Center for Embedded Computer Systems
University of California, Irvine

Improving the Accuracy of High Performance BLAS Implementations using Adaptive Blocked Algorithms

Matthew Badin, Paolo D'Alberto,
Lubomir Bic, Michael Dillencourt, Alexandru Nicolau

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

mbadin@uci.edu

CECS Technical Report 11-05
April 15, 2011

Improving the Accuracy of High Performance BLAS Implementations using Adaptive Blocked Algorithms

Matthew Badin Paolo D'Alberto Lubomir Bic
Michael Dillencourt Alexandru Nicolau

April 2011

Abstract

Matrix multiply is ubiquitous in scientific computing. Considerable effort has been spent on improving its performance. Once methods that make efficient use of the processor have been exhausted, methods that use less operations than the canonical matrix multiply must be explored. Combining the two methods yields a hybrid matrix multiply algorithm. Hybrid matrix multiply algorithms tend to be less accurate than the canonical matrix multiply implementation, leaving room for improvement. There are well-known techniques for improving accuracy, but they tend to be slow and it is not immediately obvious how best to apply them to hybrid algorithms without lowering performance. Previous attempts have focused on the bottom of the hybrid matrix multiply algorithm, modifying the high-performance matrix multiply implementation. In contrast, the top-down approach presented here does not require the modification of the high-performance matrix multiply implementation at the bottom, nor does it require modification of the fast asymptotic matrix multiply algorithm at the top. The three-level hybrid algorithm presented here not only has up to 10% better performance than the fastest high-performance matrix multiply, but is also more accurate.

1 Introduction

Matrix multiplication is a fundamental operation. It is used in everything from simulations[15] to implementation of all other level 3 Basic Linear Algebra Subroutines (BLAS) [13]. Accordingly, there has been high demand to for fast matrix multiply.

High performance implementations have reached their limit in terms of efficient use of current processor architecture [4, 8]. This has prompted researchers to introduce *hybrid algorithms* that combine two different approaches. [2]. At the top is a fast asymptotic algorithm, a recursive algorithm matrix for multiplication that reduces the number of recursive calls to multiply submatrices by increasing the number of additions [12, 14]. At the leaves, a highly tuned high-performance matrix multiplication implementation such as [4] or [8] is used. Unfortunately these two-level hybrid approaches have worse accuracy than simply using a high-performance matrix multiply implementation.

There are a number of ways of improving the accuracy of matrix multiply and consequently the hybrid matrix multiply algorithms, but they tend to have poor performance and are difficult to adapt to hybrid matrix multiply algorithms. Pairwise summation [5] is a well-known technique for improving the forward error bound of matrix multiply. It reduces the forward error from $O(n)$ to $O(\log_2 n)$ by distributing the additions over a balanced tree. Adapting pairwise summation to hybrid algorithms has been difficult. Previous attempts have focused on a bottom up approach [1], applied at the kernel level [13]. The bottom up approach has two main drawbacks. First, it requires the modification of a highly tuned matrix multiply implementation [4]. Second, the improvement in accuracy comes at a direct cost in performance. This is because the additional temporary space required for each additional level of pairwise summation and the additional instructions necessary to implement pairwise summation reduce locality and hence increase the number of cache misses.

We introduce a different way of applying pairwise summation to matrix multiply. We apply recursive matrix multiply, at the outer product instead of the inner product. Our method does not require any additional temporary space and does not require the modification of the underlying matrix multiply kernel implementation. We show how our algorithm can be applied to traditional two level hybrid matrix multiply algorithms. This not only improves the overall accuracy of the hybrid algorithm, but also achieves bet-

ter accuracy than high-performance matrix multiply implementations such as Goto[4], while still improving performance by up to 10% over just using the underlying high-performance implementation.

The rest of this paper is organized as follows. Section 2 motivates and introduces our three-level hybrid approach. In Section 3 we discuss various ways in which the inner product of a matrix multiply kernel can be implemented and how it affects accuracy. We show that the bound on the relative error in Goto matrix multiply [4] is effectively $O(\sqrt{n})$. We then present a brief overview of pairwise summation (Section 4). The fact that pairwise summation has a relative error of $O(\log_2 n)$ partially explains why our accuracy is better than that of the Goto implementation. In Section 5, we describe how best to adapt pairwise summation to matrix multiply so that it can be easily adapted to traditional two-level hybrid matrix multiply algorithms, yielding recursive matrix multiply. We then develop the forward error bound for our adaptation, which shows that the placement of recursive matrix multiply on top of the high-performance matrix multiply implementation and rather than inside it is the better approach. Finally, we present experimental results (Section 6). Our results demonstrate that our three-level hybrid approach using Strassen's algorithm [12] at the top offers better accuracy than Strassen's hybrid algorithm alone while still offering improved performance over a typical high-performance implementation. When we use Winograd's variant[14] of Strassen's method at the top, the resulting algorithm is faster than the Goto general matrix multiply implementation, and it is also more accurate when the input is positive.

2 Three level Hybrid

High performance matrix multiply implementations such as Atlas[13], Intel MKL[8] and Goto[4] attempt to maximize the utilization of the processor. These implementations have largely reached their limit. As a result, two-level hybrid matrix have been heavily explored as a mean of further improving the performance of matrix multiply [2, 6, 3]. These algorithms combine a fast asymptotic algorithm for matrix multiplication at the top levels with high performance matrix multiply kernels at the leaves. A major problem with this approach is that fast asymptotic matrix multiply algorithms trade off fewer matrix multiplications against more matrix additions. Since addition is a fundamental source of rounding error, this trade off increases the

error, even if only a few applications of the fast asymptotic matrix multiply algorithm are used[5].

Rounding errors in additions can be reduced using accurate summation algorithms, such as distillation [9] and pairwise summation [5]. Since accurate summation algorithms can be computationally expensive, it is important to decide where in the hybrid algorithm they are best applied. One possibility is to apply an accurate summation algorithm within the fast asymptotic matrix multiply algorithm, replacing all ordinary additions with the accurate summation algorithm. This fails to have any affect on the overall accuracy of the hybrid matrix multiply algorithm because even small errors are amplified by the multiplication inside the fast asymptotic matrix multiply algorithm [5]. A second possibility is to deploy the accurate summation algorithm inside the matrix multiplication kernel itself, improving the accuracy of the leaf computations of the hybrid matrix multiply algorithm. This approach degrades performance due to cache constraints. Since there must be enough cache for both the accurate summation algorithm and the high-performance matrix multiply algorithm, the accurate summation algorithm can only be used in a very limited manner before it impacts performance[1]. This is discussed further in Section 6.

If we want to avoid the above difficulties, then we must deploy the accurate summation algorithm outside of either the matrix multiply kernel or the fast asymptotic matrix multiply algorithm. This will result in a three level hybrid matrix multiply algorithm. There are three possible places where the accurate summation algorithm may be placed: (1) beneath the high-performance kernel, (2) above the fast asymptotic matrix multiply algorithm, or (3) between the two. The first option is not a reasonable choice, because placing the accurate summation algorithm beneath the high performance kernel would completely negate the benefit of using a high performance kernel. The second option has a similar disadvantage: if we deploy the accurate summation algorithm above the fast asymptotic matrix multiply, then we are partitioning the problem at the top and running the fast asymptotic matrix multiplication on smaller problems. This significantly diminishes the performance advantage gained by using fast asymptotic algorithm, defeating the purpose of using it. Thus we are left with the third option, placing the accurate summation algorithm between the fast asymptotic matrix multiply algorithm at the top and the high performance matrix multiply-kernel at the bottom.

We expand upon this notion further in Sections 4 and 6, below. To keep

the paper concise, we focus only on one family of fast asymptotic matrix multiply algorithms, namely Strassen's algorithm [12] and Winograd's variant of Strassen's algorithm[14]. But first we need to discuss the source of rounding errors in matrix multiply implementations and, in particular, high performance matrix multiply implementations. Then we can discuss how to reduce this error in the context of hybrid matrix multiply algorithms.

3 Inner Product Variations

There are many different approaches to computing matrix products and to computing individual inner products. While some of these choices are largely dictated by architecture and performance considerations [13, 4], they impact the overall accuracy of the computation. This becomes important when dealing with high performance matrix multiply kernels. It is possible for the high performance kernel to be more accurate than the canonical matrix multiply algorithm, which improves the accuracy of the overall hybrid algorithm.

Tiling is a common strategy within high performance matrix multiply kernels. The primary goal of tiling is to maximize reuse of the data in the cache and hence improve performance. Subtleties in how the individual tiles are added to the whole greatly impact the forward error bound on the inner product [5]. The difference is sometimes referred to as *preload* vs. *postload* [1]. We adopt this terminology in this paper. The following discussion will demonstrate that in practice these differences allow some high performance matrix multiply kernels to achieve an error bound that is effectively $O(\sqrt{n})$ for common problem sizes [1, 4]. We will argue (in Section 5, below) that not only does our three level hybrid algorithm beat this bound, it continues to do so even when using a fast asymptotic matrix multiply algorithm that is less accurate than the canonical matrix multiply [5].

3.1 Preload and Postload

Preload and postload are best discussed in the context of a specific tiling strategy. Consider the code shown in Figure 1, which multiplies the $M \times K$ matrix A by the $K \times N$ matrix B to produce the $M \times N$ matrix C. The matrices are stored in row-major order. The tiles are square, of size $bSize \times bSize$. For simplicity, we assume that the matrix dimensions are all even multiples of the tile size. The strategy in Figure 1 is to compute C

ALGORITHM: Tiled Matrix Multiply
procedure tiled($A, B, C, M, K, N, bSize$)
for ($i = 0, i < M, i+ = bSize$)
  for ($j = 0, j < N, j+ = bSize$)
    for ($k = 0, k < K, k+ = bSize$)
      $AOffset = i * K + k$
      $BOffset = k * N + j$
      $COffset = i * N + j$
      multiply(A+$AOffset, K, B$+$BOffset, N,$
          C+$COffset, N, bSize, bSize, bSize$)
    end for
  end for
end for
end procedure

Figure 1: Tiled Matrix Multiply

ALGORITHM: Direct Matrix Multiply (Preload)
procedure multiply($A, ldA, B, ldB, C, ldC, M, K, N$)
begin
for ($i = 0, i < M, i + +$)
  for ($j = 0, < N, j + +$)
    for ($k = 0, k < K, k + +$)
      $C[i * ldC + j]$ += $A[i * ldA + k] * B[k * ldB + j]$
    end for
  end for
end for
end procedure

Figure 2: Preload Matrix Multiply

```
ALGORITHM: Direct Matrix Multiply (Postload)
procedure multiply(A, ldA, B, ldB, C, ldC, M, K, N)
begin
for (i = 0, i < M, i++)
    for (j = 0, < N, j++)
      sum = 0
      for (k = 0, k < K, k++)
        sum += A[i * ldA + k] * B[k * ldB + j]
      end for
      C[i * ldC + j] += sum
    end for
end for
end procedure
```

Figure 3: Postload Matrix Multiply

one column of tiles at a time, always keeping in memory a $bSize \times k$ strip of A along with the tile of C currently being computed. The code repeatedly multiplies a tile of A by a tile of B and adds the result to a tile of C. Each partial tile computation is performed by calling the procedure

$$\texttt{multiply}(A, ldA, B, ldB, C, ldC, M, K, N)$$

where A,B, and C are the addresses of the start of the tiles; the tiles are of sizes $M \times K$, $K \times N$, and $M \times N$ respectively; and ldA, ldB, and ldC are the *strides* (i.e., the row lengths) of the three matrices.

The error of the algorithm shown in Figure 1 depends on how the inner kernel (i.e., the `multiply()` procedure) is implemented. If the implementation is the preload code shown in Figure 2, the summation of the inner product is recursive (standard) summation of K terms. In contrast, the postload code shown in Figure 3 computes the contribution of the tile to the inner product in a temporary variable and adds this sum to the inner product. When the code of Figures 1 and 3 is combined, the net effect is to compute the dot product as the sum of $K/bSize$ partial sums, each of which is the sum of $bSize$ terms. This has a significant effect on the forward error bound, as will be discussed in the next subsection.

3.2 Forward Error Bound

In order to do error analysis of the tiling strategies presented in the previous section, a model must be used to account for the rounding errors, particularly the rounding errors in addition. We use the standard floating point model for estimating error bounds presented in [5], as it widely used and holds for IEEE standard arithmetic. In this model, the result of an individual floating point operations is $fl(x+y) = (x+y)(1+\delta)$, where $|\delta| \leq u$ and δ is the relative error of an individual operation, u is the "unit in last place" (commonly known as ulp), and $fl(\cdot)$ is the value contained inside the parenthesis rounded to machine precision. This means the previous statement says that the real answer, rounded to machine precision, is equal to the real answer plus some small relative error, less than or equal to one ulp. Using this model, the following forward error bound for the inner product of matrix multiply is derived in [5]:

$$|x^T y - fl(x^T y)| \leq \gamma_n \sum_{i=1}^{n} |x_i y_i| = \gamma_n |x|^T |y|, \tag{3.1}$$

where $\gamma_n = nu/(1-nu)$. If the inner product is broken up into n/k strips, each containing k summands, the forward error bound becomes:

$$|s_n - \widehat{s_n}| \leq \gamma_{\frac{n}{k}+k-1} |x|^T |y|. \tag{3.2}$$

Here, s_n is the true value of the inner product, $\widehat{s_n}$ is the computed value, and k is equal to $bSize$. This means that $bSize$, which is usually chosen based upon the architecture of the machine, has a dramatic effect on the forward error bound. Obviously, by allowing $k = \sqrt{n}$ the forward error bound can be further improved, however, this is not practical for most problem sizes as k is usually quite small as it is limited by the size of the cache. An interesting side effect of (3.2) is when $k = c\sqrt{n}$, as proven in [1]:

$$\gamma_{(c\sqrt{n}+\frac{n}{c\sqrt{n}}-1)} = \gamma_{(c\sqrt{n}+\frac{1}{c}\sqrt{n}-1)} = \gamma_{((c+\frac{1}{c})\sqrt{n}-1)}. \tag{3.3}$$

As illustrated by Castaldo, Whaley and Chronopoulos[1], when $k = 60$, you achieve roughly the same bound for $n \in [900, 14400]$ as $60 = 2\sqrt{900}$ and $60 = \frac{1}{2}\sqrt{14400}$. This means for most practical problem sizes, you end up with a forward error bound that is effectively $O(\sqrt{n})$ in practice. This effect is illustrated by comparing Goto BLAS [4] to the canonical $O(n^3)$ matrix

multiply algorithm in figure 5. In the next section we will describe how it is possible to achieve a lower bound by further changing the order in which addition is carried out. This will give us a mechanism, that when combined with a traditionally less accurate two level hybrid matrix multiply algorithm, will allow us to produce an algorithm more accurate and faster than just the high performance matrix multiply algorithm alone, which we demonstrate in section 5.

4 Pairwise Summation

As discussed in the previous section, if the addition is broken up, the longest path from the summand to the total is shortened, it is possible to reduce the error. Where standard summation is simply linear, pairwise summation breaks apart the addition into a balanced tree, where the leaves are the summands and the root is the total. Pairwise summation is usually defined as a special case of recursive (standard) summation and this time will be no different. Recursive summation is traditionally defined in terms of a set of summands, by removing two elements from the set, adding them together, and placing the new sum back into the set. This recursion repeats, hence the name, until only one element remains, the final sum. In contrast, pairwise summation adds adjacent elements together creating new summands, then again adds adjacent summands together. The process of adding adjacent summands repeats until a final sum is reached. A recursive algorithm to illustrate pairwise summation can be found in Figure 4, where n is the number of elements and the elements are numbered x_1 to x_n. The following forward error bound for pairwise summation is derived in [5]:

$$|s_n - \widehat{s_n}| \leq \gamma_{\log_2 n} \sum_{i=1}^{n} |x_i| \tag{4.1}$$

In the next section we will describe some of the pitfalls of applying pairwise summation directly to the inner product along with describing a better method of adapting pairwise summation to matrix multiplication, at the outer product level.

ALGORITHM: Pairwise Summation
procedure pwsum($n, x_1, ..., x_n$)
begin
if($n == 1$)
  return x_1
else if($n == 2$)
  return $x_1 + x_2$
else
  $k = \lfloor n \div 2 \rfloor$
  return pwsum($k, x_1, ..., x_k$) + pwsum($n - k, x_{k+1}, ..., x_n$)
end if
end procedure

Figure 4: Pairwise Summation

5 Direct Recursive Matrix Multiply

As suggested by Higham, the forward error bound can be greatly improved by applying pairwise summation to the entire inner product, producing a forward relative error bound of $\gamma_{\lceil \log_2 n \rceil + 1} |x|^T |y|$ [5]. However, the full implementation of this matrix multiplication variation is not practical as the performance tends to be abysmal. Previous attempts to apply pairwise summation to dense matrix multiply kernels have therefore focused on a bottom up approach, attempting to increase the accuracy of the individual inner product (or products as tiling is usually used) while using as little additional temporary space as possible [1]. This is done by applying pairwise summation at the tile level and only applying a few levels, usually only three. The amount of temporary space required for this approach is a multiple of the tile size and grows linearly with the number of levels of pairwise summation that are applied.

This approach has two main drawbacks. The first is that performance suffers as the levels of pairwise summation increase, because as more temporary space is required to store the temporary sums. This temporary space reduces the amount of space available for the actual matrix multiply kernel, reducing matrix reuse and harming performance. This effect can be mitigated by only applying a few levels of pairwise summation, usually only three [1]. But this reduces the accuracy, partially defeating the purpose of applying pairwise

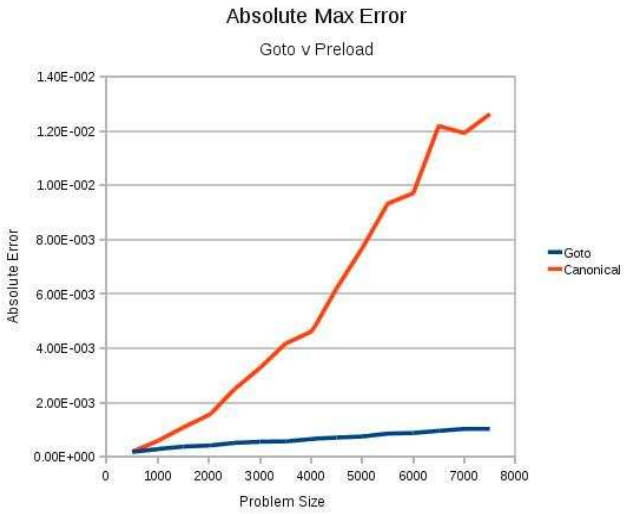

Figure 5: Goto Absolute Error

summation. The second drawback is that this implementation must be done at the kernel level of the high performance matrix multiply implementation, increasing the complexity of the implementation. This complexity issue is compounded in implementations such as Goto which have a different kernel for each specific architecture [4].

There is a much simpler way of adapting pairwise summation to matrix multiply via a top down approach, directly using the definition of matrix multiply. Given matrices

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, \text{ and}$$

$$C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

the entries of C are given by

$$\begin{aligned} C_{1,1} &= A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1}, \\ C_{1,2} &= A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2}, \end{aligned}$$

$$C_{2,1} = A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1},$$
$$C_{2,2} = A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2}.$$

Each submatrix of C can be computed using the inner products shown above. If we further decompose the submatrices of A, B, and C into submatrices, this process can be repeated, yielding a balanced tree computation. This algorithm is called recursive matrix multiply.

In the next subsection we will discuss the effect of computing a matrix product by running direct recursion until the submatrices are down to a certain size, and then passing these leaves to a high performance matrix multiply implementation. We compare this top-down approach to pairwise summation, with a bottom-up approach implemented within the high performance matrix multiply implementation.

5.1 Forward Error Bound

Using the same standard model as described in Section 5.1, let k be the leaf size at which direct recursion is terminated. From (3.1), we know that the forward error in the computation of each leaf is bounded above by

$$\gamma_k \sum_{i=1}^{k} |x_i y_i| = \gamma_k |x|^T |y|.$$

Since there are n/k leaves, it follows from (4.1) that the forward error of adding up the leaves after they are computed satisfies

$$|s_n - \widehat{s_n}| \leq \gamma_{\lceil \log_2(\frac{n}{k}) \rceil + 1} |x|^T |y|.$$

Combining the two bounds yields

$$|s_n - \widehat{s_n}| \leq |x|^T |y| (1+\delta)^k (1+\delta)^{\log_2(\frac{n}{k})}, \tag{5.1}$$

which when simplified becomes

$$|s_n - \widehat{s_n}| \leq \gamma_{(\lceil \log_2(\frac{n}{k}) \rceil + k)} |x|^T |y|. \tag{5.2}$$

In contrast, it is shown in [1] that the forward error in a bottom-up approach with three levels of pairwise summation and a leaf size of k satisfies:

$$|s_n - \widehat{s_n}| \leq \gamma_{k+2(\sqrt{\frac{n}{k}}-1)} |x|^T |y| \tag{5.3}$$

Increasing the number of levels degrades performance significantly and marginally improves the forward error bound, but for any fixed number of levels the error bound is still asymptotically worse than $\log(n/k)$. Hence our approach using direct recursion yields a smaller forward error bound than a bottom up approach applying pairwise summation.

When the leaf size k is fixed, direct recursion is more accurate for large problems than Goto BLAS [4], since the bound on the relative error in the former grows in proportion to $\log_2 n$, while the bound on the relative error in the latter grows in proportion to $\sqrt{n}$.

It can be seen from (5.2) that the leaf size dominates the relative error when recursive matrix multiply is placed in the middle. This means that if instead of fixing the leaf size we fix the number of levels of recursion, accuracy will be degraded. We will see in the results section that for the largest problem sizes, if we fix the number of levels of recursion and let the leaf size vary, the error approaches the error of not inserting the intermediate incursion at all. Empirically, the crossover point appears to be about 8k, as seen in Figure 7 and Table 1. This is discussed in more detail in Section 6.

Our approach allows a large problem size to have an error very close to that of a much smaller problem size, when k is fixed. By placing direct recursive matrix multiply in the middle of the traditional hybrid matrix multiply implementation, we offer a path between the fast asymptotic matrix multiply algorithm at the top and the high performance matrix multiply implementation at the bottom without greatly increasing the forward error in between.

6 Results

The high performance matrix multiply implementation chosen to illustrate the effects of postload was Goto [4]. Atlas [13] was also considered, as was Intel MKL [8], however, Atlas was found to have worse performance and Intel MKL had nearly identical performance to that of Goto.

6.1 Testing Methodology

Unless otherwise stated, the matrices tested were square matrices. The reference was generated with Goto BLAS [4], using double precision general matrix multiply (dgemm). Originally, the tests were checked against double

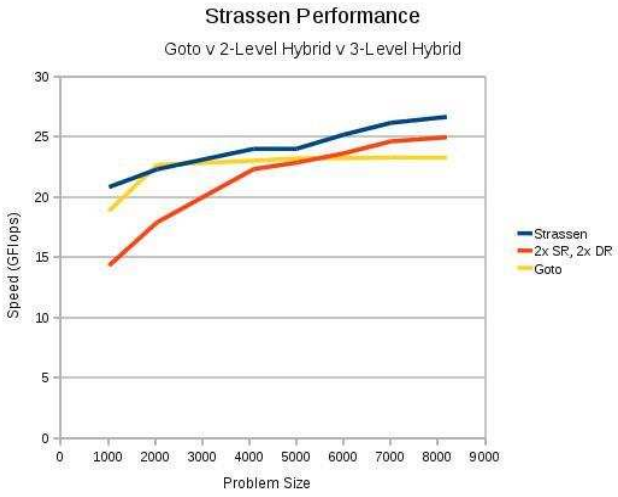

Figure 6: Strassen Three Level Hybrid Performance

precision where the summation of the individual dot products was done using doubly compensated summation [11], however, double precision appears to be sufficient when the test are conducted in single precision and has the added benefit of reducing the time to generate the reference for a large number of tests. The input for all tests was a uniform distribution between [0,1], generated using GNU R, the seeds provided by random.org. The machine used was a Q9450 Penryn Intel Quad processor running at 3GHz on a 64-bit Kubuntu 10.04 desktop installation. The metrics used to compare various implementations, in terms of accuracy, are those of relative error and absolute error [5]. Though several other metrics exist [16, 10, 1], they do not appear to be widely used. For performance, we compare implementations by how quickly each algorithm can perform the essential $O(n^3)$ operations, whether or not the implementation actually does them, measured in flops (floating point operations per second). The high performance matrix multiply implementation used was Goto [4] where not only the implementation, but also the entire hybrid matrix multiply algorithm, are single threaded. Implementation details and the impact of multi-threading on accuracy and performance will be explored in future works. The implementation used as the performance reference for Winograd's variant in Table 3 is described in [2].

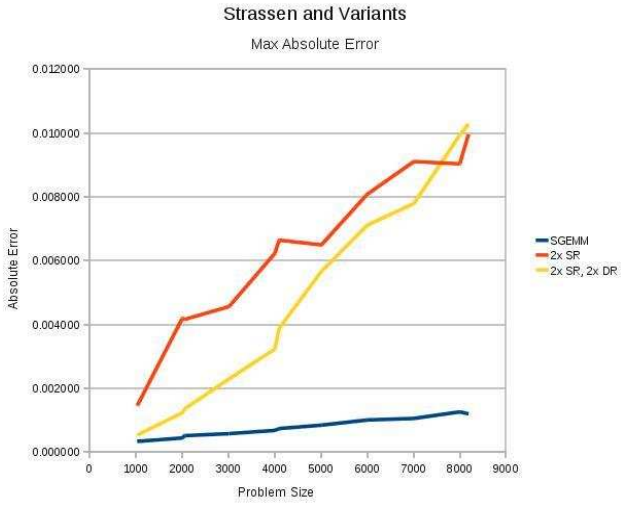

Figure 7: Strassen Three Level Hybrid Accuracy

6.2 Measuring Error

High performance implementations such as Goto have traditionally enjoyed better accuracy than the canonical matrix multiply, as illustrated in Figure 5, which compares the absolute max error of the standard matrix multiply as compared to Goto's single precision general matrix multiply (sgemm), both computed in single precision. Without this attribute of additional accuracy, hybrid matrix multiply algorithms would not be useful as the error of the asymptotically fast matrix multiply algorithm would be too large. In practice though, the additional accuracy offered by high performance implementations such as Goto makes hybrid algorithms possible[2]. The additional accuracy provided by Goto however has been traditionally attributed to a feature of the architecture, namely, 80 bit floating point registers that are available on Intel x86 processors. The idea is fairly simple, if the high performance matrix multiply kernel is effectively maximizing reuse of matrix A, B or C, or any combination of the three, in doing so, it is preventing context switching. Traditionally this is done through maximizing the reuse of whatever is stored in the L1 cache and by breaking the data into tiles that can be efficiently moved and reused in the L1 cache [13]. In addition to the previous idea, Goto's data strategy attempts to further reduce context switches explicitly by reducing translation look-aside buffer (TLB) misses by changing how the

data is accessed [4]. Both of these strategies were developed for performance reasons, as context switches are generally expensive. However, by reducing the context switches, if 80 bit extended precision floating point registers are used then additional accuracy is achieved as the summation of individual dot products of the matrix multiply have additional bits to remember what would have been lost to rounding error if only 64 bits or 32 bits were used, for double and single precision respectively. This 80 bit register is not rounded to 64 or 32 bits until it is written back to memory, such is the case when a context switch occurs. Therefore, by improving performance, by reducing context switches, accuracy is also improved.

Unfortunately the effect of extended precision registers has not been true for several years, at least in modern implementations, and most importantly, it is not true in the current version of Goto BLAS [4]. The problem dates back to the introduction of the SSE instruction set, which mapped MMX registers to FP registers, which are 80 bit registers on Intel processors[7]. When the SSE2 SIMD instruction set was introduced (single instruction, multiple data), the MMX registers were replaced with XMM registers, which are 128 bit registers [7]. These register operate on packed double precision or packed single precision data, two double precision values or four single precision values, respectively. These registers no longer offer additional accuracy. This was tested empirically by comparing the results of vector dot products produced by the $O(n^3)$ matrix multiply algorithm in single precision against that produced using inline assembly using the SSE2 instruction set (four vectors each), the results were identical. By inspecting Goto's implementation for the Penryn processor, as the inner matrix multiply kernel in Goto is tailored to specific processors, it is easy to see that XMM registers are used and not MMX. This means that the error observed in Figure 5 is entirely because of the preload affect described in Section 3 and not something specific to the architecture. It may appear in Figure 5 that Goto's implementation is logarithmic, however as discussed in Section 3, Goto's implementation is effectively square root[1]. As can be clearly seen from the graph, a simple tiling strategy combined with postload matrix multiply offers a very low error for practical problem sizes.

6.3 Strassen's Algorithm

Before discussing the data further, it is important to emphasize a key purpose of hybrid algorithms and ultimately the limitations of matrix multiply

and the machines the algorithms run on, namely, the reason for the existence of hybrid matrix multiply algorithms. Goto's implementation of matrix multiply and his resulting BLAS implementation is widely considered to be the fastest implementation of the canonical matrix multiply [4]. This is achieved through a key insight into current x86 processor architecture, resulting in a highly efficient implementation. This means, that in order to improve performance any further, a asymptotically fast matrix multiply implementation must be used, something that reduces the total number of multiplications. Though this detail may seem minor, it can be quickly illustrated in Figure 6 and 7. Figure 6 is the overall speed of the implementations, as described in Subsection 1 of Section 6, comparing Goto's matrix multiply, a standard hybrid implementation of Strassen's algorithm on top of Goto's matrix multiply which applies Strassen's algorithm until a leaf size of 1k or less before switching to Goto, and that of our adaptation by placing direct recursive matrix multiply in between. To be explicit, our algorithm runs Strassen's algorithm twice, unlike the standard hybrid implementation which is variable with a fixed leaf of 1k or less, we then run direct recursive matrix multiply twice before finally switching to a high performance matrix multiply implementation. This means the leaf size passed to the high performance matrix multiply implementation is not fixed, that the levels of recursion are, the consequences of which are discussed in Section 5, Subsection 1. By comparing the implementations it is evident that our algorithm is faster than the high performance matrix multiply implementation, but not as fast as a standard Strassen hybrid. By also looking at Figure 7, which compares the algorithms in terms of max absolute error, it is clear that our algorithm is more accurate than the standard Strassen hybrid matrix multiply and that neither are more accurate than just using Goto's matrix multiply. The importance of the discussion at the beginning of the subsection now becomes apparent, namely, though both implementations are not as accurate as the high performance matrix multiply implementation by itself, a hybrid matrix multiply algorithm is the only way to improve performance past the high performance canonical matrix multiply implementations. This means that if one wants an algorithm that is faster than the fastest implementation of the canonical matrix multiply, they must use a hybrid matrix multiply algorithm, however, if they want more accuracy than what is offered by a standard hybrid matrix multiply algorithm and still better performance than the canonical high performance matrix multiply implementation, our algorithm offers a solution. Fortunately this can be improved further by taking advantage of a peculiar-

ity of certain asymptotically fast matrix multiply algorithms, in particular, that Winograd's variant is more accurate in practice than Strassen's when the input matrices are positive, as we will see in the next subsection.

6.4 Winograd's Algorithm

Winograd luckily has an interesting property in that it does not have "true" subtraction, which means that when the input is positive, it is actually more accurate than Strassn's algorithm [2]. This is easily demonstrated in Table 1 where the first column represents the problem size and the rows are the accuracy of individual algorithms for that problem size. When Winograd is combined with a fixed two levels of direct recursive matrix multiply, the error grows linearly with the size of the leaf, as described in Section 5.1, which can be clearly seen in Table 1. This means two applications of direct recursive matrix multiply offers better accuracy up until the size of the leaf overwhelms the benefit of two levels of direct recursive matrix multiply, resulting in no benefit in terms of accuracy at around 8k. Fortunately we already know this can be further improved by using a fixed leaf size and variable levels of direct recursive matrix multiply. By drawing attention to Table 1 we can clearly see the benefit of a fixed leaf of 256. Instead of the error growing linearly in terms of the leaf size, the leaf error becomes a constant and instead the error grows linearly in terms of the level of direct recursive matrix multiply, which grows slower, as discussed in Section 5. Table 2 contains the absolute error using a slightly larger fixed leaf size of 320. Though not as accurate as a fixed leaf size of 256, it has the added benefit of being faster with this particular high performance matrix multiply kernel implementation, as seen in Table 3. By comparing both Tables 2 and 3, a fixed leaf size of 320 appears to be a good choice in terms of accuracy and speed and that a larger leaf size would not be desirable. The reason for this is though performance may vary by architecture and implementation, accuracy largely does not. The only benefit of a different leaf size would be better accuracy if the leaf was smaller and if the high performance implementation offered it, better performance for a smaller leaf size. Unfortunately Goto BLAS [4] appears to be implemented with larger problem sizes in mind. This is revealed in Table 3 where Goto's implementation appears to plateau starting at a problem size of 2k, far too large to be used as a leaf computation when accuracy is important.

Winograd Fixed Leaf Variants				
Size	SGEMM	2xWino	2xWino2xDR	WinoDR256
1024	0.000331	0.000225	0.000108	0.000136
2048	0.000509	0.000602	0.000228	0.000231
4096	0.000733	0.000848	0.000562	0.000450
8192	0.001194	0.001413	0.001415	0.000834

Table 1: Winograd Hybrids, Absolute Error

Winograd Fixed Leaf Variants				
Size	SGEMM	2xWino	2xWino2xDR	WinoDR320
1024	0.000331	0.000225	0.000108	0.000216
2048	0.000509	0.000602	0.000228	0.000334
4096	0.000733	0.000848	0.000562	0.000579
8192	0.001194	0.001413	0.001415	0.001001

Table 2: Winograd Hybrids, Absolute Error

Winograd Fixed Leaf Variants				
Algorithm	1024	2048	4096	9024
SGEMM	18.83	22.64	22.99	23.30
Wino	21.83	23.66	26.15	29.49
2xWino 2xDR	14.31	18.47	22.71	25.71
2xWino 256DR	17.89	18.57	19.80	20.42
2xWino 320DR	20.45	21.07	22.77	24.07

Table 3: Winograd Hybrids, Speed in GFlops

7 Conclusion

In this paper we have presented what we believe to be the first three level hybrid matrix multiply algorithm. We have demonstrated not only the usefulness of a three level hybrid matrix multiply algorithm, but identified a clear need for one, a need for something that improves both performance and accuracy over a traditional high performance matrix multiply implementation. We have proven that our ordering of the individual levels is the correct or-

dering for the hybrid matrix multiply algorithm. We have also built a model that can be used for determining the proper number of applications of direct recursive matrix multiply based upon the problem size, which will allow any application programmer to tailor the three level hybrid algorithm to their needs. We have also demonstrated empirically that this model is correct and that it is possible to out perform a traditional high performance matrix multiply implementation by up to 10% while still offering superior accuracy. Furthermore, we have demonstrated that the performance and accuracy of our three level hybrid matrix multiply algorithm will only improve in the future with high performance matrix multiply implementations specifically tailored for hybrid algorithms.

References

[1] A. M. Castaldo, R. C. Whaley, and A. T. Chronopoulos. Reducing floating point error in dot product using the superblock family of algorithms. *SIAM J. Sci. Comput.*, 31(2):1156–1174, 2008.

[2] P. D'Alberto and A. Nicolau. Adaptive Winograd's matrix multiplications. *ACM Trans. Math. Softw.*, 36:3:1–3:23, March 2009.

[3] C. C. Douglas, M. Heroux, G. Slishman, R. M. Smith, and R. M. Gemmw: A portable level 3 Blas Winograd variant of Strassen's matrix-matrix multiply algorithm, 1994.

[4] K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication. Working Note 9, The University of Texas at Austin, November 2002.

[5] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002.

[6] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of Supercomputing '96*, pages 9–6, 1996.

[7] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*. Intel Corporation, June 2010.

[8] Intel. Intel math kernel library (intel mkl) 10.2, 2010.

[9] Y. Nievergelt. Analysis and applications of Priest's distillation. *ACM Trans. Math. Softw.*, 30(4):402–433, 2004.

[10] F. W. J. Olver. A new approach to error arithmetic. *SIAM Journal on Numerical Analysis*, 15(2):368–393, April 1978.

[11] D. M. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California Berkeley, 1992.

[12] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[13] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *PPSC*, 1999.

[14] S. Winograd. A new algorithm for inner product. *IEEE Trans. Comput.*, 17:693–694, July 1968.

[15] M. Yokokawa, K. Itakura, A. Uno, T. Ishihara, and Y. Kaneda. 16.4 Tflops direct numerical simulation of turbulence by a Fourier spectral method on the Earth Simulator. In *Proceedings of The 2002 ACM/IEEE Conference on Supercomputing*, 2002.

[16] A. Ziv. Relative distance - an error measure in round-off error analysis. *Mathematics of Computation*, (160):563–569, October 1982.