Scientific Programming 19 (2011) 3–11
DOI 10.3233/SPR-2011-0315
IOS Press

Improving accuracy for matrix multiplications on GPUs

Matthew Badin *, Lubomir Bic, Michael Dillencourt and Alexandru Nicolau
Computer Science, UCI, Irvine, CA, USA

Abstract. Reproducibility of an experiment is a commonly used metric to determine its validity. Within scientific computing, this can become difficult due to the accumulation of floating point rounding errors in the numerical computation, greatly reducing the accuracy of the computation. Matrix multiplication is particularly susceptible to these rounding errors which is why there exist so many solutions, ranging from simulating extra precision to compensated summation algorithms. These solutions however all suffer from the same problem, abysmal performance when compared against the performance of the original algorithm. Graphics cards are particularly susceptible due to a lack of double precision on all but the most recent generation graphics cards, therefore increasing the accuracy of the precision that is offered becomes paramount. By using our method of *selectively* applying compensated summation algorithms, we are able to return a whole digit of accuracy on current generation graphics cards and potentially two digits of accuracy on the newly released "fermi" architecture. This is all possible with only a 2% drop in performance.

Keywords: GPGPU, matrix multiplication, floating point, compensated summation

1. Introduction

The graphics card is becoming increasingly attractive to scientific computing, potentially offering super computer power within the confines of a desktop PC. Consequently there is significant work in transitioning traditional scientific computing problems from the CPU to the GPU [2,13,17]. Graphics cards however are not without problems, including a lack of double precision on all but the most recent generation of graphics cards. This means the accumulation of rounding errors becomes particularly pronounced when solved using a graphics card, affecting 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 [3]. Summation in general is particularly susceptible to rounding errors [5]. The solutions offered to improve numerical stability range from simply sorting the input into either descending or ascending order, depending on the sign of the summands, to distillation algorithms which may also first sort the input [1] or require multiple iterations of a main loop [18]. However, unlike the parallel systems traditionally used for scientific computing, branching is particularly expensive on the GPU, greatly diminishing the pool of available solutions. What we instead offer is a method of applying a traditional solution to improving numerical stability by using it in a *selective* manner, thereby greatly reducing the branching cost associated with the solution while still improving accuracy. The practicality of our approach is demonstrated by implementing the solution on a Nvidia graphics card, Nvidia being the most popular GPU manufacturer and therefore the most widely available. Our results will show that when applied to matrix multiply will regain a whole digit of accuracy, a 10 fold increase in accuracy. 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, as it now does on the newly released "fermi" architecture, then we could return two digits of accuracy. Furthermore, when compared against Nvidia BLAS, this is all possible with only a 7% performance loss when doubly compensated summation is applied selectively and only a 2% loss in performance when Kahan's compensated summation algorithm is applied selectively. 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

*Corresponding author: Matthew Badin, Computer Science, UCI, Irvine, CA, USA. E-mail: mbadin@uci.edu.

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 what can be gained by applying selective doubly compensated summation and selective Kahan's compensated summation.

2. 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 do not. Since the focus of this paper is not to propose additional hardware for inclusion in the next generation of GPUs, we will 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 [8] and because they focus primarily on ill-conditioned data (data that has heavy cancellation) [19]. Simulating extra precision in software was not considered because of the poor performance, "that such simulation is too expensive to be of practical use for routine computation" [5]. 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" [1]. Within the family of compensated summation algorithms, we chose doubly compensated summation as it is more accurate than Kahan's compensated summation [12], however, at a cost of increased number of operations. What follows is the algorithm for doubly compensated summation [12].

2.1. Doubly compensated summation

Figure 1 contains the pseudo code for doubly compensated summation, where n is the number of summands and $x_1, \ldots, x_n$ are the summands (in its most general form, assumed to be sorted in decreasing order by absolute value). 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 [6].

ALGORITHM: Doubly compensated summation
1 procedure dcsum($n, x_1, \ldots, x_n$)
2 begin
3 $s_1 = x_1, c_1 = 0$
4 for $k = 2, \ldots, n$
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

Fig. 1. Doubly compensated summation.

2.2. 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.

To put the idea into more concrete terms, the algorithm would work as follows: given a set of summands, k summands 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 [4,16]. 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 [9]. 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.

ALGORITHM: Selective doubly compensated summation

```
1   procedure selective_dcsum(k, n, x_1, ..., x_n)
2   begin
3   partialSum = 0, total = 0
4   for i = 1, ..., n ÷ k
5       for j = (i − 1) × k, ..., max(i × k, n)
6           partialSum = partialSum + x_j
7       end for
8       if(|total| ⩾ |partialSum|)
9           total = dcsum(2, total, partialSum)
10      else
11          total = dcsum(2, partialSum, total)
12      partialSum = 0
13  end for
14  return total
```

Fig. 2. Selective doubly compensated summation.

2.3. 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 [14]. 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 has not 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.

Given the following matrices: $A \times B = C$, the algorithm loads a 16×16 tile of B into shared memory.

The algorithm then loads a 64×16 tile of A (64×1 at a time) and then each thread multiples 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 16×16 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 64×16 tile and applying doubly compensated summation selectively after every 16 multiplications (instead of 64 as the tile size suggests) versus loading a 16×16 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 128×16 tile of A would be loaded instead of 64×16 (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 [9]. 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 5120×5120 matrix. The data was produced on a machine running Kubuntu 9.04 64-bit using an Intel Quad Core 2.66 GHz processor, 8 GB 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 did not 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.

Table 1
Thread count vs tile size

Thread count	16×16 tile	32×16 tile	64×16 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

3. 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.

3.1. 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 [5]. The performance of the GPU algorithms was measured using CUDA Events ($(2 \times N^3) \div \mathit{time}$, 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 [5,11]; (2) doubly compensated summation only affects summation and not multiplication and we did not 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 [10] and consequently is the topic of the next subsection.

3.2. Asymmetric error range

Early on in our testing we noticed an asymmetry in the error range. Isolating the source, it turned out to be

Table 2
Error analysis of doubly compensated summation using FMADD

FMADD truncation, $N = 1024$		
Tile size	Avg error	Max error range
16	0.000011	$[-0.000035, 0.000014]$
64	0.000012	$[-0.000063, 0.000043]$

Table 3
Error analysis of doubly compensated summation without FMADD

Proper rounding, $N = 1024$		
Tile size	Avg error	Max error range
16	0.000006	$[-0.000024, 0.000024]$
64	0.000008	$[-0.000050, 0.000047]$

a design choice by Nvidia in how they implemented their floating point multiply-and-add (FMADD). The way FMADD is implemented on all current generation Nvidia GPUs (not including the "fermi" architecture) is by truncating the intermediate result of the multiplication instead of rounding [10]. Table 2 illustrates the consequences, the matrix size is 1024×1024, 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 round the result instead of truncating (at a performance cost).

By comparing the results of Table 3 against a tile size of 64 in Table 5 for $N = 1024$, it is obvious that if FMADD was implemented using rounding instead of truncation of the intermediate result, our algorithm would no longer produce an asymmetric error and would also yield two digits of accuracy instead of one. Of course the reason we do not 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 [15], so consequently the accuracy of selective doubly compensated summation will only improve, allowing for an additional digit of accuracy. Despite this issue with the implementation of the FMADD operation, our algorithm manages to recover a full digit of accuracy with only a minor performance loss, as we will demonstrate in the next section.

3.3. 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 a max error of 0.000065, Nvidia's BLAS has a max error of 0.008014 for the same problem size (Table 4), and selective doubly compensated summation achieves a max error of 0.000200 for a tile size of 64 (Table 5), or roughly a whole digit of accuracy was recaptured as opposed to using Nvidia's BLAS. It is also obvious from the same tables that a second digit of accuracy is possible if there was not an asymmetry in the error, a 100 fold improvement in accuracy. As can also be seen, selective doubly compensated summation consistently returns a whole digit of accuracy even with the asymmetry. It is worth noting that a tile size of 64 appears to be a good choice

Table 4
Nvidia BLAS GPU matrix multiply on $[0, 1]$

Size	Avg error	Max 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 5
Selective doubly compensated summation on [0, 1]

N	Tile	Avg error	Max 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

as it offers the best performance at comparable accuracy (you do not 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 $[0, 1]$.

Table 6
Nvidia BLAS GPU matrix multiply on $[-1, 1]$

Size	Avg error	Max 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 7
Selective doubly compensated summation on $[-1, 1]$

N	Tile	Avg error	Max 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

3.4. Selective Kahan's compensated summation

For sake of comparison, we also briefly explored selectively applying Kahan's compensated summation [6], the code for which is in Fig. 3 (it is important to note that unlike doubly compensated summation, Kahan's compensated summation does not require the input to be sorted). 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 is not fully sorted, though it is sup-

ALGORITHM: Kahan's compensated summation
1 procedure kahan($n, x_1, \ldots, x_n$)
2 begin
3 $s_0 = 0, c_0 = 0$
4 for $k = 1, \ldots, n$
5 $temp_k = s_{k-1}$
6 $y_k = x_k + c_{k-1}$
7 $s_k = temp_k + y_k$
8 $c_k = (temp_k - s_k) + y_k$
9 return s_n
10 end

Fig. 3. Kahan's compensated summation.

Table 8
Selective Kahan's summation on [0, 1]

N	Tile	Avg error	Max 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

posed to be [12], 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 [7]). Kahan's 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 [6].

Table 9
Selective Kahan's summation on $[-1, 1]$

N	Tile	Avg error	Max 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

4. Numerical analysis

The notation that will be used for the analysis will be that commonly used by other authors such as Higham [5]. To briefly reiterate the notation, u is the unit roundoff, $fl(x + y)$ is the floating point approximation of the value contained in the parenthesis where x and y are two real numbers. The quantities s_n and $\hat{s}_n$ denote the actual sum and the floating point computed sum, respectively. The rest of this section is organized as follows: first we will briefly discuss the affect of not fully sorting the input for selective doubly compensated summation. Then we will discuss the forward error bound of standard matrix multiply followed by how the forward error bound is affected by breaking up the summation of the inner product. Finally, we will discuss how our application of selective doubly compensated summation and selective Kahan's compensated summation affect this variation on the inner product.

4.1. Unsorted compensated summation

The importance of first sorting the summands before being passed to doubly compensated summation is emphasized by Higham in his discussion of summation methods, that it "removes the need for certain

logical tests that would otherwise be necessary" [5]. Specifically, the lines of code affected in doubly compensated summation (in Fig. 1) are 6, 8 and 11. Several assumptions are made about the input and when those assumptions are satisfied the forward error bound is $|s_n - \hat{s}_n| = 2u|s_n|$ where s_n is the actual sum and $\hat{s}_n$ is the computed sum. However, when any one of the assumptions are not met (such as sorting the input) the difference calculations (lines 6, 8 and 11 in Fig. 1) become estimates (though as demonstrated in Section 3, appear to be quite good estimates). Luckily, Kahan's compensated summation has no such restriction on the input, thus the error analysis becomes surprisingly simple, as will be discussed in the next subsection.

4.2. Forward error bound

We use the standard floating point model for estimating error bounds as presented in [5]. In this model, the result of an individual floating point operations is $fl(x + y) = (x + y)(1 + \delta)$, where $|\delta| \leqslant u$. The following forward error bound for the inner product of matrix multiply is derived in [5]:

$$\begin{aligned} |x^{\mathrm{T}}y - fl(x^{\mathrm{T}}y)| &\leqslant \gamma_n \sum_{i=1}^{n} |x_i y_i| \\ &= \gamma_n |x|^{\mathrm{T}}|y|, \end{aligned} \tag{1}$$

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:

$$|s_n - \hat{s}_n| \leqslant \gamma_{n/k+k-1}|x|^{\mathrm{T}}|y|. \tag{2}$$

(Here, s_n is the true value of the inner product and $\hat{s}_n$ is the computed value.) If all additions (i.e., the computation of each strip sum and the adding of the strip sums to compute the inner product) are to be done without compensation, then the best strategy is to minimize the error in (2) by setting $k = \sqrt{n}$. A better bound is achieved by selective application of Kahan's compensated summation (using it for the final sum but not to compute the individual strip sums), as we now show.

Let the rth strip sum be

$$p_r = \sum_{i=r\cdot(k-1)+1}^{r\cdot k} x_i y_i, \tag{3}$$

where r ranges from 1 to n/k. The computed sum $\hat{s}_n$ is the computed value of the sum of the $\hat{p}_r$ values, which are the computed values of the strip sums p_r. Let $\bar{s}_n$ be the true sum of the $\hat{p}_r$ values. By the triangle inequality, we have

$$|s_n - \hat{s}_n| \leqslant |s_n - \bar{s}_n| + |\bar{s}_n - \bar{s}_n|. \tag{4}$$

Applying (1) to the sum in (3) yields the inequality

$$|p_r - \hat{p}_r| \leqslant \gamma_k |p_r|. \tag{5}$$

Summing over r and applying the triangle inequality,

$$|s_n - \bar{s}_n| \leqslant \gamma_p |x|^{\mathrm{T}}|y|. \tag{6}$$

As shown in [5], Kahan's compensated summation has the following error bound when used to compute an entire vector sum $s_n = v_1 + \cdots + v_n$:

$$|s_n - \hat{s}_n| \leqslant (2u + \mathrm{O}(nu^2)) \sum_{i=1}^{n} |v_i|. \tag{7}$$

When we add up the n/k computed strip sums $\hat{p}_r$ using Kahan's compensated summation and apply (5), the error bound in (7) yields

$$\begin{aligned} &|\bar{s}_n - \hat{s}_n| \\ &\quad\leqslant \left(2u + \mathrm{O}\left(\frac{nu^2}{k}\right)\right)(1 + \gamma_k)|x|^{\mathrm{T}}|y| \\ &\quad= \left(2u + \frac{2ku^2}{(1 - ku)} + \mathrm{O}\left(\frac{nu^2}{k}\right)\right)|x|^{\mathrm{T}}|y|. \end{aligned}$$

Combining (4), (6), and the last estimate gives us the error bound:

$$\begin{aligned} |s_n - \hat{s}_n| \leqslant \Bigg(&2u + \gamma_k + \frac{2ku^2}{(1 - ku)} \\ &+ \mathrm{O}\left(\frac{nu^2}{k}\right)\Bigg)|x|^{\mathrm{T}}|y|. \end{aligned}$$

As can be seen by the bound (and Tables 8 and 9), the error is largely determined by k, which is fixed at compile time.

5. Conclusion

In this paper we have described a method of *selectively* applying both doubly compensated summation and Kahan's compensated summation in a lim-

ited manner whereby we give the application programmer a lever which can be used to increase accuracy at a very minor cost in performance (only a 7% performance loss when using selective doubly compensated summation and only a 2% performance loss when using selective Kahan's compensated summation). We have also demonstrated empirically that very few applications of either compensated summation algorithm are required to capture most of the benefit of the algorithm, allowing the application programmer to gain a whole digit of accuracy without incurring the cost (applying either compensated summation algorithm 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 (non-Fermi GPUs) conformed to the IEEE 754 standard it would be possible for selective doubly compensated summation to regain an additional digit of accuracy, a 100 fold improvement over Nvidia BLAS (as discussed in Section 3). We 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 compensated summation. We plan to demonstrate this in future works on the Fermi architecture.

References

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

[2] D.L. Cook, J. Ioannidis, A.D. Keromytis and J. Luck, Cryptographics: Secret key cryptography using graphics cards, in: *RSA Conference, Cryptographer's Track (CT-RSA)*, 2005, pp. 334–350.

[3] 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*, ACM, New York, NY, USA, 2000, pp. 225–234.

[4] N.J. Higham, The accuracy of floating point summation, *SIAM J. Sci. Comput* **14** (1993), 783–799.

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

[6] W. Kahan, Pracniques: further remarks on reducing truncation errors, *Commun. ACM* **8**(1) (1965), 40.

[7] J.M. McNamee, A comparison of methods for accurate summation, *SIGSAM Bull.* **38**(1) (2004), 1–7.

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

[9] Nvidia, *NVIDIA CUDA C Programming Best Practices Guide*, 2nd edn, NVIDIA, 2009.

[10] Nvidia, *NVIDIA CUDA Programming Guide*, 2nd edn, NVIDIA, 2009.

[11] B.N. Parlett, *The Symmetric Eigenvalue Problem*, Classics in Applied Mathematics, SIAM, 1998.

[12] D.M. Priest, On properties of floating point arithmetics: Numerical stability and the cost of accurate computations, PhD dissertation, University of California, Berkeley, 1992.

[13] W. Qiao, D.S. Ebert and A. Entezari, Klimeck g.: Volqd: Direct volume rendering of multi-million atom quantum dot simulations, in: *Proceedings of the IEEE Conference on Visualization*, 2005, pp. 319–326.

[14] V. Volkov and J. Demmel, Benchmarking GPUs to tune dense linear algebra, in: *Proceedings of the ACM/IEEE Conference on High Performance Computing*, November 2008, IEEE/ACM, p. 31.

[15] S. Wasson, Nvidia's 'fermi' GPU architecture revealed, 2009, available at: http://techreport.com/articles.x/17670.

[16] J.H. Wilkinson, *Rounding Errors in Algebraic Processes*, Dover, 1994.

[17] K. Yasuda, Accelerating density functional calculations with graphics processing unit, *Journal of Chemical Theory and Computation* **4**(8) (2008), 1230–1236, available at: http://dx.doi.org/10.1021/ct8001046.

[18] Y.-K. Zhu and W.B. Hayes, Algorithm 908: Online exact summation of floating-point streams, *ACM Trans. Math. Software* **37**(3), 2010, Article No. 37.

[19] Y.-K. Zhu, J.-H. Yong and G.-Q. Zheng, A new distillation algorithm for floating-point summation, *SIAM J. Sci. Comput.* **26**(6) (2005), 2066–2078.