Parallel Exact Summation Algorithms on the GPU

Ryan Killick

December 16, 2016

Abstract

The problem of exactly summing a list of \( n \) floating point numbers has received much attention in recent years due to the increasing impact of rounding errors on today’s large-datasets. One area of particular concern is on the reproducibility of floating point sums - a problem arising from the non-associativity of floating point addition. This problem becomes more apparent as industry is switching to more parallel computing schemes in which the order in which floating point inputs are visited is undefined. In this paper we outline a number of exact-summation algorithms specifically for implementation on a GPU (graphics processing unit) architecture. Inspired by similar work, we employ a specific number representation in order to remove parallelization-destroying carry-digit propagations, and thus develop highly parallelizable algorithms that outperform the best sequential exact-summation algorithm for all dataset conditions.

1 Introduction

Floating-point number systems are ubiquitous in digital computing due to their ability to represent a much wider range of numbers than, for example, a fixed-point (integer) representation with the same word size. This superior range comes at the cost of a non-exact and non-associative standard arithmetic due to the rounding error that must be incurred as a result of the fixed word-size.

Because of this rounding error, it turns out to be a somewhat difficult task if one actually wants to calculate an \textit{exact} sum of a list of floating point numbers. In the sequential-computing world this problem has been studied extensively and many efficient algorithms have been created. However, due to the increasing size of datasets that are dealt with today as well as the physical limitations on clock-speeds of modern processors, it is desirable to develop efficient \textit{parallel} algorithms for this purpose as well. There has been much less work completed on this task, most of which was produced in the last five years.

If we further restrict ourselves to parallel exact summation algorithms implemented on the GPU, then previous work becomes almost non-existent. This makes this a fruitful topic of research as well as an important one due to the increasing use of GPUs in high-performance scientific computation (the area where exact-summation algorithms would be most useful). It is the purpose of this paper to develop and analyze a variety of parallel exact-summation algorithms specifically for implementation on the GPU.

The organization of the paper is as follows: we first present a brief introduction to floating point number systems and formally describe the problem of exactly summing a list of floating point numbers. We then discuss previous work completed on this problem and identify parallel algorithms that may be suitable for implementation on the GPU (with or without modification). This is followed by a description of the algorithms that we have
implemented and some of the details of these implementations. Finally, we present and analyze the results of running-time tests and conclude with a discussion of these results.

2 Floating point numbers

Throughout this paper we will take the view that a floating point (f.p. from now on) number $x$ is represented as a base-2 number using a sign bit $s$, a mantissa $m$, and an exponent $e$ in the same style as outlined in the IEEE-754 standard. Specifically, if we fix the word size of the f.p. representation to be $B$ bits and allocate 1 bit for the sign, $p$ bits for the mantissa, and $q$ bits for the exponent then the normalized f.p. numbers ($e \in [1, 2^q - 2]$) have the following representation,

$$x = (-1)^s \times (1 + m \times 2^{-p}) \times 2^{e - 2^q - 1 + 1}$$

(1)

and the subnormal numbers ($e = 0$) are represented as,

$$x = (-1)^s \times (m \times 2^{-p}) \times 2^{2^q - 1}$$

(2)

We will in general consider both the normal and subnormal numbers in the algorithms presented.

We represent the set of possible f.p. numbers for a given $B$, $p$, and $q$ as $\mathbb{F}_{B,p,q}$.

A number of useful properties can be derived from the above representation, however, the most useful ones are the maximum and minimum (non-zero) numbers representable,

$$x_{\text{max}} = (2 - 2^{-p})2^{2^q - 1 - 1}$$

(3)

$$x_{\text{min}} = 2^{2^q - 1 - p}$$

(4)

We can define the standard f.p. addition of two f.p. numbers $x$ and $y$ using [4],

$$x \oplus y = \text{round}(x + y) = (x + y)(1 + \delta_{x,y}), \quad |\delta_{x,y}| \leq \epsilon$$

(5)

where $\text{round}()$ is some rounding function that rounds the sum $x + y$ into our f.p. system, $\delta_{x,y}$ is a summand specific rounding error, and $\epsilon$ is the machine precision$^3$.

As written, the standard f.p. addition operator is (in general) non-exact,

$$x \oplus y \neq x + y$$

and non-associative,

$$x \oplus (y \oplus z) \neq (x \oplus y) \oplus z$$

Thus, if we are presented with a list of f.p. numbers $X = \{x_0, \cdots, x_{n-1}\}$ and we simply compute the naive sum,

$$S_{\text{naive}} = \oplus_{i=0}^{n-1} x_i$$

(6)

$^1$A normalized representation is one in which each number is uniquely represented by the triplet of integers $(s, m, e)$.

$^2$An exponent value of $e = 2^q - 1$ is reserved to represent infinity (NaN) if $m = 0$ ($m \neq 0$).

$^3$In the case of round-to-nearest-even type of rounding we have $\epsilon_{\text{mach}} = 2^{-p+1}$. 

2
then we will find that we have an error bounded by \[7\],

\[
R_{\text{naive}} = \frac{|S_{\text{naive}} - S|}{|S|} \leq \frac{\epsilon n}{1 - \epsilon n} C(X) \tag{7}
\]

where \(S\) is the sum computed using the exact addition operator,

\[
S = \sum_{i=0}^{n-1} x_i \tag{8}
\]

and \(C(X)\) is called the condition number of the data,

\[
C(X) = \frac{\sum_{i=0}^{n-1} |x_i|}{\sum_{i=0}^{n-1} x_i} \tag{9}
\]

which quantifies the difficulty of the summation problem.

The error bound in Equation 7 grows linearly with the number of summands, and, even for modest condition numbers, can produce a result in which not a single significant digit matches the actual sum \(S\).

We can get a somewhat better error bound if we perform a pairwise summation of \(X\),

\[
R_{\text{pair}} = \frac{|S_{\text{pair}} - S|}{|S|} \leq \frac{\epsilon \log n}{1 - \epsilon \log n} C(X) \tag{10}
\]

however, even with the much better logarithmic error growth, we can still end up with a result that does not share any significant digits with the sum \(S\) in the case that the data \(X\) has a large condition number.

In general, even if we can compute the sum \(S\), the result may not be representable in our f.p. system and thus there will always be the possibility of rounding error in our final result. What we want for an exact sum, then, is to minimize this rounding error as much as possible. As the largest rounding error we can incur from rounding a single real number into our f.p. system is the machine precision, this is what we will want to achieve for an error bound on our exact sum. Thus, we will define the problem of exactly summing a list of \(n\) f.p. numbers \(X\) as follows,

**Problem 1** Exact-Summation

*Given a set of floating-point numbers \(X = \{x_0, \ldots, x_{n-1}\} \in \mathbb{F}_{B,p,q}\) with exact sum \(S\), find \(S^* \in \mathbb{F}_{B,p,q}\) such that,*

\[
\frac{|S^* - S|}{|S|} \leq \epsilon
\]

*In other words, we want to find the f.p. number \(S^* \in \mathbb{F}_{B,p,q}\) such that,*

\[
S^* = \text{round}(S)
\]
3 Parallel exact summation algorithms

A review of the relevant literature shows that all of the modern exact summation algorithms can be divided into three sub-classes:

- Distillation algorithms: Algorithms that use some form of an "AddTwo" function in order to distill out the exact sum from a set of numbers.
- Superaccumulator algorithms: Algorithms that use integer arithmetic on various data-structures to carry out exact intermediate sums.
- Hybrid algorithms: A combination of the two above methods.

We will describe each sub-class separately.

3.1 Distillation algorithms

All of the distillation algorithms rely on the existence of an AddTwo() function which takes in two f.p. inputs \( x \) and \( y \) and outputs the rounded sum \( s \) and rounding error or residual \( r \) in the same f.p. representation as the inputs,

\[
\text{AddTwo}(x, y) \rightarrow (s, r), \quad x + y = s + r
\]

The canonical implementation of AddTwo is given by Knuth [9] and Møller [14] and is reproduced here in Algorithm 1,

Algorithm 1 AddTwo (Knuth and Møller)

Let \( x \) and \( y \) be two \( B \)-bit \( (B \geq 3) \) f.p. numbers. Then the following algorithm outputs the non-overlapping f.p. numbers \( s \) and \( r \) such that \( x + y = s + r \),

1. \( s \leftarrow x \oplus y \)
2. \( t_{y1} \leftarrow s \odot x \)
3. \( t_{x1} \leftarrow s \odot t_{y1} \)
4. \( t_{y2} \leftarrow y \odot t_{y1} \)
5. \( t_{x2} \leftarrow x \odot t_{x1} \)
6. \( r \leftarrow t_{x2} \oplus t_{y2} \)
7. return \((s, r)\)

In this class of algorithms is the state-of-the-art sequential Online Exact Sum [16] which boasts an asymptotic time (step) complexity of \( \mathcal{O}(5n) \) (independent of the condition number) with the possibility of instruction level parallelism improving this to \( \mathcal{O}(3n) \). This is the algorithm one should compare an efficient parallel implementation to.

The only modern parallel algorithm that is solely a distillation type was introduced by Kadric et. al. in [8] (which was inspired by the work of Leuprecht and Oberaigner in [10]). The basic idea is an iterative procedure in which a modified parallel reduction is performed.
on the input f.p. numbers using the AddTwo function as an operator. In each step of the reduction the sum is accumulated and the residuals are written back into the original input array. This procedure is then iterated with the sum of the residuals from each new iteration being added to the total sum until some convergence criteria is satisfied (see Figure 1). The number of iterations necessary for convergence can be determined from,

$$k = \frac{\log C(X)}{p + 1}$$

where $C(X)$ is the condition number of the data being summed, and $p$ is the number of bits reserved for the floating point mantissa (see Section 2). For a f.p. representation with $q$ bits reserved for the exponent (and allowing subnormals), the maximum number of iterations is given by,

$$k_{\text{max}} = \frac{2^q + p}{p + 1}$$

For IEEE-754 double precision, in the worst case, we need 40 iterations in order to ensure that the sum has converged to the exact one. However, the authors observed in [8] that in most practical cases convergence is achieved in only two iterations. Unfortunately, implementing the convergence criteria outlined in [8] is somewhat tricky on the GPU and thus a full version of this algorithm was not actually implemented in this work. For this reason, because this algorithm is not the main focus of this work, and because it was observed that this algorithm is slower than the ones developed in this work, we will not discuss it further.

### 3.2 Superaccumulator algorithms

These algorithms all rely on the fact that integer arithmetic is exact and associative. They all generally consist of three stages. In the first stage (expansion stage) each input f.p. number is expanded into a large fixed-point representation called a superaccumulator. In the second stage (reduction stage) a parallel reduction is performed on each of the superaccumulators produced in the first stage resulting in a single superaccumulator representing the exact sum of the inputs. In the last stage (normalization) the final superaccumulator is normalized (if necessary) and rounded into the wanted f.p. representation.

The main drawback of superaccumulator algorithms is that they suffer from carry-bit propagations. These carry-bits tend to destroy parallelization due to the extra communication and/or branching instructions needed to handle them. In order to avoid these

---

4This, and Equation 11, are my equations (at least I have not seen them given anywhere else, yet), which I state without proof. They can be understood, however, by the fact that in each iteration we produce a sum and a non-overlapping residual (i.e., it’s most-significant-digit is at least $p + 1$ less than the most-significant digit of the sum). The condition number essentially tells us how many digits we need to look at in order to find the exact sum, and, if we are decreasing $(p + 1)$ digits per iteration then Equation 11 follows. For an f.p. representation with a fixed number of exponent bits (allowing subnormals) the maximum number of digits we need to look at (and thus an effective maximum on $C(X)$) is $2^q + p$. These iteration bounds have been extensively tested with various datasets, with known condition numbers spanning the entire domain, and are observed to be correct.

5Which stands to reason that the authors consider a practical case to be any data with $C(X) < 2(p+1) = 106$ (for double precision).

6and for lack of time
problems, carry-save superaccumulator representations are used. Essentially, these are redundant integer representations which reduce the frequency of carry-bit operations at the expense of a larger amount of memory-usage.

Neal’s small and large superaccumulator algorithms [12] are typical examples of this class of algorithm in which a carry-save representation is used. As the algorithms described in [12] are sequential by construction they are not entirely portable to the GPU, however, a very similar idea is employed in this work to produce a highly parallel “small” superaccumulator algorithm on the GPU. This algorithm is described further in Section 4.

A unique approach that eliminates carry-bit propagations completely was presented by Goodrich and Eldawy in [5] and is more so an application of an algorithm developed by Parhami in his generalized-signed-digit (GSD) integer representations [15]. Due to the elimination of carry-bit propagations this algorithm has the potential for massive parallelization and a modified version has thus been developed in this work for implementation on the GPU. It will be discussed further in Section 4.

### 3.3 Hybrid algorithms

In this class we have algorithms that use some aspect of both the distillation and SA types.

A parallel algorithm that uses a variant of the AddTwo function that is worth mentioning is provided by Demmel and Nguyen in [3]. In this work the authors do not specifically pursue the problem outlined in this paper and instead develop an algorithm that obtains a much higher accuracy than a pairwise-summation and still limit themselves to a maximum step-complexity equivalent of a single parallel reduction (which was one of the goals). They also mention that the accuracy of their algorithm can be increased (to exactness) by using more passes over the data, essentially turning into a variant of Kadric’s distillation algorithm [8]. It is an interesting algorithm, but, as it does not tackle the exact summation problem directly we do not consider it further.
A state-of-the-art hybrid exact-summation algorithm was proposed and implemented by Collange et. al. in [2]. The algorithm uses a multi-stage and hierarchal approach to memory\(^7\) in order to produce extremely fast runtimes (though, only for small input exponent ranges, \(< 2^{150}\)). In the first stage of their algorithm they use an AddTwo function to locally (per-thread) distill a batch of the f.p. inputs using a fixed number of iterations. If it is found that this number of iterations is not sufficient (or the inputs have been exhausted) than the remaining residual (or partial sum) is accumulated into a private per-thread (or per-group of threads) superaccumulator stored in shared memory. These shared-memory superaccumulators are then merged into a single superaccumulator per thread-group. Finally, each remaining superaccumulator in shared memory is merged into a single global superaccumulator that is converted to a f.p. number and rounded. Unfortunately, we did not have time to test this algorithm explicitly and will have to rely on the results given in [2] for comparison.

4 Our Algorithms

In this section we describe the algorithms that we have developed for the GPU. We have focused on two algorithms: the first is based on the ideas presented in [12] and which we will simply call SA-Summation. The second is based on the ideas presented in [5] and [15] and thus we will call it GSD-Summation. In addition, for each algorithm, we present two versions (called the local and global versions, for reasons which will become apparent) that have different runtime-dependencies. We start by describing the SA-summation algorithms.

4.1 SA-Summation

As was previously stated, these algorithms are loosely based off of the work presented in [12]. The idea is a typical superaccumulator algorithm in which each f.p. input is first converted into equivalent “super-digits” (to differentiate from binary digits) of the superaccumulator representation being used. Each super-digit is then added to a globally or locally maintained superaccumulator representing the partial sum of all of the numbers processed. As we have used the same superaccumulator representation for both the global and local versions of this algorithm we start by describing this superaccumulator in more detail.

4.1.1 SA-Summation superaccumulators

We will assume the following representation for our superaccumulator \(Y\),

\[
Y = \sum_{i=0}^{L-1} y_i 2^{ir}
\]

where the parameter \(L\) is the length or number of super-digits \(y_i\) of the superaccumulator and it is assumed that the representation has a power-of-two radix with power \(r\). This representation allows us to cover a f.p. exponent range of \([0, 2^{rL}]\).

Now, as done in [12], in order to prevent carry-bit propagations we employ a carry-save scheme that allows each super-digit to be greater in absolute value than the radix of the representation, up to a maximum of \(2^r - 1\),

\(^7\) which inspired the local versions of the algorithms developed in this paper.
\[ y_i \in [-2^t + 1, 2^t - 1], \quad t > r \]

This is only a carry-saving scheme and thus necessitates carry-bit propagations after a certain number of summands have been processed. The specific number of summands we can safely sum without carry-propagations is going to be limited by the size of the input f.p. numbers and how they are expanded into super-digits. Thus, we need to look at this expansion step in detail.

The first step in the expansion is to simply extract the mantissa, sign, and exponent of each f.p. input. We then use the exponent to determine which super-digits our number corresponds to, and then add these super-digits to our global superaccumulator. As the number of binary digits of a f.p. mantissa is fixed to be \( \leq p + 1 \) the maximum super-digit possible is \( 2^{p+1} - 1 \). Thus, the maximum number of summands we can safely process before a propagation step is,

\[ n < 2^{t-p-1} \quad (14) \]

For double precision inputs with \( t = 63 \) we can only sum 1024 numbers before requiring a carry propagation step.

This is what was done in [12] where the author was more concerned about total operations instead of total steps (as this was a sequential algorithm). For a GPU algorithm we are more concerned with total steps rather than total operations and thus we add a few more operations to the expansion stage that will ensure carry-propagations do not occur for a much more ‘reasonable’ number of summands.

Specifically, we will split each input f.p. mantissa across a fixed number of superaccumulator digits ensuring that each newly added super-digit is less in absolute value than \( 2^r \). This then allows us to safely sum \( 2^{t-r} \) summands before require a carry-propagation step. For double-precision we typically take \( r = 34 \) and \( t = 63 \) allowing us to safely sum \( 2^{29} \approx 500 \times 10^6 \) numbers before requiring a carry-propagation step. Since even the best GPUs on the market have trouble storing \( 2^{29} \) 64 bit numbers in global memory this practically means our superaccumulator representation will be carry-propagation free.

To summarize, we will use the following superaccumulator representation \( Y \) in our SA-Summation algorithms,

\[ Y = \sum_{i=0}^{L-1} y_i 2^{ir}, \quad 0 \leq |y_i| \leq 2^t - 1 \]

with typical values for double-precision inputs,

\[ L = 64, \quad r = 34, \quad t = 63 \]

4.2 Global SA-Summation

We now present the global version of our SA-Summation algorithm. We first initialize a global superaccumulator to zero. Then, for each component of our superaccumulator we perform a parallel reduction on the respective components of our f.p. inputs and write the result of each reduction to our global superaccumulator. Once each component has been reduced, we convert the final global superaccumulator into a f.p. number, rounding appropriately. A schematic of the algorithm is shown in Figure 2 and the
pseudocode is given in Algorithm 2 where the function getSuperDigit($x_i, k$) returns the $k^{th}$ super-digit of $x_i$, parallelReduce($Y_k$) computes the sum of the $k^{th}$ super-digits, and convertToFloatAndRound($Y$) converts the global superaccumulator $Y$ into a f.p. number, rounding appropriately.

The particularly nice thing about this algorithm is the ease at which each super-digit reduction can be further parallelized. Since each super-digit sum is completely independent from each other sum (provided $n < 2^{t-p-1}$) then we can simply launch each super-digit sum in parallel.

Algorithm 2  Global SA-Summation

Input: $n < 2^{2^9}$ double-precision f.p. numbers $x_i \in X$
Output: double-precision f.p. number $S$ representing exact sum of $X$
Assume one thread per data element.

1. Initialize $Y$ of length $L$ to zero, and $Y_k$ of length $n$ to be zero. $Y[k]$ will contain sum of input components in $Y_k$.
2. For $k = 0$ to $L$
   - for all $i \in [0, n-1]$ in parallel
     - $Y_k[i] \leftarrow$ getSuperDigit($x_i, k$)
     - $Y[k] \leftarrow$ parallelReduce($Y_k$)
3. $S \leftarrow$ convertToFloatAndRound($Y$)

As we only need a fixed number of operations to determine any given super-digit, the asymptotic time-complexity of this algorithm should be $O(L \times \text{reduce}(n))$, where $\text{reduce}(n)$ is the time complexity of a single parallel reduction of $n$ inputs. If we have a terribly conditioned dataset then this is not that bad, however, for many practical datasets with small values of $C(X)$ we do not need to actually compute every super-digit of the final global superaccumulator. Thus, in order to reduce this time complexity we employ an initial step that determines which super-digits will actually contribute to the sum, and then only these super-digits are actually computed. The details of determining which super-digits contribute are postponed until Appendix A and only the results for the time complexity are given here (Equation 16),

$$T = \begin{cases} 
O \left[ \left( \frac{2(p+1) + \log n - \log \log n}{r} + 2 \right) \text{reduce}(n) \right], & \text{if } \log C(X) < p + 1 - \log \log n \\
O \left[ \left( \frac{\lceil \log \bar{S} \rceil}{r} + 2 \right) \text{reduce}(n) \right], & \text{otherwise}
\end{cases}$$

where, in the above, $\bar{S}$ is the sum of absolute values of the data elements.

4.2.1 Implementation details

The implementation of the global SA-summation algorithm was actually quite simple as it is essentially just an application of a parallel reduction, with minor modifications (adding the
Figure 2: Diagram of the global SA-summation for 8 input floats $x_i$ (represented as squares). The same input data is used to calculate each of the $L$ super-digits ($d^k_i$, squares to circles), with the total $S_k$ from each super-digit reduction forming the $k^{th}$ super-digit of the global superaccumulator $Y$.

$$Y = S_0 \times 2^0 + S_1 \times 2^r + \cdots + S_{L-2} \times 2^{(L-2)r} + S_{L-1} \times 2^{(L-1)r}$$
getSuperDigit() step). The parallel reduction implementation used in this work was taken almost verbatim from the implementation given in CUDApp (CUDA parallel primitives) a description of which is given in [6].

Computing the super-digit bounds is also just an application of a parallel reduction with a minimal amount of sequential code run on the host CPU needed.

Finally, computing the rounded f.p. result from the final global superaccumulator will take $O(L)$ steps, and, since $L$ is small, this is also computed sequentially on the host CPU.

### 4.3 Local SA-summation

The local version of this algorithm, as the name suggests, uses local per-thread superaccumulators to do most of the work, with writes to shared and global memory occurring only when the data stream is exhausted. The steps are essentially identical to the global version except everything is now moved to a per-thread basis.

Each thread receives a stream of f.p. inputs, converting each one into a fixed number of super-digits. These super digits are then added to the thread’s own local superaccumulator. When the data streams are exhausted each of the per-thread superaccumulators are accumulated into a per-thread-block superaccumulator residing in shared memory. Each of these per-thread superaccumulators are then written to a global superaccumulator representing the exact sum of the data. This global superaccumulator is then converted and rounded into the required f.p. representation. For simplicity we will assume that the number of threads per block is the same as the number of superaccumulator digits. The pseudocode for the algorithm is shown below in Algorithm 3.

In Algorithm 3 the function $\text{getSuperDigits}(x_i)$ returns a fixed number (dependent on $r$, the exponent of our power-of-two radix) of super-digits representing the entire f.p. number $x_i$, as well as the location of those components in $Y_k$. The function $\text{addSuperDigitList}(P,Q)$ adds the super-digits $P$ to the locations $Q$ in $Y_k$. The function $\text{parallelReduceSA}(Y_G)$ reduces the $B$ superaccumulators in $Y_G$ to a single superaccumulator, and, finally, the function $\text{convertToFloatAndRound}(Y_G)$ converts the final superaccumulator to a f.p. number, rounding appropriately.

For this version of the SA-summation algorithm we did not try to implement any extra steps in order to restrict the number of super-digits under consideration\(^8\), and, as a result, we do not explicitly describe a time complexity for this algorithm. However, as we do not adapt the algorithm to the data under consideration we do not expect that it will depend on the condition number of the data. We do find in practice that the algorithm depends strongly on the exponent range of the input data. As this is a GPU effect it is hard to analyze rigorously. We therefore postpone a discussion until the implementation details.

#### 4.3.1 Implementation details

In order to implement this algorithm effectively we need to pay close attention to the register usage in the GPU since this will be what limits our parallelism (by limiting the number of concurrent blocks, and thus thread occupancy). However, going against this is the fact that using registers to hold local super-digits tends to speed up the algorithm due to fewer switches of super-digits in and out of registers and cached memory\(^9\). Thus a balance must be

---

\(^8\)In future work this is certainly a possibility.

\(^9\)This is what causes the dependency on the range of the input dataset. If the exponent range of the input data is large there will be lots of switching between registers and cached memory (potentially every
Algorithm 3 Local SA-Summation

Input: $n < 2^{29}$ double-precision f.p. numbers $x_i \in X$

Output: double-precision f.p. number $S$ representing exact sum of $X$

Assume $B$ blocks of $L$ threads.

1. Initialize array $Y_G$ of length $BL$ to be zero. This will hold all per-block superaccumulators.

2. For all $b \in [0, B - 1]$ do in parallel:
   - For all $k \in [0, L - 1]$ do in parallel:
     - Initialize local superaccumulator $Y_k$ of length $L$ to be zero.
     - For $j$ in $0$ to $n/(BL)$
       - $P, Q \leftarrow \text{getSuperDigits}(x_i)$, super-digits in $P$, locations in $Q$
       - $Y_k \leftarrow \text{addSuperDigitList}(Y_k, P, Q)$
     - Initialize per-block superaccumulator $Y_b$ of length $L$ in shared memory to zero.
   - For all $k \in [0, L - 1]$ do in parallel:
     - For $j = 0$ to $L - 1$:
       - $u \leftarrow (k + j) \mod L$
       - $Y_b[u] \leftarrow Y_b[u] + Y_k[u]$
   - $Y_G[b \cdot L : b \cdot (L + 1) - 1] \leftarrow Y_b[:],$ write per-block superaccumulators to global memory.

3. $Y_G \leftarrow \text{parallelReduceSA}(Y_G)$, in place reduction of all per-block superaccumulators

4. $S \leftarrow \text{convertToFloatAndRound}(Y_G)$, only first $L$ elements of $Y_G$
found between register usage and thread occupancy. This balance will be GPU dependent and thus we must tune our implementation to get the best results.

We can take a rather crude approach in order to speed things up by performing an initial sort-by-exponent on the dataset. This effectively reduces register pressure since the GPU will not switch back and forth between previously visited super-digits. Using this option has a trade off as the time to perform the sort-by-exponent can take longer than performing the actual sum for datasets with small exponent ranges. The sorting algorithm used in this work is taken from the Thrust [1] CUDA template library. Specifically, a radix-sort is employed as we will be sorting exponents for which the number of bits will be small. Details of the radix sort algorithm are given in [11].

The rest of the implementation of the algorithm is mostly straightforward as much of the code from the global version carries over to this version. It should be noted that, as a simplifying step, we perform the final reduction of the per-thread-block superaccumulators on the host CPU. The runtime of this sequential step is negligible in practice since, due to the register pressures above, the number of thread blocks will always be small.

5 GSD-Summation

In this section we describe the GSD-summation algorithms which are based off of the work done in [5] and [15]. There are actually only minor differences between these algorithms and the SA-summation algorithms and thus much of what was outlined in the previous section carries over to this section. Specifically, we will employ the same superaccumulator representation as given in Equation 13,

\[ Y = L - 1 \sum_{i=0}^{L-1} y_i 2^{ir} \]

where again \( L \) is the number of super-digits \( y_i \), and we use a power-of-two radix with power \( r \). Now, as we intend to use the carry-free addition algorithm developed in [15] we need to use a specific superaccumulator representation called an \((\alpha, \beta)\)-normalized superaccumulator. Effectively, this means that we restrict the super-digits \( y_i \) of \( Y \) to be,

\[ y_i \in [-\alpha, \beta], \quad \alpha = \beta = (2^r - 1) > 2 \]

With this definition of our superaccumulator we can now introduce the GSD carry-free addition algorithm.

Technically, the above algorithm is not carry-free since there will always be one carry-digit per summation. What is important is that there will only ever be one carry-digit, and this will always be one of \([-1, 0, 1]\). This algorithm is therefore free of long carry-propagation chains that may occur in other superaccumulator representations. This allows us to always compute the sum of two \((\alpha, \beta)\)-normalized superaccumulators in constant time, in parallel.

As we will only deal with \((\alpha, \beta)\)-normalized superaccumulators in this section we will refer to \((\alpha, \beta)\)-normalized superaccumulators as simply superaccumulators for the duration of the section.

(input) and thus the performance of the algorithm suffers.
Algorithm 4 GSD Carry Free Addition
Input: $Y_a$ and $Y_b$, two $(\alpha, \beta)$-regularized superaccumulators with $\alpha = \beta = R = 2^r - 1$.
Output: $Y$, an $(\alpha, \beta)$-regularized superaccumulator with $\alpha = \beta = R = 2^r - 1$ representing the sum of $Y_a$ and $Y_b$.
Assume $L$ threads.

1. Initialize array $Y$ of length $L$ to be zero.
2. Initialize array $C$ of length $L+1$ to be zero. Will hold carry digits. All $C[i] \in [-1, 0, 1]$.
3. For all $i \in [0, L - 1]$ do in parallel:
   - Compute each component wise sum of the super-digits, $P \leftarrow Y_a[i] + Y_b[i]$.
   - Compute a carry digit, $C[i+1] \leftarrow \lfloor P/R \rfloor$.
   - Compute the super-digit sum $Y[i] \leftarrow P - C[i+1]R + C[i]$.
4. return $Y$.

5.1 Global GSD-Summation
The global version of the GSD algorithm has three main steps: an expansion stage where each input f.p. number is expanded into a full superaccumulator, and a reduction stage where all of the $(\alpha, \beta)$-normalized superaccumulators are reduced to a single global superaccumulator representing the total sum of the data. Finally, this global superaccumulator is converted into a f.p. number, rounding appropriately.

In detail, for the expansion stage, we use $L$ threads in a block with each thread seeing the same f.p. input to calculate the super-digit corresponding to the position or ID of the thread in the block. The entire block then writes back a single superaccumulator to memory.

For the reduction stage we apply Algorithm 4 to reduce the $n$ superaccumulators produced in the expansion stage into a single superaccumulator representing the sum. Each thread in a block again deals with only the super-digit corresponding to its thread ID.

The pseudocode for the global summation algorithm is given below in Algorithm 5, and a schematic of how the algorithm might look on the GPU is shown in Figure 3.

In Algorithm 5 the function $\text{expand}(x_i)$ expands the f.p. input $x_i$ into a full superaccumulator of length $L$. The function $\text{parallelReduce}(Y_G)$ performs a parallel reduction of the superaccumulators in $Y_G$ using Algorithm 4 to add two superaccumulators. Finally, $\text{convertToFloatAndRound}(Y_G)$ converts the final global superaccumulator located in the first $L$ positions of $Y_G$ to a f.p. number, rounding appropriately.

The time complexity of this algorithm will again be $O(L \cdot \text{reduce}(n))$. However, we can use the same trick as done for the SA-summation algorithm to fix $L$ to be only as big as we need. The details of computing this bounded $L$ are given in Appendix A, and, it turns out that the time complexity will be the exact same as that for the global SA-summation algorithm,
Figure 3: Diagram of how the global GSD-summation is implemented on the GPU. In this example there are four input numbers, and each superaccumulator is assumed to have a length of 4 and a radix of $2^3 = 8$. At the top of the figure is the expansion stage, with each thread (represented as circles), computing the super-digit of its input corresponding to its position in the thread block. The next stage is the reduction stage where each of the superaccumulators are reduced to a single superaccumulator using Algorithm 4 with warp-shuffles (see the implementation details) employed to pass the carry-digits to an adjacent thread. Finally, the remaining superaccumulator is converted to a f.p. number and rounded.
Algorithm 5 Global GSD Summation

Input: $n$ f.p. numbers $x_i \in X$
Output: $S$, f.p. number representing exact sum of $X$

Assume $B = n$ blocks of $L$ threads.

1. Initialize array $Y_G$ of length $nL$ to be zero. This will hold the list of the superaccumulators.

2. For all $b \in [0, B - 1]$ do in parallel:
   - For all $i \in [0, L - 1]$ do in parallel:
     - $Y_G[b \cdot L + i] \leftarrow \text{getSuperDigit}(x_b)$

3. parallelReduceSA($Y_G$), in place parallel superaccumulator summation, Algorithm 4.

4. $S \leftarrow \text{convertToFloatAndRound}(Y_G)$, only first $L$ elements of $Y_G$

\[
T = \begin{cases} 
\mathcal{O} \left[ \left( \frac{2(p+1)+\log n - \log \log n}{r} + 2 \right) \text{reduce}(n) \right], & \text{if } \log C(X) < p + 1 - \log \log n \\
\mathcal{O} \left[ \left( \frac{\log S}{r} + 2 \right) \text{reduce}(n) \right], & \text{otherwise}
\end{cases}
\]

5.1.1 Implementation details

The main bottleneck for this algorithm is memory bandwidth due to the large amount of memory needed to expand the inputs into $n$ superaccumulators of length $L$. In order to reduce this memory footprint we have each block read in a number of f.p. inputs, using Algorithm 4 to add each new super-digit to a local, per-block superaccumulator before writing to global memory. Furthermore, we vectorize I/Os when dealing with superaccumulator digits (not f.p. inputs) effectively having each thread deal with multiple super-digits instead of just one. This then allows us to have each block maintain more than one superaccumulator at a time. These optimizations were necessary in order to produce reasonable runtimes.

A final optimization step involves how we store local per-block superaccumulators. In general we would employ shared memory for this as we need communication between threads in order to see the carry-digit produced in each summation step (see Algorithm 4). However, in this work we only use GPUs that have the warp-shuffle functionality (see [13] for details) and thus we employ warp shuffles for communication and to store the local per-block superaccumulators in local per-thread memory (as individual super-digits) providing a small boost in performance\textsuperscript{10}

One last thing to mention is the computation of the bound on $L$. To do this we use the exact same procedure as the global SA-summation algorithm with one minor modification. Due to the architecture of the GPU it is necessary to fix $L$ to be itself a power-of-two (otherwise not all threads in a block will be active). Thus, when we compute a bound we always round $L$ up to the nearest power of two.

\textsuperscript{10}This adds the further requirement that each superaccumulator must fit into a single warp of threads. This is possible by having each thread deal with multiple super-digits.
5.2 Local GSD-summation

The local version of this algorithm is almost identical to the local SA-summation algorithm. The only differences being that each thread uses Algorithm 4 to add new inputs to its local superaccumulator, and the reduction of superaccumulators is also performed using Algorithm 4. Other than this, everything else is exactly the same. The pseudocode for the algorithm is given below.

Algorithm 6 Local GSD-Summation
Input: $n$ double-precision f.p. numbers $x_i \in X$
Output: double-precision f.p. number $S$ representing exact sum of $X$
Assume $B$ blocks of $L$ threads.

1. Initialize array $Y_G$ of length $BL$ to be zero. This will hold all per-block superaccumulators.
2. For all $b \in [0, B-1]$ do in parallel:
   - For all $k \in [0, L-1]$ do in parallel:
     - Initialize local superaccumulator $Y_k$ of length $L$ to be zero.
     - For $j$ in $0$ to $n/(BL)$,
       - $P, Q \leftarrow \text{getSuperDigits}(x_i)$, super-digits in $P$, locations in $Q$
       - $Y_k \leftarrow \text{carryFreeAddSuperDigitList}(P, Q)$
     - Initialize per-block superaccumulator $Y_b$ of length $L$ in shared memory to zero.
   - For all $k \in [0, L-1]$ do in parallel:
     - For $j = 0$ to $L - 1$:
       - $u \leftarrow (k + j) \mod L$
       - $Y_b[u] \leftarrow \text{carryFreeAddSuperDigits}(Y_b[u], Y_k[u])$
     - $Y_G[b \cdot L : b \cdot (L + 1) - 1] \leftarrow Y_b[\cdot L]$, write per-block superaccumulators to global memory.
3. $Y_G \leftarrow \text{parallelReduceSA}(Y_G)$, in place reduction of all per-block superaccumulators.
4. $S \leftarrow \text{convertToFloatAndRound}(Y_G)$, only first $L$ elements of $Y_G$.

In Algorithm 6 the function $\text{getSuperDigits}(x_i)$ returns a fixed number (dependent on $r$, the exponent of our power-of-two radix) of super-digits representing the entire f.p. number $x_i$, as well as the location of those components in $Y_k$. The function $\text{carryFreeAddSuperDigitList}(P, Q)$ adds the super-digits $P$ to the locations $Q$ in $Y_k$ using Algorithm 4, and the function $\text{carryFreeAddSuperDigits}(Y_b[u], Y_k[u])$ adds the super-digits at position $u$ together using Algorithm 4. The function $\text{parallelReduceSA}(Y_G)$ reduces the $B$ superaccumulators in $Y_G$ to a single superaccumulator using Algorithm 4, and, finally, the function $\text{convertToFloatAndRound}(Y_G)$ converts the final superaccumulator to a f.p. number, rounding appropriately.

Again, for this local version we did not implement a bound on $L$, although this is a possibility in future work. The implementation details are the same as the local SA-summation algorithm, with register pressure being the main bottleneck, and, as a result, an
6 Experimental results

In this section we present the results of runtime experiments for our algorithms. The algorithms that have been explicitly tested are:

1. Pair-Sum, this a normal parallel reduction of the f.p. inputs. It is not exact.
2. OnlineSum, this is the state-of-the-art sequential exact summation algorithm [16].
3. GlobalSA, this is our global SA-summation algorithm (with bound, see Appendix A).
4. LocalSA, this is our local SA-summation algorithm
5. LocalSA-sort, this is our local SA-summation algorithm with an initial sort-by-exponent performed.
6. GlobalGSD, this is our global GSD-summation algorithm (with bound, see Appendix A).
7. LocalGSD, this is our local GSD-summation algorithm
8. LocalGSD-sort, this is our local GSD-summation algorithm with an initial sort-by-exponent performed.

In all cases we are dealing with IEEE-754 double-precision inputs (allowing subnormals), and for all cases a GeForce GTX TITAN GPU was used.

In Figure 4 the effect of varying the exponent range of the input data is shown. Again, in all cases the data is distributed uniformly across the respective exponent ranges, and the number of elements summed is $2^{26} \approx 70 \times 10^6$. As the condition number is not changing in this algorithm both of our global algorithms show no dependence on the exponent range, and these end up being the fastest two parallel exact summation algorithms. The two local algorithms (without the initial sort) show a strong dependence on the exponent range, while the two local algorithms with per-sorting on the exponents do not have any dependence on the exponent range. This shows that the sort effectively removes the register pressures caused by widely distributed inputs. It is interesting to note that the difference in runtimes between the global algorithms and local algorithms with a sort is just about equal to the time it takes to do a sort-by-exponent of the input data. The fastest parallel algorithm (GlobalGSD) is roughly ten times faster than the state-of-the-art sequential Online Exact summation algorithm.

In Figure 5 the effect of varying the condition number of the input data is shown. To generate these datasets we generate numbers uniformly across a fixed exponent range in such a way that the sum is a very small value, and the condition numbers plotted are the average condition-numbers of the data. The number of elements summed is $2^{26}$. In this case the global algorithms have a dependence as expected from the complexity bounds derived in Appendix A. The plateaus in the global GSD algorithm are due to the restriction of the superaccumulator representation having a length of a power-of-two, with jumps between plateaus occurring when this length increases to accommodate the more poorly conditioned
datasets. The apparent condition number dependence of the local algorithms without pre-
sorts are due to the fact that we need to increase the exponent range to get datasets with
these condition numbers. The local algorithms with a sort are again not affected, and thus
are the fastest algorithms when the datasets are poorly conditioned (around 4× faster than
the Online Exact Sum in this case).

The effect of varying the dataset size is shown in Figures 6, 7, and 8 for inputs with
exponent ranges of $2^5$, $2^{500}$, and $2^{1500}$ respectively. In all cases the data is distributed
uniformly across these exponent ranges (such that $C(X)$ will be small). We can see from
these plots that, for very small exponent ranges, the local algorithms without a sort perform
the best, followed closely by the global GSD algorithm. However, as the exponent range
increases the benefit of using the sort becomes more apparent. We can also see from these
plots that it is not until a data size of roughly $2^{22}$ that the parallel algorithms have a linear
dependence on the data-size.

Unfortunately it is difficult to make a direct comparison with results presented in other
work as the GPUs used are not the same as the one used in this work. In [2] the authors
were able to sum around $2^{26}$ in about 5 ms for a small input exponent range ($< 2^{132}$). They
were, however, using a much more powerful GPU (Tesla K20) and thus in comparison our
runtimes are not that far off from what is achieved in [2].

We would have also liked to compare to the GSD-summation algorithm provided in [5]
as this was the paper that inspired this work. Specifically, the sparse-superaccumulator
algorithms in [5] seemed promising for effective implementations on the GPU, however, in
practice, we found that the changing of superaccumulator sizes during runtime (as needed
in their algorithm) did not fit well with the GPU and is what led us to the algorithms
presented here. More unfortunate still is that the implementation in [5] was not completed
on a GPU and thus a direct comparison to their work is not possible. Suffice it to say,
however, that the runtime of the implementation given in [5] matches, almost identically,
to the state-of-the-art sequential Online Exact Sum algorithm used as a comparison in this
paper.

To conclude this section we note that the characteristics of the dataset really drive the
runtimes of the various algorithms. For the global versions of our algorithms we employed
two parallel reductions in order to bound our superaccumulators to just accommodate
the numbers and still produce an exact sum, and were able to remove the dependence on the
exponent range, in return making the algorithms dependent on condition number. The
local versions of our algorithms perform well only for small exponent ranges, and would
be the algorithms of choice if it is known ahead of time that the input data does have a
small exponent range. The local versions of our algorithms with the initial pre-sort are
pretty good general purpose algorithms as they have no dependence at all on the dataset
characteristics. It would be interesting to investigate a way that could switch between these
algorithms depending on the input data in order to always choose the best algorithm to
suit the data at hand.
Figure 4: Runtime dependence of various algorithms on the exponent range of the input dataset. In all cases $n = 2^{26}$.

Figure 5: Runtime dependence of various algorithms on the condition-number of the input dataset. In all cases $n = 2^{26}$. 
Figure 6: Runtime dependence on the number of data elements for an exponent range of $2^5$.

Figure 7: Runtime dependence on the number of data elements for an exponent range of $2^{500}$.
Figure 8: Runtime dependence on the number of data elements for an exponent range of $2^{1500}$. 
7 Conclusions

To conclude, we have presented various parallel exact summation algorithms specifically for implementation on a GPU. These algorithms show varying dependencies on the characteristics of the input data, and, if they are used in practice, one should choose an algorithm to best suit the characteristics of the data at hand. These algorithms also appear to be quite fast, with one or many outperforming the best sequential exact summation algorithm for all dataset conditions. With more time it is expected that a more detailed analysis of register and shared memory usage could increase the performance of these algorithms even further.

A Global summation bounds

We determine the bounds for the global summation algorithms using two parallel reductions: one is used to get an estimate of the sum of positive elements, $S_+$, and the other to get an estimate of the sum of negative elements, $S_-$ for the data,

$$ S_+ = \oplus_{i=0,x_i > 0}^{n-1} x_i, \quad S_+ \geq 0 $$

$$ S_- = \oplus_{i=0,x_i < 0}^{n-1} x_i, \quad S_- \leq 0 $$

For both of these sums the condition number is unity and, since we are doing this in parallel, the relative error bound for both of these datasets is that for a pair-wise sum, Equation 10,

$$ R_{S_+,S_-} \leq \frac{\log n \times \epsilon}{1 - \log n \epsilon} \approx 2^{\log \log n - p - 1} $$

or,

$$ \log R_{S_+,S_-} < -(D + 1), \quad D = p + 1 - \log \log n $$

where $p$ is the number of mantissa bits for the f.p. representation.

This tells us that we will have at least $D$ significant binary-digits in each sum and can thus trust the first $D + 1$ digits of $S_+$ and $S_-$, and the first $D$ digits of their sum. If any of these $D$ digits are non-zero then we can say with certainty what the most-significant-digit (msd) of the exact sum is.

Furthermore, as we define an exact sum as any sum with a relative error $\leq \epsilon$ (as compared to the truly exact sum) we will not need to consider any digits that fall a distance of $\log n + p + 1$ digits from the first non-zero digit of the sum of $S_+$ and $S_-$. If we form the two sums,

$$ \hat{S} = |S_+ + S_-| $$

and

$$ \bar{S} = |S_+| + |S_-| $$

and let $e_{\text{max}}$ be the binary position of the msd of $\bar{S}$,

$$ e_{\text{max}} = \left\lceil \log \bar{S} \right\rceil $$
and set,
\[
e_{\text{min}} = \begin{cases} 
\lceil \log \hat{S} \rceil - \log n - p - 1, & \text{if non-zero msd found in first } D \text{ digits} \\
0, & \text{otherwise}
\end{cases}
\]
(15)
then the only binary digits we have to consider in order to get an exact sum are those that fall into the range \([e_{\text{min}}, e_{\text{max}}]\).

For the global SA-summation algorithm this means we only have to calculate the super-digits of our superacucumulator representation which fall in \([e_{\text{min}}/r, e_{\text{max}}/r]\). This translates into a time complexity for the global SA-summation algorithm of\(^{11}\)
\[
T = \begin{cases} 
\mathcal{O}\left[\left(\frac{2^{(p+1)+\log n - \log \log n}}{r} + 2\right) \text{reduce}(n)\right], & \text{if } \log C(X) < p + 1 - \log \log n \\
\mathcal{O}\left[\left(\frac{\lceil \log \hat{S} \rceil}{r} + 2\right) \text{reduce}(n)\right], & \text{otherwise}
\end{cases}
\]
(16)

For the global GSD-summation algorithm we bound the actual length of the superaccumulator \(L\),
\[
L_b = \left\lceil \frac{e_{\text{max}} - e_{\text{min}}}{r} \right\rceil
\]
which results in the exact same time complexity as the SA-summation algorithm given above.

References


\(^{11}\)Finding a non-zero msd in the first \(D\) digits of \(\hat{S}\) is equivalent to the statement that \(\log C(X) < D = p + 1 - \log \log n\).


