

Pretty Good Accuracy in Matrix Multiplication with GPUs

Matthew Badin
Computer Science

UCI

Irvine, USA

Email: mbadin@uci.edu

Lubomir Bic
Computer Science

UCI

Irvine, USA

Email: bic@ics.uci.edu

Michael Dillencourt
Computer Science

UCI

Irvine, USA

Email: dillenco@ics.uci.edu

Alexandru Nicolau
Computer Science

UCI

Irvine, USA

Email: nicolau@ics.uci.edu

Abstract—With systems such as Road Runner, there is a trend in super computing to offload parallel tasks to special purpose co-processors, composed of many relatively simple scalar processors. The cheaper commodity class equivalent of such a processor would be the graphics card, potentially offering super computer power within the confines of a desktop PC. Graphics cards however are not without problems, these range from the lack of double precision on most cards to a fairly steep drop in performance for using double precision on others, the end result being that in order to utilize the graphics card the computation must be done using single precision. In this paper we propose a method whereby a whole digit of the accuracy lost in single precision matrix multiply can be regained with only a 7% loss in performance by applying a compensated summation algorithm in a manner previously unexplored, a manner in which, at first glance, shouldn't provide any benefit but empirical evidence will show that though the novel idea is simple, provides unexpected benefits in terms of accuracy at little cost to performance.

Keywords-GPGPU; Matrix Multiplication; Floating Point; Compensated Summation

I. INTRODUCTION

The graphics card is becoming increasingly attractive to scientific computing, lured by untold performance gains and relatively inexpensive hardware there is significant work in transitioning traditional scientific computing problems from the CPU to the GPU [1], [2], [3]. Graphics cards however are not without problems, these range from the lack of double precision on most cards to a fairly steep drop in performance for using double precision on others, the end result being that in order to utilize the graphics card the computation must be done using single precision [4]. In scientific computing in particular, parallel systems are used to simulate physical phenomena, consequently, the validity of the results, like any other experiment, is based upon the reproducibility of the experiment. One way to improve numerical stability and reproducibility (particularly when precision is limited, as is the case with GPUs) is by using an accurate summation algorithm [5]. Summation in general is particularly susceptible to rounding errors [6]. Consequently, in this paper we will explore the cost of accurate computation on Nvidia GPUs with a special focus on summation which we will demonstrate through matrix multiply. We will present a novel way of applying

doubly compensated summation in a selective manner, that when applied to matrix multiply will gain a whole digit of accuracy as compared against Nvidia BLAS, this is as opposed to doing the computation on the CPU (described in detail in section 3). Our results will further show that if the FMADD instruction (floating point multiply-and-add) on Nvidia GPUs were to conform with the IEEE 754 standard then we could generate a solution very close to the gold standard (the gold standard is described in detail in section 3.1), further diminishing the cost of trading accuracy for performance (double precision for single precision) when transitioning projects from the CPU to the GPU. Furthermore, this is all possible with only a 7% performance loss. The rest of this paper is organized as follows: First we will discuss accurate summation methods followed by an explanation of doubly compensated summation. We will then introduce selective doubly compensated summation along with how best to apply it to matrix multiply. We will finish by describing how the error analysis was conducted, how the FMADD implementation on Nvidia GPUs affected our error analysis, and finally what can be gained by applying selective doubly compensated summation.

II. ACCURATE SUMMATION

There are many ways to recover or prevent the rounding error introduced during summation in matrix multiply. These methods of providing accurate summation can be broken into two very broad categories, those that require special purpose hardware and those that don't. Since the focus of this paper isn't to propose additional hardware for inclusion in the next generation of GPUs, we'll ignore solutions that require additional hardware (or changing the hardware to include extra precision). The software solutions can be broken down further into two families, distillation and compensated summation. Even though distillation algorithms offer the promise of exact arithmetic, we ruled them out early on because of the computational cost [7] and because they focus primarily on ill-conditioned data (data that has heavy cancellation) [8]. Simulating extra precision in software wasn't considered because of the poor performance, "that such simulation is too expensive to be of practical use for routine computation" [6]. This leaves us with compensated summation algorithms

which are, by comparison, less computationally expensive and we will show (in section 3.3) can be applied in such a way as to yield most of the benefit they offer in terms of accuracy without paying for the cost in performance usually associated with accurate summation methods. In addition, and as noted by other authors, “in practice, compensated summation performs well with most data sets and will frequently give results that are better than the method of recursive [standard] summation and its variants” [9]. Within the family of compensated summation algorithms, we chose doubly compensated summation as it is more accurate than Kahan’s compensated summation [10], however, at a cost of increased number of operations. What follows is the algorithm for doubly compensated summation [10].

A. Doubly Compensated Summation

ALGORITHM: Doubly Compensated Summation

```

1  procedure dcsum( $n, x_1, \dots, x_n$ )
2  begin
3   $s_1 = x_1, c_1 = 0$ 
4  for  $k = 2, \dots, n$ 
5       $y_k = c_{k-1} + x_k$ 
6       $u_k = x_k - (y_k - c_{k-1})$ 
7       $t_k = y_k + s_{k-1}$ 
8       $v_k = y_k - (t_k - s_{k-1})$ 
9       $z_k = u_k + v_k$ 
10      $s_k = t_k + z_k$ 
11      $c_k = z_k - (s_k - t_k)$ 
12 return  $s_n$ 
13 end
```

Figure 1.

Figure 1 contains the pseudo code for doubly compensated summation, where n is the number of summands and x_1, \dots, x_n are the summands, (in its most general form, assumed to be sorted in decreasing order by absolute value). If the input is not sorted, it is important that the left summand is larger than the right and consequently the algorithm must be changed to include a comparison and swap to test and correct for this case. The algorithm is attempting to compensate for the rounding error introduced when adding two numbers together, namely the next summand of the dot product and the partial sum (which will end up being the final result for an element of the result matrix). It does this by first adding the previous rounding error (the rounding error from the last iteration) to the next summand (line 5) followed by attempting to calculate the rounding error of that previous operation (line 6). The algorithm then adds the current summand (with the compensation already added) to the partial sum (line 7) followed by calculating the rounding error of this previous operation (line 8). The algorithm then adds the two rounding errors (that of the summand and the

partial sum and that of the summand and the compensation) together (line 9). The algorithm finishes by then adding this summed error to the partial sum (line 10) and calculating the new compensation for the next iteration of the algorithm (line 11), hence the name, doubly compensated summation as it attempts to compensate for the error twice, once with the new summand and once with the result of the summation. This differs from Kahan’s compensated summation as his algorithm only compensates once by adding the error to the new summand [11].

B. Selective Application of Doubly Compensated Summation

As noted at the top of the section, doubly compensated summation is rather expensive, requiring 10 operations whereas standard (recursive) summation only requires one. This leads to a severe performance drop of approximately an order of magnitude when applying doubly compensated summation. What we propose instead in this paper is a possible trade off, a trade off between performance and accuracy. This is done by not replacing every ordinary addition with doubly compensated summation but rather by *selectively* applying doubly compensated summation after every fixed number of standard additions. This allows for one to trade a certain amount of numerical stability for performance, most of the numerical instability residing in the recursive summation that occurs in between the applications of doubly compensated summation. As will be discussed in detail in section 3, surprisingly, it is possible to capture most of the benefit of doubly compensated summation without incurring the cost, when it is applied selectively. Figure 2 contains the pseudo code for *selective* doubly compensated summation.

ALGORITHM: Selective Doubly Compensated Summation

```

1  procedure selective_dcsum( $k, n, x_1, \dots, x_n$ )
2  begin
3   $partialSum = 0, total = 0$ 
4  for  $i = 1, \dots, n \div k$ 
5      for  $j = (i - 1) \times k, \dots, \max(i \times k, n)$ 
6           $partialSum = partialSum + x_j$ 
7      end for
8      if ( $|total| \geq |partialSum|$ )
9           $total = dcsum(2, total, partialSum)$ 
10     else
11          $total = dcsum(2, partialSum, total)$ 
12      $partialSum = 0$ 
13 end for
14 return  $total$ 
```

Figure 2.

To put the idea into more concrete terms, the algorithm would work as follows: Given a set of summands, k sum-

mands would be recursively summed (lines 5-7), where k is given (decided by the user). This partial sum would then be added to the total, using doubly compensated summation (lines 8-11). This would repeat until all of the summands have been added to the total. The special case of this algorithm being where $k = 1$, which would simply be doubly compensated summation. It should be obvious that most of the error will be introduced when the partial sum is being produced (lines 5-7) as this places the algorithm at the mercy of the instability of recursive summation, where the error is completely dependent upon the order in which the elements are added and the difference in magnitude between the two elements being added. In practice however, as noted by other authors and as will be shown below, the error is relatively small [12], [13]. This is largely due to the relatively small size of k we will be using relative to the number of elements being summed. The size of k is also partially limited by the relatively small amount of shared memory available on the GPU [14]. This idea does however afford one with an easy lever for determining how much performance or accuracy is desired, and as such, can be easily tuned for specific applications, up to the limitations of doubly compensated summation.

C. Applying Selective Doubly Compensated Summation to Matrix Multiply

The implementation of SGEMM (single precision general matrix multiply) that was adapted for use with selective doubly compensated summation was originally developed by Volkov and Demmel [15]. The reason this algorithm was chosen is that it was designed for Nvidia GPUs, is written for the CUDA environment, and that to the best of our knowledge it is the basis for the SGEMM implementation used by Nvidia in their BLAS (basic linear algebra subroutines) implementation since CUDA 2.0. Unfortunately Nvidia hasn't released their BLAS implementation since CUDA version 1.1, therefore we had to verify empirically whether or not the implementation proposed by Volkov and Demmel was still being used. This was necessary so that we could make a fair comparison between our implementation of SGEMM and that provided by Nvidia BLAS, that way we could be certain that any improvement in accuracy or cost in performance was not that of the SGEMM implementation we were adapting, but rather by our own contribution. The metrics employed for purposes of verification were performance and accuracy, for simplicity, we only tested square matrices. When the size of the matrix was a multiple of 32, the performance and accuracy was identical (the performance being measured using CUDA Events), consequently, all matrices used in comparison of accuracy and performance in this paper will always be square and a multiple of 32. As for the implementation of SGEMM written by Volkov and Demmel and how it is adapted for selective doubly compensated summation, it is organized as follows:

Thread Count vs Tile Size			
Thread Count	16x16 Tile	32x16 Tile	64x16 Tile
64	315.8 GFlops	368.2 GFlops	375.4 GFlops
128	319.9 GFlops	376.5 GFlops	412.3 GFlops
256	311.8 GFlops	370.3 GFlops	408.0 GFlops

Table I

Given the following matrices: $A \times B = C$, the algorithm loads a 16 x 16 tile of B into shared memory. The algorithm then loads a 64 x 16 tile of A (64 x 1 at a time) and then each thread multiplies one element of row A against 16 columns of B. This is repeated until all 16 elements of A have been multiplied against all 16 columns of B. At this point the algorithm must then load the next 16 x 16 tile of B from the GPU's global memory, this is where we decided to apply selective doubly compensated summation. It is important to note however that this idea need not be tied directly to the tile size. In fact, the performance and accuracy is similar whether loading a 64x16 tile and applying doubly compensated summation selectively after every 16 multiplications (instead of 64 as the tile size suggests) versus loading a 16x16 tile and applying selective doubly compensated summation before the next tile is loaded. To simplify the analysis though, when selective doubly compensated summation is applied will be tied to the tile size. The other subtle point about this algorithm is that the number of elements of A that are loaded is related to the thread count, for instance, if 128 threads are used instead of 64, then a 128 x 16 tile of A would be loaded instead of 64 x 16 (128 elements at a time). Though the original algorithm uses 64 threads, this seemed contradictory as in general Nvidia recommends using at least 192 threads for any algorithm to hide pipeline latency [14]. Because of the subtlety just described, the suggestion made by the manufacturer, and the numerous data dependencies that reside in doubly compensated summation, we briefly explored the consequences of changing the thread count when applying selective doubly compensated summation. Table 1 displays the performance in GFlops when using this modified SGEMM implementation (the Volkov and Demmel implementation with selective doubly compensated summation applied) as it relates to thread count and tile size on a 5120 x 5120 matrix. The data was produced on a machine running Kubuntu 9.04 64-bit using an Intel Quad Core 2.66ghz processor, 8GB of RAM, Nvidia 285gtx, and CUDA 2.3.

As can be seen in table 1, the performance is similar, however, using 128 threads appears to slightly outperform using 64 or 256 threads. Consequently, the implementation of SGEMM that we use for error analysis also uses 128 threads. We didn't test 192 threads (as is recommended by Nvidia) simply because of the difficulty of adapting 192

threads to this algorithm and because the focus of this paper is not to produce empirical evidence of the Nvidia GPU’s pipeline length but rather the benefit of applying doubly compensated summation in a limited manner.

III. RESULTS

This section is organized as follows: First we will discuss the method by which the algorithm was analyzed (including how the gold standard was computed) followed by discussing some peculiarities of the GPU and how they affected our error and performance analysis, finishing with our error and performance data.

A. Method of Analysis

Our error analysis was done by comparing the result of matrix multiply computed on the Nvidia GPU to that of a gold standard generated on the CPU. The algorithm was tested with two different types of inputs, a random uniform distribution between $[0, 1]$ and $[-1, 1]$ respectively, which appears to be representative of most data sets as error bounds and ill-conditioned data represent only the worst case whereas random appears to be a better representative of the average case[6]. The performance of the GPU algorithms was measured using CUDA Events $((2 \times N^3) \div time)$, where N is the size of the matrix since we only test square matrices). The gold standard was computed using the naive $O(N^3)$ algorithm where the multiplication was still carried out in single precision whereas the summation was computed in double precision. This was done for two reasons: 1) As noted by previous authors, summation is where most of the rounding error occurs [6], [16]. 2) Doubly compensated summation only affects summation and not multiplication and we didn’t want to measure the accuracy of multiplication using double precision against the accuracy of single precision multiplication, especially when you consider that single precision multiply-add is not strictly IEEE compliant on the Nvidia GPU [4] and consequently is the topic of the next subsection.

B. Asymmetric Error Range

Early on in our testing we noticed an asymmetry in the error range, isolating the bug, the culprit is how Nvidia chose to implement floating point multiply-and-add (FMADD). The way FMADD is implemented on all current generation Nvidia GPUs is by truncating the intermediate result of the multiplication instead of properly rounding [4]. Table 2 illustrates the consequences, the matrix size is 1024 x 1024, on an input uniformly randomly distributed between $[0, 1]$, the algorithm used is the one proposed in section 2.3 (matrix multiply with selective doubly compensated summation). For comparison, table 3 is the same matrix (using the same algorithm), computed by separating the FMADD into two instructions (multiply and add), which forces the GPU to properly round the result (at a performance cost).

FMADD Truncation, $N = 1024$		
Tile Size	Avg Error	Error Range
16	0.000011	$[-0.000035, 0.000014]$
64	0.000012	$[-0.000063, 0.000043]$

Table II
ERROR ANALYSIS OF DOUBLY COMPENSATED SUMMATION USING FMADD

Proper Rounding, $N = 1024$		
Tile Size	Avg Error	Error Range
16	0.000006	$[-0.000024, 0.000024]$
64	0.000008	$[-0.000050, 0.000047]$

Table III
ERROR ANALYSIS OF DOUBLY COMPENSATED SUMMATION WITHOUT FMADD

By comparing the results of table 3 against a tile size of 1 in table 5 for $N = 1024$ (which means doubly compensated summation was applied to every operation), it is obvious that if FMADD was properly implemented, our algorithm would either match the gold standard for average error or be very close, depending on the tile size that was chosen. Of course the reason we don’t purposely keep FMADD separated is due to performance, the difference of separating one operation into two, 408.0 GFlops as one operation and 326.2 GFlops as two. To force a fair comparison between selective doubly compensated summation and Nvidia BLAS SGEMM, in terms of performance, we decided to leave the algorithm as is and keep the FMADD. The logic being that this is not a problem of the algorithm or doubly compensated summation or even how doubly compensated summation is being applied but rather how the single operation was implemented on the Nvidia GPU. It does however appear that future generations of the Nvidia GPU will implement the FMADD operation according to the IEEE 754 standard [17], so consequently the accuracy of selective doubly compensated summation will only improve. The problem with the implementation of FMADD however does not render our algorithm useless as it still performs much better in terms of error than not using it at all (at a very minor performance cost), as we will demonstrate in the next section.

C. Accuracy and Performance

To reiterate, the inputs that were tested were randomly distributed between $[0, 1]$ and $[-1, 1]$ respectively (which as noted in section 3.1 appears to be representative of the average case). The algorithm that was used was discussed in detail in section 2. To make sense of the tables, a tile size of 1 is the best one can do, doubly compensated summation is applied to every operation. Any further improvement would be limited by the stability of doubly compensated summation. A tile size of 1 represents the lower bound on $[0, 1]$ (for table 5) in terms of accuracy (i.e. if you

applied doubly compensated summation for every addition in single precision, this is the best you could do as compared against double precision, computed on the CPU). Table 4 is the error analysis of Nvidia’s BLAS implementation as compared against the gold standard (described in section 3.1). Table 5 is what is possible if you use selective doubly compensated summation as it relates to problem size and tile size on a random input between $[0, 1]$. For instance, if you compare $N = 5120$ in table 5, you can see the best you can do is an average error of 0.000031, Nvidia’s BLAS has an average error of 0.001129 for the same problem size (table 4), and selective doubly compensated summation achieves an average error of 0.000056 for a tile size of 64 (table 5), or roughly two extra digits of accuracy were recaptured as opposed to using Nvidia’s BLAS, which is the same average number of digits expected to be accurate even under ideal conditions (for $N = 5120$, table 5, tile size of 1). As can also be seen, selective doubly compensated summation consistently returns a whole digit of accuracy (not just in the average error, but also can return a digit of accuracy in the range for specific tile sizes, even with the asymmetry). It is unfortunate Nvidia’s decision to implement the FMADD instruction as they did. As already discussed in the previous section (section 2.2), you can see where it might have been possible to return two digits of accuracy in the range and not just the average (for matrix size of 5120 and tile size of 64 on an input of $[0, 1]$ in table 5) if the FMADD instruction had followed the IEEE 754 standard. A tile size of 64 appears to be a good choice as it offers the best performance at comparable accuracy (you don’t gain any additional digits of accuracy by using a smaller tile size). When comparing the performance of selective doubly compensated summation with a tile size of 64 (table 5) against the performance of vanilla SGEMM (table 4) for a problem size of 5120, there is only a 7% drop in performance. The larger performance difference (between tile sizes and between different SGEMM implementations) for smaller problem sizes is because the time it takes for the program to complete is dominated by the cost of launching the kernel onto the GPU and not the time it takes to compute the problem. This is compounded by the relatively short amount of time it takes to compute a small matrix (measured in the thousandths place), causing any small variation to have a disproportionate impact on the performance, a larger problem size is therefore required to get any meaningful performance results. Tables 6 and 7 cover the analysis in the $[-1, 1]$ range (table 6 is the error analysis of Nvidia BLAS in the $[-1, 1]$ and table 7 is the error analysis of selective doubly compensated summation also on $[-1, 1]$). As is evident, selective doubly compensated summation offers the same benefit and performance in the $[-1, 1]$ range as well as the $[0, 1]$.

Nvidia BLAS GPU Matrix Multiply on $[0, 1]$			
Size	Avg Error	Error Range	GFlops
1024	0.000088	$[-0.000551, 0.000534]$	429.5
2048	0.000246	$[-0.001657, 0.001773]$	440.5
3072	0.000511	$[-0.003693, 0.003278]$	439.3
4096	0.000691	$[-0.005259, 0.004995]$	441.9
5120	0.001129	$[-0.008014, 0.007527]$	442.2

Table IV

Selective Doubly Compensated Summation on $[0, 1]$				
N	Tile	Avg Error	Error Range	GFlops
1024	1	0.000006	$[-0.000016, 0.000016]$	59.7
	16	0.000011	$[-0.000035, 0.000014]$	306.8
	32	0.000011	$[-0.000045, 0.000020]$	357.9
	64	0.000012	$[-0.000061, 0.000039]$	357.9
2048	1	0.000011	$[-0.000032, 0.000032]$	61.6
	16	0.000021	$[-0.000065, 0.000023]$	318.1
	32	0.000022	$[-0.000078, 0.000034]$	373.5
	64	0.000023	$[-0.000104, 0.000055]$	409.0
3072	1	0.000015	$[-0.000033, 0.000033]$	62.1
	16	0.000031	$[-0.000080, 0.000018]$	320.3
	32	0.000032	$[-0.000095, 0.000031]$	376.5
	64	0.000033	$[-0.000126, 0.000061]$	411.2
4096	1	0.000023	$[-0.000064, 0.000064]$	62.1
	16	0.000043	$[-0.000120, 0.000038]$	319.6
	32	0.000044	$[-0.000141, 0.000057]$	375.5
	64	0.000045	$[-0.000173, 0.000092]$	411.5
5120	1	0.000031	$[-0.000065, 0.000065]$	62.2
	16	0.000052	$[-0.000136, 0.000033]$	320.3
	32	0.000054	$[-0.000161, 0.000052]$	376.5
	64	0.000056	$[-0.000200, 0.000089]$	412.3

Table V

Nvidia BLAS GPU Matrix Multiply on $[-1, 1]$			
Size	Avg Error	Error Range	GFlops
1024	0.000004	$[-0.000069, 0.000096]$	429.5
2048	0.000009	$[-0.000166, 0.000150]$	440.5
3072	0.000013	$[-0.000246, 0.000290]$	439.3
4096	0.000017	$[-0.000344, 0.000391]$	441.9
5120	0.000021	$[-0.000396, 0.000563]$	442.2

Table VI

D. Selective Kahan’s Compensated Summation

For sake of comparison, we also briefly explored selectively applying Kahan’s compensated summation[11]. As can be seen in tables 8 and 9, surprisingly, the technique performs as well using Kahan’s compensated summation as it does for doubly compensated summation in terms of accuracy (e.g. the same number of digits are restored when using either technique), even though doubly compensated summation should outperform Kahan’s compensated summation in terms of accuracy (selective doubly compensated summation isn’t fully sorted, though it is supposed to be [10], which maybe why Kahan’s compensated summation performs as well. However, there is other work that also suggests that Kahan’s compensated summation performs as well as doubly compensated summation in practice [18]). Kahan’s

Selective Doubly Compensated Summation on $[-1, 1]$				
N	Tile	Avg Error	Error Range	GFlops
1024	1	0.000000	$[-0.000003, 0.000003]$	59.7
	16	0.000001	$[-0.000006, 0.000006]$	306.8
	32	0.000001	$[-0.000007, 0.000007]$	357.9
	64	0.000001	$[-0.000009, 0.000009]$	357.9
2048	1	0.000001	$[-0.000004, 0.000004]$	61.6
	16	0.000001	$[-0.000010, 0.000009]$	318.1
	32	0.000001	$[-0.000012, 0.000011]$	373.5
	64	0.000002	$[-0.000014, 0.000013]$	409.0
3072	1	0.000001	$[-0.000006, 0.000006]$	62.1
	16	0.000001	$[-0.000011, 0.000010]$	320.3
	32	0.000002	$[-0.000014, 0.000013]$	376.5
	64	0.000002	$[-0.000017, 0.000017]$	411.2
4096	1	0.000001	$[-0.000006, 0.000006]$	62.1
	16	0.000002	$[-0.000012, 0.000013]$	319.6
	32	0.000002	$[-0.000016, 0.000015]$	375.5
	64	0.000003	$[-0.000022, 0.000020]$	411.5
5120	1	0.000001	$[-0.000009, 0.000007]$	62.2
	16	0.000002	$[-0.000014, 0.000016]$	320.3
	32	0.000002	$[-0.000017, 0.000018]$	376.5
	64	0.000003	$[-0.000022, 0.000023]$	412.3

Table VII

Selective Kahan's Summation on $[-1, 1]$				
N	Tile	Avg Error	Error Range	GFlops
1024	1	0.000000	$[-0.000003, 0.000003]$	119.3
	16	0.000001	$[-0.000006, 0.000005]$	358.0
	32	0.000001	$[-0.000007, 0.000007]$	358.0
	64	0.000001	$[-0.000010, 0.000009]$	397.7
2048	1	0.000000	$[-0.000004, 0.000004]$	125.4
	16	0.000001	$[-0.000010, 0.000009]$	373.5
	32	0.000001	$[-0.000011, 0.000011]$	409.0
	64	0.000002	$[-0.000014, 0.000014]$	426.3
3072	1	0.000001	$[-0.000006, 0.000006]$	126.3
	16	0.000001	$[-0.000011, 0.000011]$	376.5
	32	0.000002	$[-0.000014, 0.000013]$	411.2
	64	0.000002	$[-0.000018, 0.000017]$	432.7
4096	1	0.000001	$[-0.000006, 0.000006]$	126.7
	16	0.000002	$[-0.000012, 0.000013]$	377.6
	32	0.000002	$[-0.000015, 0.000016]$	414.0
	64	0.000003	$[-0.000021, 0.000020]$	433.6
5120	1	0.000001	$[-0.000007, 0.000007]$	126.9
	16	0.000002	$[-0.000015, 0.000014]$	379.1
	32	0.000002	$[-0.000017, 0.000017]$	414.9
	64	0.000003	$[-0.000025, 0.000024]$	433.0

Table IX

Selective Kahan's Summation on $[0, 1]$				
N	Tile	Avg Error	Error Range	GFlops
1024	1	0.000006	$[-0.000016, 0.000016]$	119.3
	16	0.000011	$[-0.000035, 0.000013]$	358.0
	32	0.000011	$[-0.000044, 0.000021]$	358.0
	64	0.000012	$[-0.000058, 0.000035]$	397.7
2048	1	0.000011	$[-0.000032, 0.000033]$	125.4
	16	0.000021	$[-0.000064, 0.000021]$	373.5
	32	0.000022	$[-0.000079, 0.000034]$	409.0
	64	0.000023	$[-0.000101, 0.000062]$	426.3
3072	1	0.000015	$[-0.000033, 0.000033]$	126.3
	16	0.000031	$[-0.000077, 0.000015]$	376.5
	32	0.000032	$[-0.000095, 0.000032]$	411.2
	64	0.000033	$[-0.000121, 0.000059]$	432.7
4096	1	0.000023	$[-0.000065, 0.000064]$	126.7
	16	0.000043	$[-0.000122, 0.000039]$	377.6
	32	0.000044	$[-0.000145, 0.000061]$	414.0
	64	0.000045	$[-0.000178, 0.000090]$	433.6
5120	1	0.000031	$[-0.000064, 0.000065]$	126.9
	16	0.000052	$[-0.000135, 0.000033]$	379.1
	32	0.000054	$[-0.000157, 0.000050]$	414.9
	64	0.000056	$[-0.000196, 0.000091]$	433.0

Table VIII

summation does however outperform doubly compensated summation (only 2% slower than not using any compensated summation as opposed to doubly compensated summation which is 7% slower), which is expected as it requires fewer operations than doubly compensated summation[11].

IV. CONCLUSION

In this paper we have described a method of applying doubly compensated summation in a limited manner whereby we give the application programmer a lever which can be used to increase accuracy at a very minor cost in performance. We have also shown that a whole digit of accuracy can be recaptured with only a small performance loss when

using *selective* doubly compensated summation thereby allowing the application programmer to gain most of the benefit of doubly compensated summation without incurring the cost (applying doubly compensated summation every operation is an order of magnitude slower than applying it *selectively*, as seen in tables 5 and 7). We have also shown that if the FMADD instruction on all current generation Nvidia GPUs conformed to the IEEE 754 standard it would be possible for selective doubly compensated summation to generate the gold standard or fairly close to it (as discussed in section 3). We also have reason to believe that future generations of the Nvidia GPU will implement the FMADD operation in a manner consistent with the IEEE 754 standard, further increasing the utility and benefit of selective doubly compensated summation. Finally, we have demonstrated that this technique can also be applied to Kahan's compensated summation with similar effect and better performance.

REFERENCES

- [1] K. Yasuda, "Accelerating density functional calculations with graphics processing unit," *Journal of Chemical Theory and Computation*, vol. 4, no. 8, pp. 1230–1236, August 2008. [Online]. Available: <http://dx.doi.org/10.1021/ct8001046>
- [2] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck, "Cryptographics: Secret key cryptography using graphics cards," in *In RSA Conference, Cryptographers Track (CT-RSA)*, 2005, pp. 334–350.
- [3] W. Qiao, D. S. Ebert, and A. Entezari, "Klimateck g.: Volq: Direct volume rendering of multi-million atom quantum dot simulations," in *In Proceedings of the IEEE Conference on Visualization*, 2005, pp. 319–326.
- [4] Nvidia, *NVIDIA CUDA Programming Guide*, 2nd ed., NVIDIA, July 2009.

- [5] Y. He and C. H. Q. Ding, "Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications," in *ICS '00: Proceedings of the 14th international conference on Supercomputing*. New York, NY, USA: ACM, 2000, pp. 225–234.
- [6] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM, 2002.
- [7] Y. Nievergelt, "Analysis and applications of priest's distillation," *ACM Trans. Math. Softw.*, vol. 30, no. 4, pp. 402–433, 2004.
- [8] Y.-K. Zhu, J.-H. Yong, and G.-Q. Zheng, "A new distillation algorithm for floating-point summation," *SIAM J. Sci. Comput.*, vol. 26, no. 6, pp. 2066–2078, 2005.
- [9] I. J. Anderson, "A distillation algorithm for floating-point summation," *SIAM J. Sci. Comput.*, vol. 20, no. 5, pp. 1797–1806, 1999.
- [10] D. M. Priest, "On properties of floating point arithmetics: Numerical stability and the cost of accurate computations," Ph.D. dissertation, University of California Berkeley, 1992.
- [11] W. Kahan, "Pracniques: further remarks on reducing truncation errors," *Commun. ACM*, vol. 8, no. 1, p. 40, 1965.
- [12] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Dover Publications, Incorporated, 1994.
- [13] N. J. Higham, "The accuracy of floating point summation," *SIAM J. Sci. Comput.*, vol. 14, pp. 783–799, 1993.
- [14] Nvidia, *NVIDIA CUDA C Programming Best Practices Guide*, 2nd ed., NVIDIA, July 2009.
- [15] V. Volkov and J. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proceedings of the ACM/IEEE Conference on High Performance Computing*. IEEE/ACM, November 2008, p. 31.
- [16] B. N. Parlett, *The Symmetric Eigenvalue Problem*, ser. Classics In Applied Mathematics. SIAM, 1998.
- [17] S. Wasson, "Nvidia's 'fermi' gpu architecture revealed," September 2009. [Online]. Available: <http://techreport.com/articles.x/17670>
- [18] J. M. McNamee, "A comparison of methods for accurate summation," *SIGSAM Bull.*, vol. 38, no. 1, pp. 1–7, 2004.