the older library when K is 16, 32, 48. This is because those are cases hand tuned and the DGEMM algorithm implemented takes a different path at those junctures.

If you compare to the self-modifying library, you see a much more uniform and smooth picture. Keep in mind that this figure was based on a single kernel and a single C wrapper (which ignored k) blocking around it. Notice also that the performance is monotonically increasing as k increases, which is exactly what the data re-use theory tells us should happen.

Because the DGEMMs in Figure 1 have different peaks, we present the same data in terms of a ratio of peak speed in Figure 2. Here, we define peak speed as not the peak speed DGEMM is capable of (in this case, 600 Mflops), but the peak observed performance for that library in the set of runs. Therefore, both curves are normalized so that at least one of the data points for each curve is the maximum ratio of 1.0.

Figure 3 is a duplicate of Figure 2, but with a more common case of DGEMM, $C \leftarrow C + A_{mxk}B_{kxn}$.

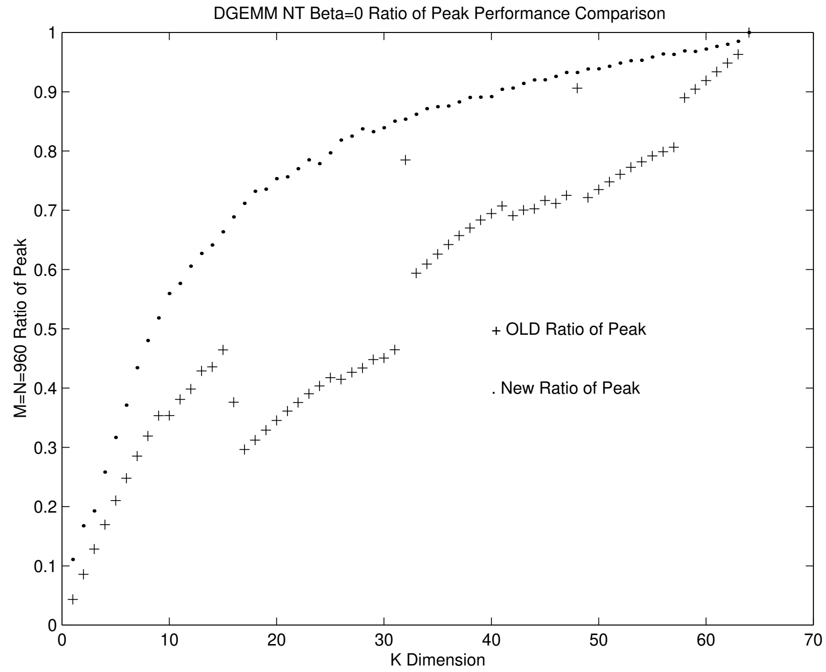

Figure 2: DGEMM $C \leftarrow AB^T, \beta = 0, m = n = 960$ Ratio of Peak Comparison

Here, we see the older library has spent more time optimizing critical cases like $k = 32$ and $k = 48$, and that effort has paid off in creating very fast kernels. However, it is not clear whether or not other blocking factors (like m_b or n_b) may be different as well for those two outliers. Recall that we intentionally used the exact same inner kernel and outer blocking for each of the performance figures. Presumably, the older library's odd behavior around $k = 16$ is a mistake, but notice this is another argument for a uniform performance approach. Incidentally, the peak performance of the older library and the other approach were within one percent at $k = 64$ for this one case of DGEMM. For other cases of DGEMM, that was not the case, possibly because less kernels were available to the older library compared to all the self-modifying possibilities.

Finally, one claim in this paper is that self-modifying the unroll factor of k_b is a useful technique compared to the $\log(k_b)$ approach common in the literature. The best way to prove this point is to create a DGEMM that uses the exact same inner kernels for the log points, but uses sums of these to determine the intermediate values. That is, when $k_b = 63$, this computes that by doing the following subproblems: $k_b = 32, k_b = 16, k_b = 8, k_b = 4, k_b = 2, k_b = 1$. As mentioned before, a hybrid method that does $k_b = 7$ (or some other "cut off" point) via a non-unrolled loop is bound to be faster. This represents just one more level of complication library writers normally worry about. In addition to implementing $\log(k_b)$ inner kernels for each case, a good library might also have a few good border cases written to use for a good "cut off" technique (unless the method presented here is used instead). Our performance results can safely ignore the "cut off" techniques, since an argument for breaking away from the $\log(k_b)$ technique for some arbitrary number like $k_b = 8$ does not apply to any data points that are multiples of 8, which we provide. We elected to run all data points from $k = 1 \ldots 64$, just as we did for the other figures.

The $\log(k_b)$ approach and the new approach for this experiment result in the same thing when

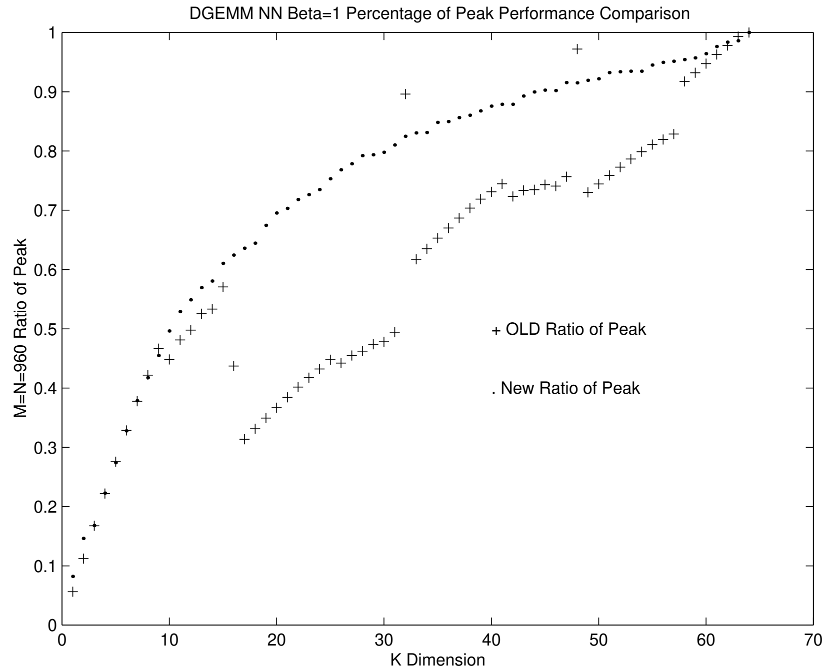

Figure 3: DGEMM $C \leftarrow C + AB, \beta = 1, m = n = 960$ Ratio of Peak Comparison

k_b is a power of 2. The results are in Figure 4. Figure 4 had the exact same set up as Figures 2 and 3, and was the exact same DGEMM case given in Figure 3. Given the scattered nature of the results of the $\log(k_b)$ approach, it is clear that the $\log(k_b)$ approach on this problem is less than adequate, and a bit surprising given that this is the approach most commonly used.

Additionally, we must compare at least once against ATLAS' approach for handling the extreme variability of DGEMM by enumerating all possibilities at the cost of huge computer code generation. Figure 5 does just this on an Intel Pentium III processor. This figure illustrates why we did not use ATLAS comparisons elsewhere in the paper: the performance of ATLAS was too slow to focus attention to advantages of this new approach. On the plus side, we see how ATLAS also avoids many of the dips and valleys, through their "brute force create a routine for every case" type of strategy.

4 Self-Modifying Libraries and Conclusions

The technology presented here can be used in any library, not just DGEMM, with the following feature-set:

1. The library is CPU-intensive, looping over the same instructions (the ones that will be modified) a significant amount.

2. The performance of the library is sensitive to multiple user-configurable inputs which change the resulting implementation in marginal but performance notable ways.

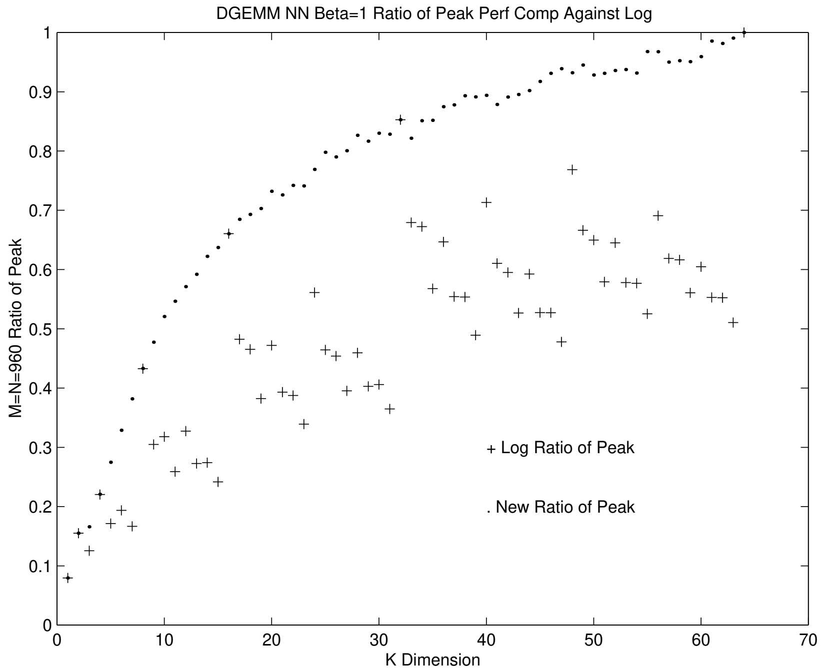

Figure 4: DGEMM $C \leftarrow C + AB, \beta = 1, m = n = 960$ Ratio of Peak Comparison Log vs. New

In the past, people have depended on heuristic algorithms to determine the best hybrid choices between many different inner kernels and techniques. The computer serves to evaluate the heuristic, the human programs the choices. With this technology, the computer programs itself, not at a high level and at build time, but at a machine level and dynamically during run time. The heuristics are eliminated.

The unique elements of this approach are both the computer programming itself at a machine level and that this occurs dynamically during run time. The benefits of this approach is that now there is a single inner kernel to worry about instead of myriads of routines, and one effectively eliminates the performance "hills and valleys" standard of mathematical software libraries, by allowing the computer to make every case a "special case" tuned for optimal performance.

One could apply these techniques to get superior performance, to get uniform performance, to eliminate the need for special cases, or to minimize code explosion. The pleasant notion is that in our case, we got all of these.

Acknowledgements

Peter Tang and Robert van de Geijn provided detailed feedback on the initial draft.

References

[1] Agarwal, R.C., Gustavson, F.G., Zubair, M., *Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms*, IBM Journal of Research and Development, Vol. 38, No. 5., pp. 563–576,

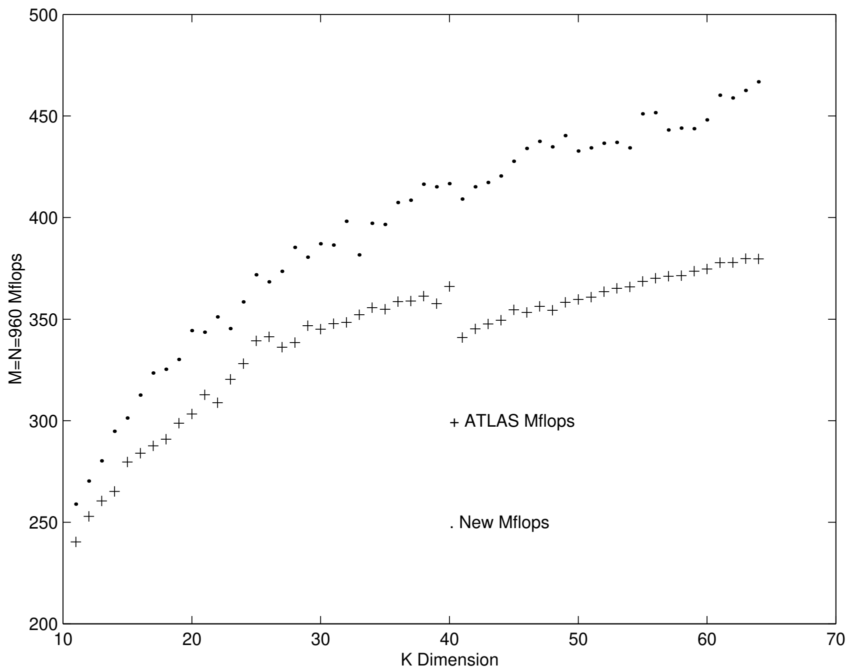

Figure 5: `DGEMM` $C \leftarrow C + AB, \beta = 1, m = n = 960$ Performance Comparison

Sept. 1994

[2] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorenson, D., LAPACK Users' Guide, SIAM Publications, Philadelphia, PA, 1992

[3] Bilmes, J., Asanovic, K, Chin, C.W., Demmel, J., *PHiPAC: a portable, high-performance, ANSI C coding methodology*, In Proceedings of the International Conference on Supercomputing, ACM SIGARC, July 1997

[4] Dongarra, J.J., Bunch, J.R., Moler, C.B., Stewart, G.W., LINPACK Users' Guide, SIAM Publications, Philadelphia, PA, 1978

[5] Dongarra, J. J., Du Croz, J., Hammarling, and Duff, I. S., 1988, *A set of Level 3 basic linear algebra subprograms*, Report AERE R 13297. Oxford: Computer Science and Systems Division, Harwell Laboratory.

[6] Golub, G., Van Loan, C., Matrix Computations, 2nd Ed., 1989, *The John Hopkins University Press.*

[7] Gunnels, J., Gustavson, F., Henry, G., van de Geijn, R., *FLAME: Formal Linear Alegra Methods Environment*, To appear in ACM Transactions on Mathematical Software.

[8] Gunnels, J., Henry, G., van de Geijn, R., *A Family of High-Performance Matrix Multiplication Algorithms*, Proceedings of International Conference on Computational Science (ICCS) May 2001, Part 1, page 51, Alexandrov, V.N. et. al. editors

[9] Gustavson, F., Henriksson, A., Jonsson, I., Kågström, B., Ling, P., Applied Parallel Computing, Large Scale Scientific and Industrial Problems, *Superscalar GEMM-based Level 3 BLAS – The On-going Evolution of a Portable and High-Performance Library*, Springer-Verlag Publ., pp. 207–215, 1998

[10] Henry, G., *BLAS based on Block Data Structures*, Theory Center Technical Report, CTC92TR89, 2/92

[11] Henry, G., *Improving Data Re-Use in Eigenvalue-Related Computations*, Ph.D. Thesis, Cornell University, January 1994

[12] Kågström, B., Ling, P., Van Loan, C.F., *GEMM-based Level-3 BLAS: High Performance Model Implementations and Performance Evaluation Benchmark*, ACM Trans. Math. Software, Vol. 24, No. 3, pp. 268–302, 1998

[13] Smith, B., Boyle, J., Dongarra, J., Garbow, B., Ikebe, Y., Klema, V., Moler, C. *Matrix Eigensystem Routines - EISPACK guide, 2nd Ed.* Lecture Notes in Computer Science 6 Springer-Verlag, 1976

[14] Whaley, R.C., Dongarra, J.J., *Automatically tuned linear algebra software*, Proceedings of SC98, November 1998