Improving Numerical Accuracy for Non-Negative Matrix Multiplication on GPUs using Recursive Algorithms

Matthew Badin
University of California Irvine
Irvine, CA 92697
mbadin@uci.edu

Paolo D'Alberto
FastMMW
paolo@FastMMW.com

Lubomir Bic
University of California Irvine
Irvine, CA 92697
bic@ics.uci.edu

Michael Dillencourt
University of California Irvine
Irvine, CA 92697
dillenco@ics.uci.edu

Alexandru Nicolau
University of California Irvine
Irvine, CA 92697
nicolau@ics.uci.edu

ABSTRACT

Scientific computing is only bound by the limits of Moore's Law and the scalability of high performance mathematical library implementations. Most mathematical libraries however tend to focus only on general inputs, limiting their potential performance and scalability by not tailoring their implementation to specific inputs, such as non-negative inputs. By removing this limitation it is possible to improve the performance and accuracy of a range of problems. In this paper we explore the limitations of hardware to improve accuracy of non-negative matrix multiply by specifically comparing implementations on the GPU and CPU and propose algorithmic solutions to improve accuracy. Next, we demonstrate a matrix multiply implementation that takes advantage of asymptotically fast matrix multiply algorithms, which have been shown to scale better than $O(N^3)$ matrix multiply implementations, and improve accuracy by up to a whole digit while increasing performance by up to 27% for matrices where the input is positive. Finally, we propose to extend the BLAS level 3 specification to non-negative matrices to allow easy integration of our solution and allow other library authors to implement their own solutions as part of an existing standard.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Algorithm Design and Analysis

General Terms

Algorithms, Performance

Keywords

GPGPU, Hybrid Matrix Multiply, BLAS, Accuracy

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.

1. INTRODUCTION

The adoption of parallel processors, and in particular graphics cards (GPUs), are becoming increasingly popular in scientific computing because of their low price and high performance [17]. This has lead researchers to offer high performance matrix multiply implementations for the GPU [22] along with BLAS 3 (Basic Linear Algebra Subroutines) for use in general scientific computing [1]. GPUs and GPU like devices though, with idiosyncratic architecture and a flat memory hierarchy, will yield drastically different matrix multiply implementations [22] than their CPU counterparts [23]. Architectural limitations such as the size or even availability of a cache not only change how matrix multiply is implemented [12, 1], but also affects the overall accuracy of the computation as it dictates how the computation is decomposed [5, 13], resulting in poorer accuracy on the GPU. This deviation in accuracy due to algorithmic differences is true even though the GPU uses an FMA (Fused-Multiply-Add) [20] which produces fewer rounding errors than the CPU implementation [13, 12]. This is analyzed and discussed in Section 3 and demonstrated empirically in Section 5. To solve this accuracy problem, we approach solutions by breaking apart the problem into two subclasses, matrices with positive inputs and those with both positive and negative inputs. Specifically, we will use uniform random between $[0, 1]$ for the input matrices. We place no restriction on the result matrix. This is natural as uniform random matrices with inputs between $[0, 1]$ and $[-1, 1]$ are considered representative of real world inputs and are widely used [11, 13, 3]. In particular, problems that are represented by the $[0, 1]$ range include Markov chains [9], computational biology [2] and pattern recognition [8]. Knowing this and knowing the input allows us to specifically tailor algorithmic solutions, combined with our understanding in how subtle differences in the computation can be broken up to transcend the architectural limitations of the GPU, we are able to not only improve performance by up to 27%, we can also improve accuracy when the matrix input is positive. In addition, because of the quantity of problems represented by the $[0, 1]$ input range and the ability and availability of algorithmic solutions that cater to problems of this type, we propose to extend the BLAS 3 specification to allow other library authors to likewise provide solutions for these types of problems instead of always implementing solutions that cater to the worst case, general inputs. We go into detail about the

BLAS 3 extension proposal in Section 6. This will all be presented through the lens of a new scalable GEMM implementation for the GPU. The rest of the paper is organized as follows: We will start by briefly covering previous work in the area of high performance matrix multiply and BLAS 3 implementations along with hybrid matrix multiply implementations. We will then discuss how the decomposition of the problem affects the overall numerical accuracy of the result, demonstrating this difference empirically by comparing different matrix multiply implementations on different architectures, namely the CPU and the GPU. Next we will demonstrate how this information can be used to build a hybrid matrix multiply implementation that improves performance on the GPU by up to 27% over the native implementation and also improves accuracy by exploiting asymptotically fast matrix multiply algorithms, such as Winograd's variant of Strassen's algorithm [24], that behaves like the canonical $O(N^3)$ matrix multiply algorithm in terms of accuracy when the input is non-negative. Finally, capitalizing on this insight, we propose an extension to the BLAS 3 specification to allow other library authors to take advantage of other asymptotically fast matrix multiply algorithms with similar properties.

2. PREVIOUS WORK

Matrix multiply is ubiquitous in scientific computing. Consequently, considerable effort has been spent on improving its performance. One of the first BLAS 3 implementations to be widely available was ATLAS [23]. ATLAS was unique in that instead of writing architectural specific implementations it would attempt to automatically find the best implementation for a given architecture through an exhaustive search of standard optimization techniques such as blocking and loop unrolling. Various combinations of these techniques would be compiled and tested on a given architecture, the fastest implementation being saved and then used as the basis for all other BLAS 3 functions. Further performance could be found though by ignoring the L1 cache and instead targeting the L2 cache. This insight became the basis of Goto's BLAS implementation [12]. Goto's implementation is widely considered the most efficient on x86 based machines and is even used in official vendor provided libraries such as Intel's Math Kernel Library [16]. With Goto's BLAS implementation, high performance implementations on x86 based machines reached their limit in terms of efficient use of the processor. In order to further increase performance hybrid matrix multiply implementations had to be considered. Hybrid matrix multiply implementations work by first decomposing a problem using matrix multiply algorithms that are asymptotically faster than the canonical $O(N^3)$ algorithm (An example of the canonical matrix multiply algorithm can be found in Section 3.4 Figure 4). The hybrid matrix multiply algorithm then solves the decomposed parts using a high performance implementation like the previously mentioned ATLAS [23] or Goto BLAS [12]. The current state of the art for hybrid matrix multiply algorithms on x86 based machines is the implementation by D'Alberto and Nicolau [10].

Unlike x86 based machines, GPU implementations vary widely due to constantly changing architecture. Within the Nvidia family of GPUs, the best currently available implementation for the Tesla chipset is that offered by Volkov and Demmel [22]. The Fermi chipset, a more modern chipset offered by Nvidia, the best implementation is that offered through MAGMA BLAS [1]. Both implementations though, like all high performance matrix multiply implementations, are limited in terms of their efficient use of the architecture. The only hybrid implementation on the GPU is that by Li, Ranka and Sahni [18]. This implementation demonstrated the performance hybrid matrix multiply algorithms on the GPU. The accuracy analysis of their hybrid matrix multiply algorithm however was fundamentally limited by hardware implementation choices made by Nvidia for that chipset. On the Tesla chipset, the FMADD (floating point multiply and add) did not conform to the IEEE 754 standard, the intermediate result was truncated instead of rounded [20]. The effect of which was exhaustively measured in our previous work [4]. High performance CPU implementations were also not compared, which are well known to be more accurate than the canonical $O(N^3)$ matrix multiply implementation [5]. Additionally, with the Fermi chipset Nvidia replaced the FMADD operation with FMA (fused-multiply-and-add) which halves the number of rounding errors committed [13]. In the next section we will explore the consequences of FMADD versus FMA in terms of theoretical bounds. We will then show empirically how algorithmic decisions play a larger overall role in the accuracy of high performance matrix multiply implementations than whether an FMA or an FMADD is used to compute individual partial products. We will then show in Section 4 how high performance matrix multiply implementations on the GPU can be improved using asymptotically fast implementations, both in terms of performance and accuracy.

3. NUMERICAL ACCURACY

As discussed in Section 2, there are a number of high performance matrix multiply implementations that exist for both the CPU and the GPU. In order to compare the different matrix multiply implementations in terms of numerical stability it becomes useful to have a way of modeling individual rounding errors created by individual operations. The operations of primary concern are those associated with the innermost dot product of each matrix multiply implementation, namely multiplication and addition. Though the model also captures subtraction and division. This model is discussed in Section 3.1.

Empirical results are also useful as they help to not only validate a model and bring to light hidden constants in any error analysis but they can also sometimes illustrate just how pessimistic a model can be. This is the case when comparing asymptotically fast matrix multiply implementations as their accuracy in terms of their error analysis is usually much worse than they are in practice [5, 11]. The empirical results are measured in terms of relative error and absolute error. Relative error is useful as it attempts to capture the error independently of the magnitude of the input, thus allowing for direct comparison of different implementations and architectures. Absolute error is useful if the input matrices are between $[0, 1]$ and $[-1, 1]$ as one can simply scale this difference relative to the magnitude of the input matrices they commonly use. Though other metrics for measuring error exist [25, 21], they do not appear to be commonly used.

For the rest of the paper we also define blocking to mean breaking apart the computation on a single dimension. For instance, by using the canonical three nested loop version of matrix multiply from Section 3.4 Figure 4 as an exam-

ple, there are three different loops, or rather, three different dimensions that can be blocked on. Blocking involves computing a certain number of elements of a given dimension together instead of computing all elements available in that dimension. For example, given three matrices A, B and C with dimensions $M \times K$, $K \times N$ and $M \times N$, respectively, and a matrix product $A \times B = C$. If we were to write that we are blocking on M and N, we would be blocking on the result matrix C. This means that all of K would be computed for a given block of M and N. Blocking on the M and N dimensions, also known as blocking on the result matrix C, is a common strategy for parallelization as each block of C can be computed independently. For instance, if each block size was $M/2$ by $N/2$, four independent submatrices of C could be computed in parallel. It is important to note that blocking on the K dimension is the only dimension that will affect accuracy. A more detailed blocking example can be found in Section 3.4 as well as an analysis of how blocking on K affects accuracy in Section 3.3.

3.1 Measuring Rounding Error

Before discussing the standard model, some concepts and notation must first be introduced, namely the concept of "unit roundoff". Unit roundoff, also known as *machine epsilon*, is defined as β^{1-p} where β is the base, in this case 2 as we are dealing with binary arithmetic and p is the precision being used. This means that unit roundoff will vary with the precision used in the computation. For the rest of the paper we will represent unit roundoff with the letter u. We also assume round to nearest in all our analysis and testing as it is the IEEE default rounding mode. In the next subsection we will cover the notation we will be using to analyze the various matrix multiply implementations.

3.2 Notation

In this paper we will use identical notation as that used in "Accuracy and Stability of Numerical Algorithms" by Higham [13]. To briefly cover the important ideas from the book that are relevant to this paper: If x and y are values, and their product is expressed as xy, then $fl(xy)$ would be the product rounded to the precision of the machine. Likewise, if we assign $z = xy$ then we can represent the machine computed result using $\widehat{z}$ so that $\widehat{z} = fl(xy)$. The difference between the compute result and the actual result can then be expressed simply as $|z - \widehat{z}|$. To quantify exactly how much that difference is, we turn to the standard model as presented in Higham's book [13] as the model is widely used and holds for IEEE standard arithmetic. In this model, the result of an individual floating point operations is:

$$fl(x \ op \ y) = (x \ op \ y)(1 + \delta) \tag{3.1}$$

Here δ represents the relative error of an individual operation and is assumed to be $|\delta| \leq u$. This means that equation 3.1 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 machine epsilon.

3.3 Analysis

Using the standard model represented by equation 3.1, the following forward error bound for the inner product of matrix multiply is derived in [13]:

$$|x^T y - fl(x^T y)| \leq \gamma_K \sum_{i=1}^{K} |x_i y_i| = \gamma_K |x|^T |y|, \tag{3.2}$$

where $\gamma_K = Ku/(1 - Ku)$. Simplified, this becomes:

$$|s_K - \widehat{s_K}| \leq \gamma_K |x|^T |y|, \tag{3.3}$$

where s_K is the correct result and $\widehat{s_K}$ is the computed result. This model holds for the canonical $O(N^3)$ matrix multiply (or specifically $O(MNK)$), however it no longer holds when the computation is broken up, the normal result of implementing a blocking strategy. If the inner product is broken up into K/b strips, where K is the problem size and each strip contains b summands, the forward error bound becomes:

$$|s_K - \widehat{s_K}| \leq \gamma_{\frac{K}{b}+b-1} |x|^T |y| \tag{3.4}$$

The benefit of blocking on K is empirically demonstrated in Figures 1 and 2 by comparing SGEMM (single precision general matrix multiply) implementations from Nvidia's CUBLAS on a Nvidia Tesla C2070 as compared to GotoBLAS on a Q9450 Penryn Intel Quad Core processor. For both Figures 1 and 2 the input is uniform random between $[0, 1]$, the result is computed using single precision (SGEMM) and the standard the result is compared against is computed using double precision on the CPU. There are no restrictions on the result. Individual points on the graph are the max of the max for 10 runs for each problem size. Figure 1 is the absolute error and Figure 2 is the relative error. The relative error for GotoBLAS in Figure 2 appears to improve because the tile size chosen by GotoBLAS is targeting a larger problem size, effecting the way the computation is broken up. One interesting side effect of (3.4) is when $b = \sqrt{K}$, which is being illustrated in Figure 2 as the problem size approaches b^2. These results are particularly striking since the CPU implementation is using a floaing-point-multiply-and-add (FMADD) [12, 14, 15] whereas the GPU implementation is using a fused-multiply-and-add (FMA) [20]. How peculiar this result at first appears becomes more apparent when directly comparing the two operations using the standard model. In the standard model, an FMA is modeled as [13]:

$$fl(x + y \times z) = (x + y \times z)(1 + \delta), \tag{3.5}$$

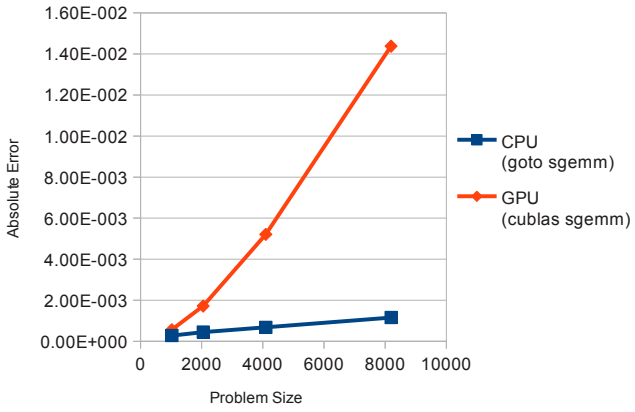

Figure 1: Single - Uni Random [0,1] - Absolute Error

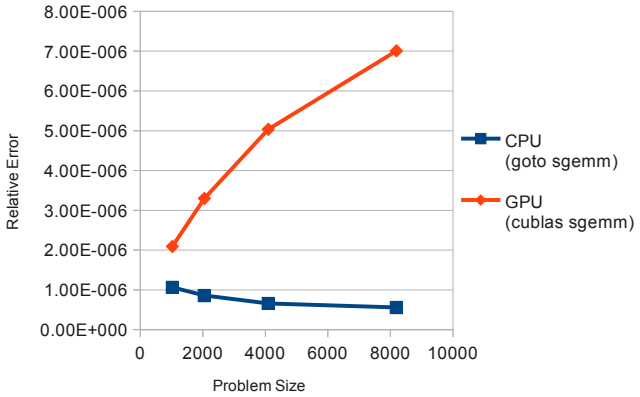

Figure 2: Single - Uni Random [0,1] - Relative Error

where $|\delta| \leq u$. What this means is that an FMA effectively commits roughly half as many rounding errors as the FMADD since the result is only rounded once, after the addition, instead of twice as is the case with the FMADD (once after the multiplication, once after the addition). However, the worst case forward error for a dot product computed using an FMA is still the same as 3.3. A simple analysis is as follows:

Let us assume we have two vectors, y and z, both of length n and a scalar variable x. Using the standard model, an FMA can be expressed with equation 3.5. Applying this model to the first two products we have as follows:

$$\widehat{s_K} = (x + y_1 \times z_1)(1 + \delta)$$

$$\widehat{s_K} = [(x + y_1 \times z_1)(1 + \delta) + (y_2 \times z_2)](1 + \delta)$$

$$\widehat{s_K} = (x + y_1 \times z_1)(1 + \delta)^2 + (y_2 \times z_2)(1 + \delta)$$

It is easy to see that if this pattern was carried out to n elements, the result would be:

$$\widehat{s_K} = (x + y_1 \times z_1)(1 + \delta)^K + (y_2 \times z_2)(1 + \delta)^{K-1} ... + (y_K \times z_K)(1 + \delta)$$

Using the same analysis that was used to derive the inner product forward error bound for the canonical matrix multiply algorithm [13], we end up with the same forward error bound as represented by equation 3.3. What this means is that the inner product computed using an FMADD and the inner product computed using an FMA have the same worst case forward error bound. More importantly, both are clearly worse than the forward error bound represented by equation 3.4, which is empirically demonstrated in Figures 1 and 2 by comparing the CUBLAS SGEMM implementation against the SGEMM implementation in GotoBLAS [12]. This is because the SGEMM implementation in GotoBLAS is broken up along the K dimension of the matrix product whereas the computation is not broken up along the K dimension in the CUBLAS implementation. This is because the GotoBLAS implementation is blocked on all three dimensions of the matrix product whereas the GPU implementation is only blocked on the result matrix, dimensions M and N. (For reference, GotoBLAS[1], the Tesla chipset implementation by Volkov and Demmel[2] as well as the MagmaBLAS implementation[3] for the Fermi chipset are publicly available.) This means the accuracy difference between the two SGEMM implementations is almost entirely algorithmic and not architectural. In the next subsection we offer a simple example to help illustrate this analysis.

[1]http://www.tacc.utexas.edu/tacc-projects/gotoblas2

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

Figure 3: Blocked Matrix Multiply

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

Figure 4: Preload Matrix Multiply

3.4 Matrix Multiply Blocking Example

Blocking is a common strategy within high performance matrix multiply kernels. The primary goal of blocking is to maximize reuse of the data in the cache and hence improve performance. Subtleties in how the individual blocks are added to the whole greatly impact the forward error bound on the inner product [13]. This difference is sometimes referred to as *preload* versus *postload* [7]. We adopt this terminology in this paper.

Preload and postload are best discussed in the context of a specific blocking strategy. Consider the code shown in Figure 3, 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 blocks are square, of size $bSize \times bSize$. For simplicity, we assume that the matrix dimensions are all even multiples of the block size. The strategy in Figure 3 is to compute C one column of blocks at a time, always keeping in memory a $bSize \times k$ strip of A along with the block of C currently being computed. The code repeatedly multiplies a block of A by a block of B and adds the result to a block of C. Each partial block computation is performed by calling the procedure

`leafMultiply(`A`,`ldA`,`B`,`ldB`,`C`,`ldC`,`M`,`K`,`N`)`

where A,B, and C are the addresses of the start of the blocks; the blocks 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 3 depends on

[2]http://www.cs.berkeley.edu/~volkov/
[3]http://icl.cs.utk.edu/magma/software/index.html

```
ALGORITHM: Direct Matrix Multiply (Postload)
procedure leafMultiply(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
```

Figure 5: Postload Matrix Multiply

how the inner kernel (i.e., the `leafMultiply()` procedure) is implemented. If the implementation is the preload code shown in Figure 4, the summation of the inner product is the standard summation of K terms, for which the forward error bound is modeled by equation 3.3. In contrast, the postload code shown in Figure 5 computes the contribution of the block to the inner product in a temporary variable and adds this sum to the inner product. When the code of Figures 3 and 5 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. As discussed in the previous subsection, breaking apart the summation affects the accuracy. By applying equation 3.4, the forward error bound for the postload implementation becomes:

$$|s_K - \widehat{s_K}| \leq \gamma_{\frac{K}{bSize}+bSize-1}|x|^T|y| \tag{3.6}$$

Which obviously produces a smaller forward error than the preload implementation for values of $bSize$ that are not equal to 1 or K.

Current GPU implementations on both the Tesla chipset and the Fermi chipset effectively use a preload implementation [1, 22], as illustrated in Figure 4. The forward error bound for the preload implementation is exactly the same as a standard summation, represented by equation 3.3 in Section 3.3. Notice that there is no difference in error between blocking on k and using a preload implementation versus using a postload implementation and changing the update rule in the inner most loop of Figure 3 from "$k = k + bSize$" to "$k = k + 1$". This is what GPU implementations do[2,3]. This implementation detail is in contrast to CPU implementations that effectively use a postload implementation [12, 23], as illustrated in Figure 5. The forward error bound for the postload implementation is represented by equation 3.4 in Section 3.3. This difference has a significant effect on the forward error bound, as was analyzed in Section 3.3 and demonstrated empirically in Figures 1 and 2. In the next section we will discuss algorithmic solutions to improve accuracy on the GPU.

4. ALGORITHMIC SOLUTIONS

As we discussed in Section 3, in order to improve the accuracy of a dot product the computation must be broken up. The most obvious way to break up the inner product is by blocking on k [7]. This is the preferred method for processors with deep memory hierarchies like the CPU as it maximizes cache hits which maximizes reuse of the data, thus avoiding costly trips to main memory [12, 23]. As we demonstrated in our previous work on the CPU, a similar effect can be achieved by breaking up the computation by using the outer product [5]. In general though, and in particular for flat memory hierarchy machines such as that provided by the GPU, blocking on k general leads to poorer performance since any resources allocated for blocking along k could be used to further improve performance as blocking on k has to be simulated with additional registers. Likewise, if the high performance matrix multiply implementation on the GPU is anywhere near 100% utilization of the GPU, you must steal resources from the implementation, reducing performance to increase accuracy. Applying the outer product recursively though requires no additional resources nor does it apply additional operations. The overhead of recursion is entirely limited to how the GPU driver handles multiple kernel launches, namely, if it inserts barriers between kernel launches that access disparate pieces of data. We discuss outer product recursion in detail in the following subsection.

4.1 Outer Product - Recursion

An alternative approach to breaking up the computation is by recursively breaking apart the matrix and computing the outer product. This can be accomplished as follows, given the following matrices:

$$A = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix}, \quad B = \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}, \text{ and}$$

$$C = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}$$

the entries of C are given by

$$\begin{aligned} C11 &= A11 \times B11 + A12 \times B21, \\ C12 &= A11 \times B12 + A12 \times B22, \\ C21 &= A21 \times B11 + A22 \times B21, \text{ and} \\ C22 &= A21 \times B12 + A22 \times B22. \end{aligned}$$

Each submatrix of A, B and C can likewise be broken up, yielding:

$$A11 = \left[\begin{array}{cc|cc} A11_{11} & A11_{12} & A11_{13} & A11_{14} \\ A11_{21} & A11_{22} & A11_{23} & A11_{24} \\ \hline A11_{31} & A11_{32} & A11_{33} & A11_{34} \\ A11_{41} & A11_{42} & A11_{43} & A11_{44} \end{array}\right],$$

where the first half of $C11_{11}$ is now computed by

$$\begin{aligned} C11_{11} = (A11_{11} \times B11_{11}) + (A11_{12} \times B11_{21}) + \\ (A11_{13} \times B11_{31}) + (A11_{14} \times B11_{41}) \end{aligned}$$

In this way, you are effectively halving the error with each additional level of recursion, a special case of this being pairwise summation [13, 5]. What is important here to realize is that this recursive decomposition need not be done with simply the canonical $O(N^3)$ matrix multiply algorithm, this can be done with any asymptotically fast matrix multiply algorithm that can be expressive recursively. Normally this is not pursued because as seen in Figures 1 and 2 Goto's implementation of SGEMM is very accurate and simply reducing the leaf size to even $\frac{K}{8}$, the result of applying a recursive matrix multiply implementation to a depth of three, is not

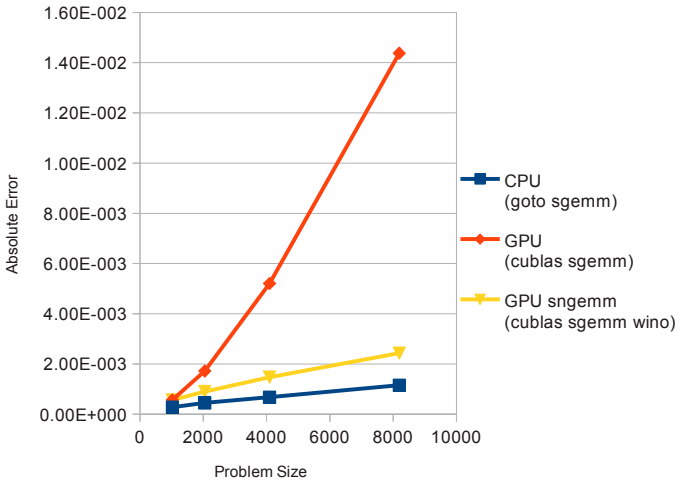

Figure 6: Single - Uni Random [0,1] - Absolute Error

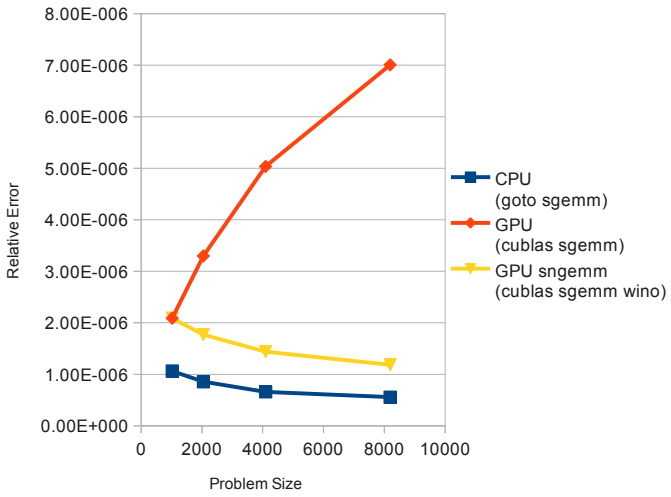

Figure 7: Single - Uni Random [0,1] - Relative Error

enough to out run the constant error that an asymptotically fast matrix multiply algorithm multiplies for each level of recursion. However, this is not the case for preload implementations like those found on GPUs, as we will discuss in the next subsection.

4.2 Winograd

Winograd's variant of Strassen's algorithm, which will be referred to as Winograd's algorithm, is an asymptotically fast matrix multiply algorithm that changes the schedule of operations in Strassen's algorithm, thereby reducing the number of additions in Strassen's algorithm [24]. Additionally, both algorithms only require 7 matrix multiply operations whereas the canonical $O(N^3)$ algorithm requires 8, making them asymptotically faster than the canonical matrix multiply algorithm. Given the same matrices as in Subsection 4.1, the specific schedule we use for Winograd's algorithm (as there are many different schedules and each one has different effects on accuracy) is described in Table 1. The important thing to notice is that for each multiplication in Winograd's algorithm, when the final value is written to matrix C it is done so with either an assignment or through addition, never subtraction. This includes the temporary value U. What this means is that if Winograd's algorithm is applied to input matrices that are all of the same sign, or in this case, every element in both matrix A and B is non-negative, then there is only cancellation if the sub-product is negative. This is commonly referred to as lacking "true" subtraction, as subtraction only occurs if the submatrix being added contains elements of a different sign than the matrix it is being added to. This means that though the overall worst case accuracy bound of Winograd's algorithm has not changed (as the bound makes no assumption on the input), in practice, the error is contained within the least significant bits as catastrophic cancellation through subtraction (actually using the subtraction operator when applying the partial sum to the result matrix C) is generally avoided. This allows Winograd's algorithm, in practice, to behave more like the canonical matrix multiply algorithm when the input is uniform (in this case, uniformly random in $[0, 1]$). Winograd's algorithm has been demonstrated to be, in practice, more accurate than Strassen's algorithm in previous work [5, 11] when the input is uniform random in $[0, 1]$. The key difference here is that since the underlying high performance implementation on the GPU is a preload implementation, the recursive decomposition of Winograd's algorithm breaks up the chains of additions. This effect is not as pronounced when the high performance implementation blocks on K, as most CPU implementations do [12, 23]. The reason for this is there is only really one source of rounding errors but two ways it can be exposed. The first way is if there is some small error in the least significant digits that is suddenly exposed by subtraction, that the least significant digits become the most significant digits (this is commonly referred to as catastrophic cancellation). The second way, and only true source of rounding errors, is the slow accumulation of rounding errors from each individual operation. The pace of this march can be slowed though by breaking up the chains of operations, as we explained in Subsection 4.1.

The implication of this insight can be clearly seen in Figures 6 and 7, (performance for which is provided in Figure 8), where Winograd's variant of Strassen's algorithm (labeled as 'GPU sngemm' and 'GPU dngemm', respectively) is applied to a leaf size of 1024 and the input is between $[0, 1]$. Additional levels of Winograd's algorithm actually improve the accuracy as compared to the implementation offered by Nvidia's CUBLAS. This improvement gives back a digit of accuracy over using Nvidia's CUBLAS, as seen in Figure 6. Though the accuracy of working on the GPU is still worse than that of the CPU, it is much improved over all other implementations offered on the GPU. It is also important to note that blocked postload CPU implementations like Goto's are amongst the most accurate high performance implementations [5]. In order to improve accuracy beyond that provided by blocked postload high performance implementations, such as that provided by GotoBLAS [12], requires a three level hybrid to maintain performance [5] or can be done using pairwise summation at the cost of performance [13, 5].

The benefit of choosing to use Winograd's algorithm over the canonical $O(N^3)$ algorithm is also obvious as seen by the performance gained in Figure 8. The GPU in the general case is faster than the CPU for matrix multiply [1] and we further extend this lead. In the next section we will

Winograd's Algorithm	
Multiplication #	Operations
1	$S = A21 + A22, T = B12 - B11,$ $U = S \times T,$ $C12 = U, C22 = U$
2	$U = A11 \times B11$ $C11 = U$
3	$C11 += A1 \times B21$
4	$S = S - A11, T = B22 - T$ $U += S \times T$ $C12 += U$
5	$S = A12 - S$ $C12 += S \times B22$
6	$T = B21 - T$ $C21 = A22 \times T$
7	$S = A11 - A21, T = B22 - B12$ $U += S \times T$ $C22 += U, C21 += U$

Table 1: Winograd's Variant of Strassen's Algorithm

demonstrate and discuss the benefits to both performance and accuracy to using Winograd's algorithm on the GPU.

5. EMPIRICAL RESULTS

In this section we will demonstrate empirically that it is possible to get both performance and accuracy when using hybrid matrix multiply algorithms, we demonstrate this on the GPU. The results were computed on Kubuntu 11.04 with CUDA version 4.1 using a Nvidia Tesla C2070. The CPU is a Q9450 Penryn Intel Quad Core processor using GotoBLAS2 version 1.13. The performance was measured by comparing the various implementations to an $O(N^3)$ implementation, assuming $2 \times N^3$ operations. Timing was done by using CUDA Events. The matrices computed are square for convenience. Accuracy was measured using relative error and absolute error [13], which is discussed at length in Section 3.1. Winograd's variant of Strassen's algorithm is labeled as 'GPU sngemm' for single precision matrix multiply and 'GPU dngemm' for double precision matrix multiply. An explanation of this naming scheme is provided in Section 6. For our implementation we wrote custom CUDA kernels for assignment, addition and subtraction. Our algorithm is implemented recursively using the schedule in Table 1, calling the CUBLAS GEMM implementation in the leaves for the multiplications. In the next two subsections we will provide and discuss additional performance and accuracy data, including for problems computed in double precision.

5.1 Performance

As seen in Figures 8 and 9, the Winograd variant of Strassen's algorithm clearly scales better than the canonical implementation offered by Nvidia, offering up to 26% better performance in single precision and up to 27% better performance in double precision. The key difference though between the single and double precision implementation is that of the crossover point. As seen in Figure 8, the break even point is a leaf of 1k whereas the break even point for double precision is 2k, as seen in Figure 9. This makes perfect sense given the size difference between single and double precision,

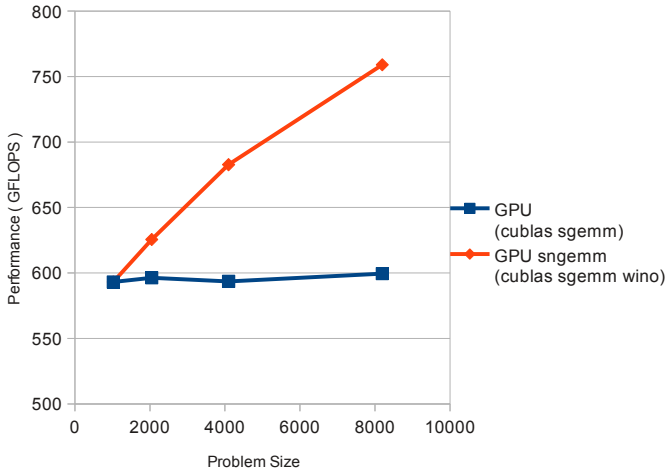

Figure 8: Single - Performance (GFLOPS)

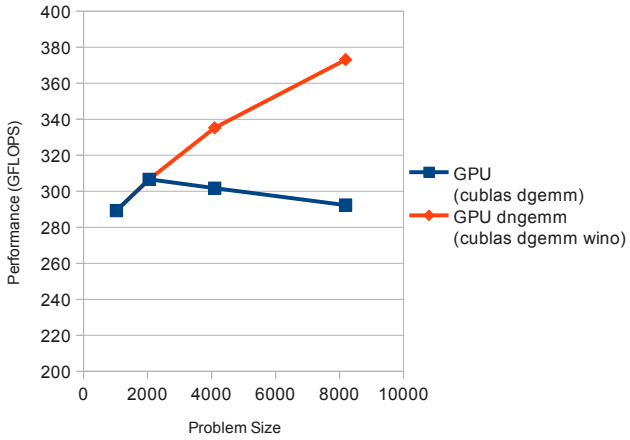

Figure 9: Double - Performance (GFLOPS)

namely, the cost associated with trading multiplications for additions and the additional overhead in bandwidth associated with the additions. To be explicit, it costs more to add matrices in double precision than it does if they are stored in single precision, so the time saved on the multiplication needs to be greater to offset the time lost doing the additions. Thus the crossover point for double precision is a leaf size of 2k whereas the crossover point for single precision is 1k. In either case though, the performance benefit of applying Winograd's variant of Strassen's algorithm is obvious.

5.2 Accuracy

As seen in Figures 10 and 11, double precision behaves identically in terms of absolute and relative error to that of single precision, as introduced in Subsection 4.2 and seen in Figures 6 and 7. Though the overall accuracy of this solution is not better than that of the CPU, if one is choosing to use the GPU for performance reasons, the solution provided in this paper is much more accurate than simply using the standard matrix multiply implementation provided. This difference can be quite large, as seen in Figure 10, for a problem size of 8k, a whole digit of accuracy is returned. In

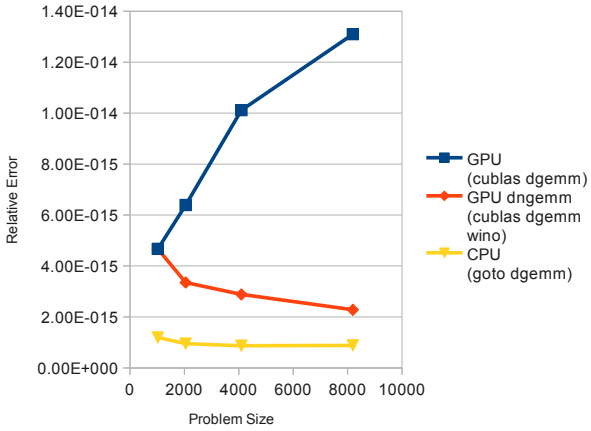

Figure 10: Double - Random [0,1] - Relative Error

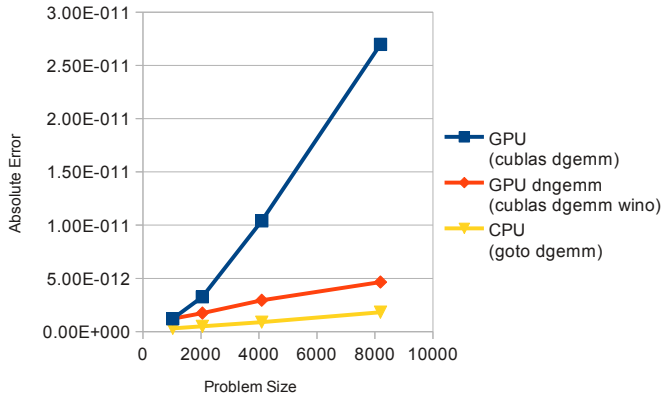

Figure 11: Double - Random [0,1] - Absolute Error

the next section we will discuss expanding the Basic Linear Algebra Subroutines level 3 (BLAS 3) specification in order to offer a standardized method for library authors to implement, as opposed to forcing the general scientific community to reimplement various solutions, or worse, to constantly reorganizing the structure of their programs as new algorithms are implemented using different, incompatible standards.

6. BLAS EXTENSION

It should be clear by now that there are many problems that have non-negative matrix inputs and would benefit from solutions like those offered in this paper. The problem however is the lack of a standard makes adoption unlikely as new implementations will offer new function definitions, forcing the general scientific community to constantly update their program to match the new implementation. In addition, simply replacing an existing BLAS (Basic Linear Algebra Subroutines) level 3 function introduces confusion and creates a conflict with the existing function and limits the new implementation to programs that do not require both the new and original functionality of the method being replaced. The current BLAS level 3 specification is organized as follows:

In the existing BLAS level 3 specification, the functions are named according to the following scheme: One letter for type, followed by two letters specifying something special about the input, for instance, if the input is symmetric, followed by two letters specifying the operation performed. The types supported are 'S' for single precision, 'D' for double precision, 'C' for single complex and 'Z' for double complex. The options that specify something about the input are:

GE General Matrix

HE Hermitian Matrix

SY Symmetric Matrix

TR Triangular Matrix

To give an example, this means a function called DGEMM would translate into a function that performs an operation on Double precision, on a GEneral matrix input, and the operation performed is Matrix Multiply. What we wish to do is provide a narrow extension to the input specification. Over the years, several broader extensions to BLAS have been proposed and implemented [19, 6]. What we aim to do is simply avoid confusion with the existing methods already provided. Keeping with the existing nomenclature, we propose the addition of 'N' to specify non-negative inputs. This 'N' would be inserted between the type of input and the two letter option that further specifies something about the input. In this way, we can allow for both non-negative and symmetric inputs, for instance. Using the DGEMM example from before, this means a new function would be available called DNGEMM, translating into a function that performs on an input that is Double precision, Non-Negative and performs GEneral Matrix Multiply. The benefit of adapting this existing standard as opposed to proposing an entirely new standard is that current scientific programs need only change the name of the function being called in order to gain the benefit of implementations like those provided in this paper as the rest of the function prototype is identical.

7. CONCLUSION

In this paper we have explained and demonstrated empirically that the simple addition of an FMA (Fused-Multiply-Add) will not necessarily result in better accuracy, specifically with regards to matrix multiply. We demonstrated this by comparing high performance implementations on two different architectures, a GPU that provided an FMA, and a CPU that did not. In Subsection 3.3 we showed that surprisingly, the architecture that lacked an FMA, the CPU, actually had better accuracy with regards to matrix multiply than that of the architecture that did have an FMA, the GPU. Next, we analyzed why using the standard model in Section 3, and explored algorithmic solutions to improve the accuracy of matrix multiply on the GPU in Section 4, proposing Winograd's variant of Strassen's algorithm in Subsection 4.2 for improving the accuracy of non-negative matrices. Next, in Section 5, we empirically verified our solution, demonstrating that not only can our solution improve accuracy over the standard high performance implementation provided by Nvidia, it can also improve performance by up to 26% in single precision and 27% in double precision. Finally, in Section 6, we proposed to extend the BLAS (Basic Linear Algebra Subroutines) level 3 specification to include non-negative matrix inputs, 'N', thus allowing library authors to offer their own solutions under an already

accepted standard, thereby enabling the general scientific community to easily transition existing programs to using this new solution.

8. ACKNOWLEDGMENTS

We wish to thank the reviewers for their thorough and thoughtful comments and suggestions. In particular, reviewer 1 for the extremely helpful and insightful comments which greatly improved the quality of the paper. We would also like to thank the National Science Foundation for partial support of this research, NSF Grant # 1249449.

9. REFERENCES

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, 2009.

[2] T. Akutsu, S. Miyano, and S. Kuhara. Algorithms for identifying boolean networks and related biological networks based on matrix multiplication and fingerprint function. *Journal of Computational Biology*, 7(3-4):331–343, 2000.

[3] I. J. Anderson. A distillation algorithm for floating-point summation. *SIAM J. Sci. Comput.*, 20(5):1797–1806, 1999.

[4] M. Badin, L. Bic, M. B. Dillencourt, and A. Nicolau. Pretty good accuracy in matrix multiplication with gpus. In *ISPDC*, pages 49–55. IEEE Computer Society, 2010.

[5] M. Badin, P. D'Alberto, L. Bic, M. Dillencourt, and A. Nicolau. Improving the accuracy of high performance BLAS implementations using adaptive blocked algorithms. In *Proceedings of 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 120–127, 2011.

[6] L. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002.

[7] 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.

[8] E. Cohen and D. D. Lewis. Approximating matrix multiplication for pattern recognition tasks. In M. E. Saks, editor, *SODA*, pages 682–691. ACM/SIAM, 1997.

[9] B. A. Craig and P. P. Sendi. Estimation of the transition matrix of a discrete-time markov chain. *Health Economics*, 11(1):33–42, 2002.

[10] P. D'alberto, M. Bodrato, and A. Nicolau. Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems: Matrix-multiplication and matrix-addition algorithm optimizations by software pipelining and threads allocation. *ACM Trans. Math. Softw.*, 38(1):2:1–2:30, Dec. 2011.

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

[12] 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.

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

[14] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2A: Instruction Set Reference A-M*. Intel Corporation, June 2010.

[15] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2B: Instruction Set Reference N-Z*. Intel Corporation, June 2010.

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

[17] V. V. Kindratenko, J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. mei W. Hwu. Gpu clusters for high-performance computing. In *CLUSTER*, pages 1–8. IEEE, 2009.

[18] J. Li, S. Ranka, and S. Sahni. Strassen's matrix multiplication on gpus. In *Proceedings of 17th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 157–164. IEEE, December 2011.

[19] X. S. Li, J. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. Thompson, T. Tung, and D. J. Yoo. Design, implementation and testing of extended and mixed precision blas. *ACM Trans. Math. Softw.*, 28(2):152–205, 2002.

[20] Nvidia. *NVIDIA CUDA C Programming Guide*. NVIDIA, 4.1 edition, November 2011.

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

[22] V. Volkov and J. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the ACM/IEEE Conference on High Performance Computing*, page 31. IEEE/ACM, November 2008.

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

[24] S. Winograd. On the multiplication of 2 x 2 matrices. *Linear Algebra and its Applications*, 4(4):381 – 388, October 1971.

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