Parallel floating-point expansions for extended-precision GPU computations

Caroline Collange, Mioara Joldes, Jean-Michel Muller, Valentina Popescu

Parallel floating-point expansions for extended-precision GPU computations

Caroline Collange
INRIA Rennes
caroline.collange@inria.fr

Mioara Joldes
LAAS CNRS, Toulouse
joldes@laas.fr

Jean-Michel Muller
CNRS, ENS Lyon
jean-michel.muller@ens-lyon.fr

Valentina Popescu
ENS Lyon
valentina.popescu@ens-lyon.fr

***Abstract*—GPUs are an important hardware development platform for problems where massive parallel computations are needed. Many of these problems require a higher precision than the standard double floating-point (FP) available. One common way of extending the precision is the multiple-component approach, in which real numbers are represented as the unevaluated sum of several standard machine precision FP numbers. This representation is called a FP expansion and it offers the simplicity of using directly available and highly optimized FP operations.**

In this article we present new data-parallel algorithms for adding and multiplying FP expansions specially designed for extended precision computations on GPUs. These are generalized algorithms that can manipulate FP expansions of different sizes (from double-double up to a few tens of doubles) and ensure a certain worst case error bound on the results.

***Index Terms*—floating-point arithmetic, floating-point expansions, high precision arithmetic, multiple-precision arithmetic, graphics processing unit, parallel computations**

I. Introduction

In numerical computing, we sometimes encounter calculations that require more precision than the one offered by current processors. A way of handling such higher-precision calculations is to represent real numbers by floating-point (FP) *expansions*, i.e., by unevaluated sums of FP numbers. Several slightly different definitions of what a FP expansion is have been introduced in the literature, together with different arithmetic algorithms for manipulating them [1], [2], [8]. Choosing between these algorithms frequently depends on a compromise between accuracy, speed, and safety (because some of the more complex algorithms are just heuristic: they do not come with a proof). With Graphics Processing Units (GPUs) oriented implementation in mind, other important aspects are parallelism and locality. Should we try to parallelize the arithmetic algorithms that manipulate FP expansions (or, rather, build variants that are suitable for parallelization), or should we keep them sequential and try to parallelize at a higher level? In a previous study [3], some of us have dealt with an "embarrassingly parallel" problem with compact intermediate data. However, many applications do not provide as much parallelism. Even for those that do, locality can be a problem. Increasing the precision of sequential arithmetic operations requires a corresponding increase in the amount of intermediate data to keep. Thus, parallel arithmetic algorithms are attractive not just by the extra parallelism they provide, but also by the locality improvements they enable. Here, with different numerical problems in mind, we parallelize the addition and multiplication of FP expansions on GPUs.

Section II recalls the basic notions related to FP expansions. In Sections III and IV we present data-parallel algorithms for addition and multiplication, respectively, of FP expansions. The implementation details are given in Section V followed by some performance assessment in Section VI. We finish by concluding our work in Section VII.

II. Floating-point expansions

A normal binary precision-p floating-point (FP) number is a number of the form

$$x = M_x \cdot 2^{e_x - p + 1},$$

with $2^{p-1} \leq |M_x| \leq 2^p - 1$. The integer e_x is called the *exponent* of x, and $M_x \cdot 2^{-p+1}$ is called the *significand* of x. Using Golberg's definition we denote $\mathrm{ulp}(x) = 2^{e_x - p + 1}$ [4, Chap. 2].

A natural generalization of the notion of double-double or quad-double is the notion of *floating-point expansion*.

Definition II.1. A *FP expansion* u with n terms is the unevaluated sum of n FP numbers u_0, u_1, ..., u_{n-1}, in which all nonzero terms are ordered by magnitude (i.e., if v is the sequence obtained by removing all zeros in the sequence u, and if sequence v contains m terms, $|v_i| \geq |v_{i+1}|$, for all $0 \leq i < m - 1$).

Arithmetics on FP expansions have been introduced by Priest [1], and later on by Shewchuk [2].

To make sure that such an expansion carries significantly more information than only one FP number, it is required that the u_i's do not "overlap". The notion of *(non-)overlapping* varies depending on the authors. Intuitively, the stronger the sense of the nonoverlapping definition, the more difficult it is to obtain, implying extra manipulation of the FP expansions. In what follows we give the definition that we chose to use, referred to as ulp-*nonoverlapping*, in an attempt to allow for a more relaxed manipulation of the FP expansions. We specify first that even if a FP expansion may contain interleaving zeros, the definition that follows applies only to the non-zero terms of the expansion (i.e., the array v in Definition II.1).

Definition II.2. An expansion u_0, u_1, ..., u_{n-1} is ulp-*nonoverlapping* if for all $0 < i < n$, we have $|u_i| \leq \mathrm{ulp}(u_{i-1})$.

In other words, the components are either $\mathcal{P}$-*nonoverlapping* (that is, nonoverlapping according to Priest's definition [5]) or they overlap by one bit, in which case the second component is a power of two.

Depending on the nonoverlapping type of an expansion, when using standard FP formats for representation, the exponent range forces a constraint on the number of terms. The largest expansion can be obtained when the largest term is close to overflow and the smallest is close to underflow. We remark that, when using ulp-*nonoverlapping* expansions, for the two most common FP formats, the constraints are:

- for *double*-precision (exponent range $[-1022, 1023]$) the maximum expansion size is 39;
- for *single*-precision (exponent range $[-126, 127]$) the maximum is 12.

In this article we deal with so called parallel FP expansions, i.e., the expansion is stored on parallel execution threads, with one term/thread. This implies that the user has to launch as many threads as the expansion size.

The majority of algorithms performing arithmetic operations on expansions are based on the so-called *error-free transforms* (EFT) (such as the algorithms 2Sum, Fast2Sum, Dekker's product and 2MultFMA presented for instance in [4]), that make it possible to compute both the result and the rounding error of a FP addition or multiplication. This implies that in general, each such EFT applied to two FP numbers, also returns two FP numbers. More precisely, the sum of two FP numbers can be represented *exactly* as a FP number which is the correct rounding of the sum, plus a second FP number corresponding to the rounding error. Under certain assumptions, this decomposition can be computed at a very low cost by a simple sequence of standard precision FP operations. For instance, assuming that a and b are two FP numbers, a simple algorithm (called *2Sum* and due to Knuth [4]) computes S and E, the decomposition of $a + b$ using only 6 FP operations. Similarly, if a FMA operator[1] is available, *2ProdFMA* [4] returns P and the error E (namely $ab - P$) in 2 FP operations.

Algorithms like these can be extended to be used with arbitrary precision computations by chaining, resulting into the so-called *distillation* algorithms [6], for summing several FP numbers. From these we make use of the *VecSum* algorithm [2], [7], which is simply a chain of *2Sum* that performs an EFT on n FP numbers.

A potential problem appears when subsequent computations are done using these results; the size of the exact result is going to increase more and more. To avoid this, some "truncation" methods (both *on-the-fly* or *a-posteriori*) may be used to compute only an approximation of the exact result. Also, so-called *(re-)normalization* algorithms are used to render the result non-overlapping, which implies also a potential reduction in the number of components.

[1]A FMA (Fused Multiply-Add) operator evaluates an expression of the form $xy + t$ with one final rounding only.

III. Data-parallel addition algorithm for FP expansions

An algorithm that performs addition of two FP expansions x and y with n and m terms, respectively, will return a FP expansion with at most $n + m$ terms as the exact result. This implies a continuous increase in the number of terms as subsequent computations are done using the obtained result. This is why, in practice, the results are "truncated" to obtain only an approximation of $x + y$.

Many variants of algorithms that compute the sum of two FP expansions have been presented in the literature [1], [2], [8], [6], each using different methods and having different complexities, but, from our knowledge, none of these algorithms are implemented in parallel and, even more, some of them are highly sequential, making them unsuitable for parallel architectures.

In what follows we will present a safe data-parallel algorithm for adding FP expansions that offers a tight error bound on the result and it allows to prove a clear constraint between the terms of the result. We also present a fast version with more relaxed error bounds based on the same scheme that may be used if computations are not cancellation-prone.

The safe data-parallel summation algorithm is presented in Fig. 1 and Algorithm 1. All arithmetic operations including EFTs like 2Sum are performed in parallel element-wise on R-element vectors. We assume vectors can be merged and elements inside a vector can be shuffled. These assumptions make the algorithms applicable to most SIMD units, including Intel SSE/AVX instruction set extensions [9] and recent Nvidia GPUs [10].

Algorithm 1 Safe data-parallel algorithm for adding FP expansions.

Input: FP expansion vectors $\mathbf{x} = (x_0, x_1, \ldots, x_{R-1})$ and $\mathbf{y} = (y_0, y_1, \ldots, y_{R-1})$.
Output: FP expansion vector $\mathbf{s} = (s_0, s_1, \ldots, s_{R-1})$.

$\mathbf{a} \leftarrow (x_0, 0, 0, \ldots, 0)$
$\mathbf{b} \leftarrow (y_0, 0, 0, \ldots, 0)$
$(\mathbf{s}, \mathbf{e}) \leftarrow 2\mathrm{Sum}(\mathbf{a}, \mathbf{b})$
for $i \leftarrow 1$ to R **do**
  $\mathbf{e}' \leftarrow (x_i, e_0, e_1, \ldots, e_{R-2})$ //Shift right & insert x_i
  $(\mathbf{s}, \mathbf{e}) \leftarrow 2\mathrm{Sum}(\mathbf{s}, \mathbf{e}')$
  $\mathbf{e}' \leftarrow (y_i, e_0, e_1, \ldots, e_{R-2})$ //Shift right & insert y_i
  $(\mathbf{s}, \mathbf{e}) \leftarrow 2\mathrm{Sum}(\mathbf{s}, \mathbf{e}')$
end for
for $i \leftarrow 1$ to $R - 2$ **do**
  $\mathbf{e}' \leftarrow (0, e_0, e_1, \ldots, e_{R-2})$
  $(\mathbf{s}, \mathbf{e}) \leftarrow 2\mathrm{Sum}(\mathbf{s}, \mathbf{e}')$
end for
$\mathbf{e}' \leftarrow (0, e_0, e_1, \ldots, e_{R-2})$//Shift right
$\mathbf{s} \leftarrow \mathbf{s} + \mathbf{e}'$
return $\mathbf{s}$.

For the sake of simplicity, we only present here the "input-R-output-R" version of the algorithm, even though the generalized version allows for different input sizes.

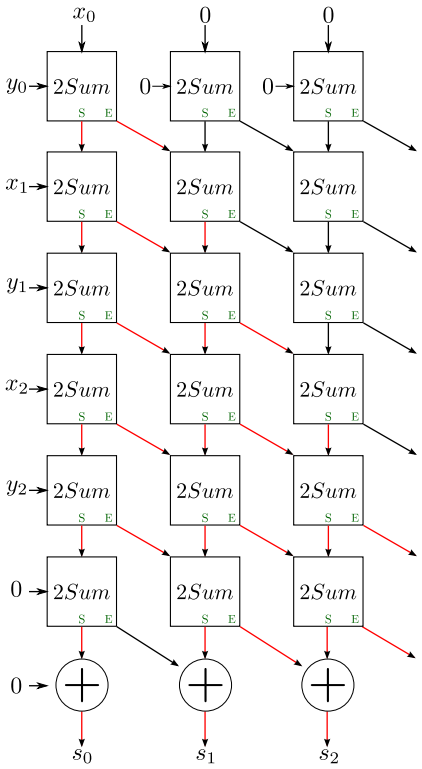

Fig. 1: Safe data-parallel FP expansion addition algorithm, illustrated for the case when $R = 3$ terms.

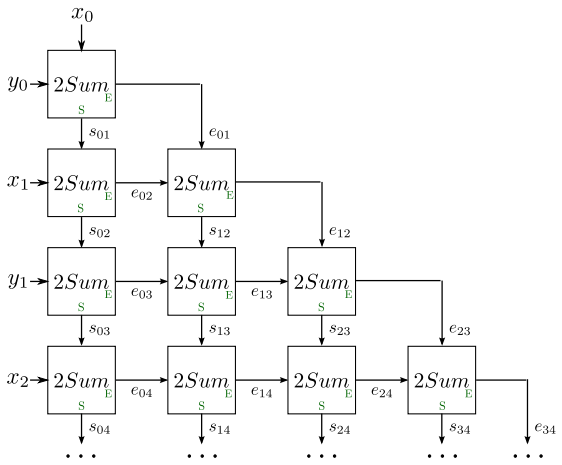

Fig. 2: Sequential scheme of the FP expansion addition algorithm.

The algorithm is based on a pipelined error propagation. We start by adding the first elements of each expansions, x_0 and y_0 on the first vector component. We continue to add the rest of the elements on the first component one by one and propagating the error upwards, to the other vector components. When we run out of elements to add we continue to propagate the errors for another $R-1$ steps by injecting 0s on the first component. In the last step of the algorithm there is no need to use EFT, since we are not going to propagate the errors anymore; this is why we use only simple addition.

By using this scheme to add the two expansions we ensure that the most significant term of the output, s_0, is the sum of the inputs rounded to nearest. Moreover, the terms of the output are arranged in terms of magnitude in decreasing order, with some constraints. Let us prove this in what follows.

It is easily seen that the parallel scheme presented in Fig. 1 can be reduced to the sequential one presented in Fig. 2.

If there is a "Sterbenz relation" between x_0 and y_0 (i.e., if they are of opposite signs and $\left|\frac{x_0}{2}\right| \leq |y_0| \leq |2x_0|$) then $e_{01} = 0$ and $s_{01} = x_0 + y_0$. This implies that $s_{12} = e_{02}$ and $e_{12} = 0$, and so on. In this case we end up propagating a 0, to the end of the result expansion, and we are left with the same scheme as we began with (illustrated in Fig. 3). This means that we can eliminate the case in which we have a "Sterbenz relation".

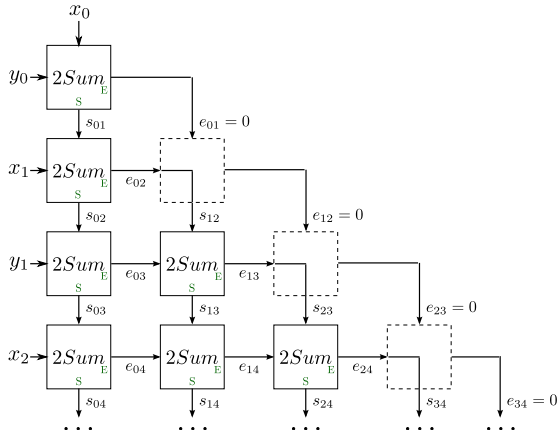

Fig. 3: Reduction of the sequential scheme in Fig. 2 based on the "Sterbenz relation".

Now let us prove the following proposition that refers to only one horizontal line of the scheme, as presented in Fig. 4.

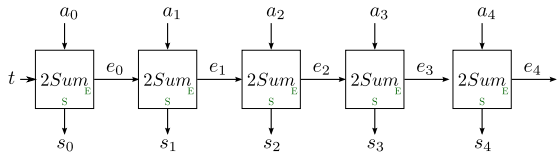

Fig. 4: Addition of a FP number t to the array a, starting from the left side and propagating the error.

Proposition III.1. *Let us assume $a = a_0, a_1, \ldots a_{n-1}$, an array of FP numbers that satisfy $|a_i| \leq 2^{-i(p-1)+\delta}|a_0|$, for some (presumably small) integer δ and $|t| \leq 2^{-p+\ell}|a_0|$, for some (presumably small) integer ℓ. If $s = s_0, s_1, \ldots, s_n$ is the sum obtained by adding t to a as shown in Fig. 4 (from left to right, propagating the error), then all the terms in s satisfy: $|s_i| \leq 2^{-i(p-1)+\delta+1}$, for all $0 < i \leq n$.*

Proof: From the proof of the algorithm *2Sum* we know that $|e_i| \leq \frac{1}{2}\,\mathrm{ulp}(s_i) = 2^{-p}|s_i|$.

From

$$\begin{aligned} |s_0|(1-2^{-p}) \leq \quad & |a_0+t| \quad \leq |s_0|(1+2^{-p}) \text{ and} \\ |a_0|(1-2^{-p+\ell}) \leq \quad & |a_0+t| \quad \leq |a_0|(1+2^{-p+\ell}), \end{aligned}$$

we get

$$|s_0|\frac{1-2^{-p}}{1+2^{-p+\ell}} \leq |a_0| \leq |s_0|\frac{1+2^{-p}}{1-2^{-p+\ell}}. \qquad (1)$$

It follows that

$$|a_1| \leq 2^{-(p-1)+\delta}|a_0| \leq 2^{-(p-1)+\delta} \cdot \frac{1+2^{-p}}{1-2^{-p+\ell}} \cdot |s_0|.$$

This gives

$$|e_0 + a_1| \leq 2^{-p}\left[1 + \frac{2^{\delta+1}(1+2^{-p})}{1-2^{-p+\ell}}\right]|s_0|.$$

From which we deduce

$$|s_1| \leq 2^{-p}(1+2^{-p})\left[1 + \frac{2^{\delta+1}(1+2^{-p})}{1-2^{-p+\ell}}\right]|s_0|. \quad (2)$$

We can continue, by noticing that $|e_1|$ is bounded by $2^{-p}|s_1|$, and bounding $|a_2|$ by $2^{-2(p-1)+\delta}\frac{1+2^{-p}}{1-2^{-p+\ell}} \cdot |s_0|$. This gives a bound on $|e_1 + a_2|$, and a bound on s_2 is obtained by multiplying that last bound by $(1+2^{-p})$. An easy induction finally gives

$$|s_i| < 2^{-ip}\theta_i|s_0|, \quad (3)$$

with

$$\theta_i = (1+2^{-p})^i + \frac{1}{1-2^{-p+\ell}}\sum_{j=1}^{i} 2^{j+\delta}(1+2^{-p})^{i-j+2}. \quad (4)$$

One easily finds

$$\theta_i = (1+2^{-p})^i + \frac{2^{\delta+1}(1+2^{-p})^2}{1-2^{-p+\ell}}\left[\frac{2^i - (1+2^{-p})^i}{1-2^{-p}}\right],$$

hence,

$$\theta_i = 2^{i+\delta+1}H_i,$$

with

$$H_i = \frac{(1+2^{-p})^i}{2^{i+\delta+1}} + \frac{(1+2^{-p})^2}{1-2^{-p+\ell}}\left(\frac{1-\frac{(1+2^{-p})^i}{2^i}}{1-2^{-p}}\right).$$

Denote $u = 2^{-p}$. In all practical cases $\ell \geq 2$ and $\delta \geq 0$, so that $H_i \leq G_i$, with

$$G_i = \frac{1}{2}\left(\frac{1+u}{2}\right)^i + \frac{(1+u)^2}{1-4u}\left(\frac{1-\left(\frac{1+u}{2}\right)^i}{1-u}\right).$$

We have,

$$\begin{aligned} G_i \quad = \quad & 1 - \tfrac{1}{2}\left(\tfrac{1+u}{2}\right)^i + \tfrac{u(u+6)}{1-4u} - \\ & -\tfrac{1}{2}\left(\tfrac{1+u}{2}\right)^i \cdot \left(\tfrac{2u(7-3u)}{(1-4u)(1-u)}\right) \end{aligned}$$

The only positive term (after the initial "1") in that sum is $\frac{u(u+6)}{1-4u}$, which is less than $7u$ for all pertinent values of $u = 2^{-p}$. Hence $G_i < 1$ as soon as $2^{i+1} \leq 2^p/7$, which occurs in all practical cases. This gives

$$|s_i| < 2^{-i(p-1)+\delta'}|s_0|,$$

with $\delta' = \delta + 1$.

This concludes our proof. ∎

Hence, in the array of Fig. 1, δ is increased by 1 at each line. For instance, if we add two order-n expansions (i.e., we use $2n-1$ lines in the Array of Fig. 1, then the terms $s_0, s_1, \ldots$ of the result satisfy

$$|s_i| \leq 2^{-(p-1)i+2n-1}|s_0|,$$

and the expansion obtained by keeping the first n terms only represents the sum with an error less than

$$(2^{-np+3n-1} + 2^{-(n+1)p+3n} + 2^{-(n+2)p+3n+1} + \cdots) \cdot |a_0|,$$

i.e., with a relative error bounded by a value slightly larger than $2^{-np+3n-1}$.

We would like to also mention here a faster, relaxed version of the above algorithm, that requires at most $R-1$ steps in order to compute the result (the last step using only simple addition). This algorithm (Fig. 5) offers a worse error bound and it does not ensure the correct result when cancellation occurs, if no *re-normalization* algorithm is applied on the result. It performs best when FP expansions of the same sign and close magnitudes are added. We recommend using it for fast computations that can be validated a-posteriori.

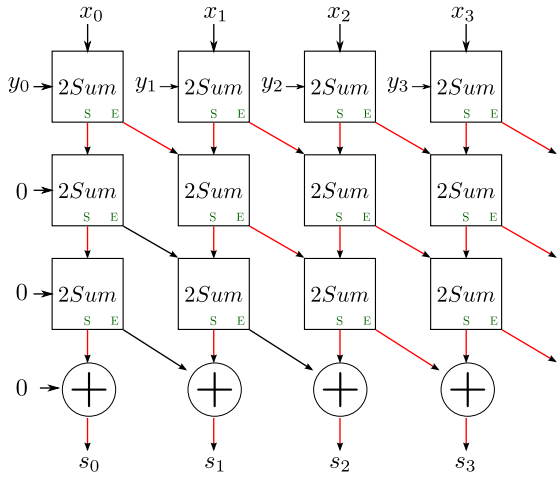

Fig. 5: Fast data-parallel FP expansion addition algorithm, illustrated for the case when $R = 4$ terms.

IV. Data-parallel multiplication algorithm for FP expansions

In general, an algorithm that performs the multiplication of two expansions x and y with n and m terms, respectively, will return a FP expansion with at most $2nm$ terms [1], which, as in the case of addition, implies an increase in the number of terms.

The algorithm that we present (Algorithm 2) computes an approximation of $x \cdot y$, where x and y are two parallel expansion. Here we also present just the "input-R-output-R" version.

We consider two parallel FP expansions x and y, each with R terms and we compute the R most significant FP components of the product $\pi = x \cdot y$. We use the following intuition: let $\varepsilon = \frac{1}{2}\,\mathrm{ulp}(\pi_0)$, then roughly speaking, if π_0 is of order of $\mathcal{O}(\Lambda)$, then e_0 is of order $\mathcal{O}(\varepsilon\Lambda)$. This means that for each product $(p, e) = \textit{2MultFMA}(x_i, y_j)$, p is of order $\mathcal{O}(\varepsilon^{i+j}\Lambda)$ and e of order $\mathcal{O}(\varepsilon^{i+j+1}\Lambda)$. We truncate the result on-the-fly, by considering only the terms for which $0 \leq i+j \leq R-1$, since the smaller terms have an order of magnitude much smaller and usually they will not influence the result.

Algorithm 2 Data-parallel algorithm for multiplying FP expansions.

Input: FP expansion vectors $\mathbf{x} = (x_0, x_1, \ldots, x_{R-1})$ and $\mathbf{y} = (y_0, y_1, \ldots, y_{R-1})$.
Output: FP expansion vector $\pi = (\pi_0, \pi_1, \ldots, \pi_{R-1})$.

1: $\mathbf{s} \leftarrow (0, \ldots, 0)$
2: $\pi \leftarrow (0, \ldots, 0)$
3: **for** $i \leftarrow 0$ to $R-2$ **do**
4: $\quad \mathbf{y}' \leftarrow (y_i, y_i, \ldots, y_i)$ //Broadcast
5: $\quad (\mathbf{p}, \mathbf{e}) \leftarrow 2\mathrm{MultFMA}(\mathbf{x}, \mathbf{y}')$
6: $\quad (\mathbf{s}, \mathbf{e}') \leftarrow 2\mathrm{Sum}(\mathbf{s}, \mathbf{p})$
7: $\quad \pi_i \leftarrow s_0$ //Insert into vector
8: $\quad \mathbf{s} \leftarrow (s_1, s_2, \ldots s_{R-1}, 0)$ //Shift left
9: $\quad$ **while** $\mathbf{e} \neq 0$ **do**
10: $\quad\quad (\mathbf{s}, \mathbf{e}) \leftarrow 2\mathrm{Sum}(\mathbf{s}, \mathbf{e})$
11: $\quad\quad \mathbf{e} \leftarrow (0, e_0, e_1, \ldots, e_{R-2})$ //Shift right
12: $\quad$ **end while**
13: $\quad$ **while** $\mathbf{e}' \neq 0$ **do**
14: $\quad\quad (\mathbf{s}, \mathbf{e}') \leftarrow 2\mathrm{Sum}(\mathbf{s}, \mathbf{e}')$
15: $\quad\quad \mathbf{e}' \leftarrow (0, e'_0, e'_1, \ldots, e'_{R-2})$ //Shift right
16: $\quad$ **end while**
17: **end for**
18: $\mathbf{p} \leftarrow \mathbf{x} \cdot \mathbf{y}$
19: $\mathbf{s} \leftarrow \mathbf{s} + \mathbf{p}$
20: $\pi_{R-1} \leftarrow s_0$ //Insert into vector
21: **return** π.

The multiplication algorithm runs as follows: at each iteration i of the for loop (lines 3-16) we compute $p+e = x \cdot y$; we add p to the result of the same order, using an EFT, which also generates an error, e'. After that, using the two while loops (lines $8-11$ and $12-15$) we propagate the two generated errors, e and e' to the lower order results. In the last step of the algorithm, we do not use any EFT, because the errors that are supposed to be computed are going to be of order $\mathcal{O}(\varepsilon^R \Lambda)$, and we do not need to propagate them anymore. This algorithm has the same behavior as the sequential algorithm presented in Fig. 6, but a graphical representation of the parallel one would be too difficult to read.

While the exact product xy of two FP expansions x and y is computed as $\sum_{i=0;j=0}^{R-1;R-1} x_i y_j = \sum_{k=0}^{2R-2} \sum_{i+j=k} x_i y_j$, in this algorithm we "truncate" it on-the-fly by computing and adding only the relevant part of the scalar products (the first $\sum_{k=1}^{R} k$ individual products) and after that outputting the R most significant components in the result. In what follows we compute the error bound in two steps: first we compute the error generated by "truncating" the partial products, and second we compute the error given by the discarded errors.

Proposition IV.1. *(Error bound on the truncated products) Let x and y be two* ulp-nonoverlapping *FP expansions, with R terms. If, when computing the product xy we "truncate" the operations by computing and adding only the most significant individual products, the first $\sum_{k=1}^{R} k$ products, then the maximum error that we can obtain is: $\left| xy - \sum_{k=0}^{R-1} \sum_{i+j=k} x_i y_j \right| = \sum_{k=R}^{2R-2} \sum_{i+j=k} x_i y_j \leq |x_0 y_0|\, 2^{-(p-1)R} \frac{R-1}{1-2^{-(p-1)}}$.*

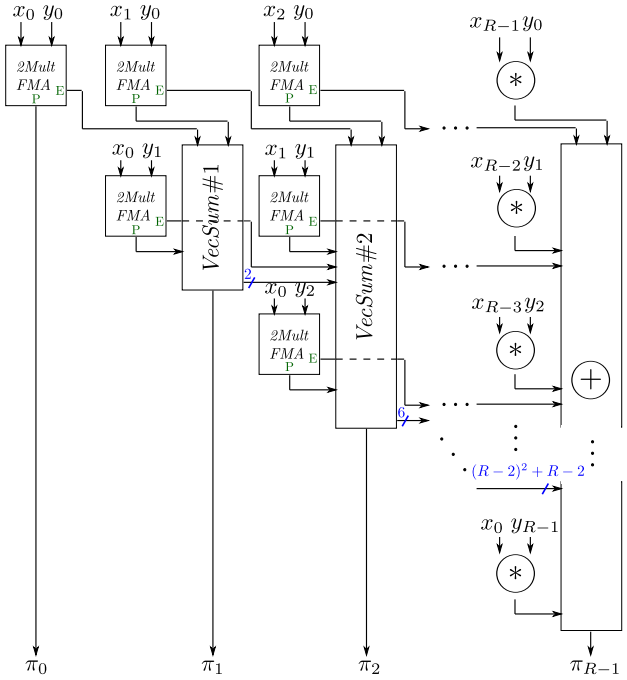

Fig. 6: Graphical representation of the sequential algorithm that behaves like Algorithm 2.

Proof: The maximum error given by the discarded products satisfies:

$$\begin{aligned}
\sum_{k=R}^{2R-2} \sum_{i+j=k} x_i y_j &\leq \sum_{k=R}^{2R-2} \sum_{i+j=k} 2^{-p(i+j)+i+j} |x_0 y_0|\,; \\
&\leq |x_0 y_0| \sum_{k=R}^{2R-2} (2R-1-k) 2^{-(p-1)k}; \\
&\leq |x_0 y_0|\, 2^{-(p-1)R} \sum_{k'=0}^{R-2} (R-1-k') 2^{-(p-1)k'}.
\end{aligned}$$

We now consider the function $\phi(\alpha) = \sum_{k=0}^{\infty} (R-1-k)\alpha^k$ and we get:

$$\begin{aligned}
\phi(\alpha) &= \sum_{k=0}^{\infty} -k\alpha^k + \sum_{k=0}^{\infty} (R-1)\alpha^k; \\
&= -\alpha \frac{d}{d\alpha} \left(\sum_{k=1}^{\infty} \alpha^k \right) + (R-1) \frac{1}{1-\alpha}; \\
&= \frac{-\alpha}{(1-\alpha)^2} + \frac{R-1}{1-\alpha}.
\end{aligned}$$

Using $\phi(2^{-(p-1)})$, we obtain:

$$\sum_{k=R}^{2R-2}\sum_{i+j=k} x_i y_j \leq |x_0 y_0|\, 2^{-(p-1)R} \cdot \cdot \left(\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2} + \frac{R-1}{1-2^{-(p-1)}}\right),$$

and since $\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2}$ is negative and very small we conclude our proof by:

$$\sum_{k=R}^{2R-2}\sum_{i+j=k} x_i y_j \leq |x_0 y_0|\, 2^{-(p-1)R}\frac{R-1}{1-2^{-(p-1)}}. \quad (5)$$

■

Proposition IV.2. *(Final error bound) Let x and y, two* ulp-nonoverlapping *FP expansions, with R terms. When computing π, an approximation of xy to R terms, as shown in Algorithm 2 and Fig. 6, the result satisfies:*

$$\begin{aligned}|xy-\pi| \leq\ & |x_0y_0|2^{-(p-1)R}(R-1)\\ & \cdot\Big[1+2^{p-2}(1+2^{-p})+\\ & +(R^3-R)((R-1)!)^2\Big].\end{aligned} \quad (6)$$

Proof: From the definition we know that $|x_i| \leq 2^{-i(p-1)}|x_0|$ and $|y_j| \leq 2^{-j(p-1)}|y_0|$, so we can deduce

$$|x_i y_j| \leq 2^{-(p-1)(i+j)}|x_0 y_0|.$$

For computing π_0 we use only *2MultFMA(x_0, y_0)*, and we get $|\pi_0| \leq |x_0y_0|(1+2^{-p})$.

For computing π_1 we use *VecSum*#1, with 3 entries of order $\mathcal{O}(\varepsilon\Lambda)$: the error from the previous step, and two partial products, which are less than $2^{-(p-1)}|x_0y_0|(1+2^{-p})$. It is easily seen that all these entries are bounded by the same value. We define the following notation:

$$\Omega_1 = 2^{-(p-1)}|x_0y_0|(1+2^{-p}). \quad (7)$$

It follows that $\pi_1 < 3\cdot 2^{-(p-1)}|x_0y_0|(1+2^{-p})^3$ and the outputted errors are less than $\Omega_2 = 3\cdot 2^{-2p+1}|x_0y_0|(1+2^{-p})^3$.

For computing π_2 we use *VecSum*#2 which is going to have 7 entries of order $\mathcal{O}(\varepsilon^2\Lambda)$: 2 errors outputted by *VecSum*#1, bounded by Ω_2; 2 errors from the previous step's partial products, which are less than $2^{-2p+1}|x_0y_0|(1+2^{-p})$; and 3 partial products, less than $2^{-2(p-1)}|x_0y_0|(1+2^{-p})$. We observe once more that all the entries are less than Ω_2.

For the induction step we consider *VecSum*#$i-1$. For computing π_{i-1} we have $(i-1)^2+i$ entries, which we assume are all less than Ω_{i-1}. It follows:

$$\begin{aligned}\pi_{i-1} &< (2\Omega_{i-1}(1+2^{-p})+\Omega_{i-1})(1+2^{-p})+\ldots\\ &< (i^2-i+1)\Omega_{i-1}(1+2^{-p})^{i^2-i}\end{aligned}$$

and also, the largest error term outputted and implicitly all the others are less than $\frac{1}{2}\,\mathrm{ulp}(\pi_{i-1})$.

This implies that all error terms are less than:

$$\begin{aligned}&= (i^2-i+1)\Omega_{i-1}(1+2^{-p})^{i^2-i}\cdot 2^{-p};\\ &= 2^{-(p-1)}|x_0y_0|(1+2^{-p})\cdot\\ &\quad\cdot\prod_{n=1}^{i}(n^2-n+1)(1+2^{-p})^{n^2-n}2^{-p};\\ &= 2^{-(p-1)-ip}|x_0y_0|\overbrace{(1+2^{-p})^{1+2+\ldots+(i^2-i)}}^{<2}\cdot\\ &\quad\prod_{n=1}^{i}(n^2-n+1);\\ &< 2^{-(i+1)p+2}|x_0y_0|(i!)^2. \end{aligned} \quad (8)$$

and this last value will define Ω_i.

This implies that, since we use only simple summation for computing π_{R-1}, in the last step we neglect R^2+R terms, all less than Ω_R.

We also have to account for the errors that occur when computing the last partial products using only simple multiplication. This means $R-1$ terms less than $2^{-R(p-1)+p-2}|x_0y_0|(1+2^{-p})$

When adding all these errors with the one given in (5) we get the following bound:

$$\begin{aligned}|xy-\pi| \leq\ & |x_0y_0|2^{-(p-1)R}(R-1)\\ & \cdot\Big[\underbrace{\frac{1}{1-2^{-(p-1)}}}_{<1}+2^{p-2}(1+2^{-p})+\\ & +\underbrace{2^{-p-R+2}}_{<1}(R^2+R)(R-1)((R-1)!)^2\Big]\\ \leq\ & |x_0y_0|2^{-(p-1)R}(R-1)\\ & \cdot\Big[1+2^{p-2}(1+2^{-p})+\\ & +(R^3-R)((R-1)!)^2\Big].\end{aligned} \quad (9)$$

And this concludes our proof ■

Unfortunately, for the multiplication algorithm we are unable to prove any constraints on the terms of the result. Even though cancellation cannot happen when multiplying two FP numbers, it may happen during the summation process, in which case we can get $|\pi_i| < |\pi_j|$, with $i < j$. If this happens we can apply a *re-normalization* algorithm, like the one presented in [11], in order to render the result non-overlapping. Since this implies adding a sequential step at the end of Algorithm 2, slowing it's performance, we recommend using this algorithm only if computations are known not to be cancellation-prone or if the result can be verified a-posteriori.

V. Warp-synchronous GPU implementation

GPUs are highly multi-threaded SIMD architectures [12]. They are programmed in languages like CUDA and OpenCL, that expose a pure multi-thread programming model. Programmers describe compute kernels as a single program run by

many fine-grained threads. The compiler and hardware group these threads into so-called warps, containing 32 threads on current Nvidia architectures. Threads inside a warp run in lockstep and share a single control flow, and their instructions are executed on SIMD units, with one thread per lane.

This implicit SIMD model is equivalent to explicit SIMD: a GPU program can be also understood as computations on vectors from the point of view of a warp. This enables direct implementation of the data-parallel algorithms we propose. Recent additions to the available hardware primitives make this *warp-synchronous* programming style particularly efficient [10]. Our implementation targets GPUs with compute capability 3.0 or above, such as Kepler and Maxwell architectures, that support *warp vote* and *shuffle* instructions. The code was written in CUDA C, using *double*-precision numbers (i.e. $p = 53$).

Warp vote instructions perform boolean reductions across all threads within a warp. For instance, they can check whether a condition holds for all the threads, or for any of the threads of the warp. The `__any` function computes the logical OR of a warp-sized vector of predicates and broadcast it to all elements. We use it to implement the loop exit conditions of the FP expansion multiplication in Algorithm 2.

Warp shuffle instructions allow arbitrary communication between threads in a warp, without having to go through memory. They are analogous to shuffle or permute instruction in explicit SIMD instruction sets [9]. We use them to shift vector components to propagate the errors across expansion terms, and to insert and extract scalar values inside vectors.

- `shfl_up(x, n, R)` and `shfl_down` shift components respectively upward or downward by n positions within each R-element vector;
- `shfl` reads an arbitrary vector component within each vector lane.

As mentioned in Section III, similar shuffle and permute operations are available in explicit SIMD instruction sets such as Intel AVX. GPUs that do not support the shuffle instruction can also exchange data through the OpenCL local memory at a cost in performance.

We illustrate the warp-synchronous implementation of Algorithm 2 for FP expansion multiplication in Fig. 7. The code appears from a single thread's perspective, but it runs in parallel and it takes decisions based on the vector lane within an expansion (i.e. *threadIdx.x*). Although the hardware only support shuffling 32-bit data, we implemented shuffle instructions on the `double` type by shuffling each half separately.

Although we present here a version of the code that is parameterized by only one parameter, R, our actual implementation uses different template parameters for inputs and output, meaning that we allow static generation of any input-output precision combinations.

We exploit both the parallelism that exist between expansion terms and across different expansions. To benefit from the SIMD execution and intra-warp communication primitives, all terms in a given expansion have to be computed by

```
template<int R>
__device__ double parallelMul(double x, double y){
int lane = threadIdx.x;  // Index within expansion
double s = 0., r = 0., y_i, p, s, e, ep;
for(int i=0; i<R-1; i++) {
  y_i = shfl(y, i, R);  // Broadcast y_i
  p = TwoProdFMA(x, y_i, &e);
  s = TwoSum(s, p, &ep);

  double tmp = shfl(s, 0, R);
  if(lane == i) r = tmp; // Save s_0 to r_i
  s = shfl_down(s, 1);   // Shift left
  if(lane == K-1) s = 0.;

  while(__any(e != 0.)) { // Accumulate e
    s = TwoSum(s, e, &e);
    e = shfl_up(e, 1, R); // Shift right
    if(lane == 0) e = 0.;
  }
  while(__any(ep != 0.)) { // Accumulate e'
    s = TwoSum(s, ep, &ep);
    ep = shfl_up(ep, 1, R); // Shift right
    if(lane == 0) ep = 0.;
  }
}
y_i = shfl(y, R-1, R);
p = x * y_i;
s = s + p;
double tmp = shfl(s, 0, R); // save r_{R-1}
if (lane == R-1) r = tmp;
return r;
}
```

Fig. 7: Multiplication Algorithm 2 implemented in CUDA C, simplified for the case in which the inputs and the output have the same power-of-two size, R.

threads of the same warp. As warps have 32 threads on Nvidia architectures, the maximal supported expansion size is 32. Smaller expansions are packed together inside warps. Although this approach works with any expansion size R between 1 and 32 using appropriate padding, we recommend using power of two sizes, which allow filling the whole warp.

VI. Comparison and discussion

In this section we present performance measurements obtained on a Tesla K40 GPU (Kepler architecture), using the CUDA 7.5 software architecture. The code also runs on the newer Maxwell architecture, although the performance of all algorithms becomes limited by the lower double-precision performance of current Maxwell-based GPUs. We measure throughput on embarrassingly-parallel computations. The values are given in million of operations per second (Mop/s). By one op we understand one operation using extended precision. The tests were done using random generated examples, running on 1024 blocks each with 1024, 512 or 256 execution threads, depending on the expansion size and the required resources to run the algorithms.

To analyze the effect of parallelism on memory footprint, we consider two different shared memory usage scenarios: one best case that assumes the application uses no intermediate data outside of the registers used for the computation, and one worst case where the application uses 256 bytes of CUDA shared memory for each term of the expansion.

For comparison, we report to Bailey's GQD library [8] and CAMPARY [11], which offer extended precision using FP

expansions on GPU. The arithmetic algorithms they employ are not by themselves parallel. Moreover, the GQD library is limited to double-double and quad-double precisions, and the algorithms used in the implementation are not provided with any correctness proof and there is no specific error bound given. CAMPARY is a recent software, that comes with correctness proofs and results guaranteed within a certain error bound.

In Table I we show the addition algorithm's performance for the best-case, no-memory configuration, followed by the performance obtained in the memory-constrained configuration, in Tables II.

TABLE I: Performance in Mop/s for adding two FP expansions on the GPU in the best case with no internal memory usage; R represent the number of terms in both input and output expansions. $*$ precision not supported

R	Safe (Alg. 1)	Fast	CAMPARY	QD
2	6,294	11,914	3,188	41,549
4	1,141	4,637	1,448	5,310
8	314.4	2,234	493.6	$*$
16	73.89	830.3	158.6	$*$
32	13.57	131.4	41.77	$*$

TABLE II: Performance in Mop/s for FP expansion addition algorithms in the memory-constrained case with 256B shared memory per expansion term

R	Safe (Alg. 1)	Fast	CAMPARY	QD
2	1,774	1,424	971.9	4,830
4	337	528	131.3	391.4
8	70.52	229.4	28.2	$*$
16	16.67	103.9	6.52	$*$
32	3.82	16.03	1.04	$*$

Even in the worst-case embarrassingly-parallel setup, the performance of data-parallel addition algorithm is competitive with the best known sequential algorithms for smaller expansions: the parallelism comes at little cost in number of operations per expansion.

For the multiplications algorithm we asses performance in Tables III and IV, for the best case setting and for the memory constrained one, respectively.

TABLE III: Performance in Mop/s for multiplying two FP expansions on the GPU in the best case with no internal memory usage

R	Algorithm 2	CAMPARY	QD
2	6,484	11,100	27,390
4	756.2	1,363	2,726
8	135.2	47.19	$*$
16	25.07	12.10	$*$
32	2.63	2.62	$*$

TABLE IV: Performance in Mop/s for FP expansion multiplication algorithms in the memory-constrained case with 256B shared memory per expansion term

R	Algorithm 2	CAMPARY	QD
2	882.87	2,093	3,501
4	99.37	180.8	250.9
8	18.05	7.06	$*$
16	4.04	0.84	$*$
32	0.23	0.096	$*$

From the data obtained in the best case scenario we observe that addition on larger expansions and multiplications suffer from parallelization overhead. The benefits of exploiting the parallelism available within each expansion are fully realized when parallelism is constrained by internal memory usage. The performance of data-parallel algorithms remains stable in this setup, while the performance of sequential algorithm decreases sharply with memory usage. Although the QD library remains faster on expansions of size 2 (double-double), the data-parallel algorithms significantly outperform their sequential counterparts for all larger expansions. The performance gap increases with the expansion size, eventually reaching an order of magnitude for 32-term expansions.

VII. Conclusion

We presented data-parallel algorithms for addition and multiplication of floating-point expansions. By taking advantage of data parallelism within FP expansions, they are suitable for SIMD architectures. We present fast addition and multiplication algorithms, as well as a safe addition algorithm with rigorous error bounds. A GPU implementation of the algorithms using warp-synchronous programming in CUDA is already competitive with state-of-the-art sequential algorithm in an idealistic embarrassingly parallel setup. However, data-parallel algorithms really shine in the more realistic case of applications that manipulate a sizable amount of intermediate data, significantly outperforming sequential algorithms for expansions of size 4 and greater.

References

[1] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic," in *Arith-10*, Jun. 1991, pp. 132–144.

[2] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," *Discrete Computational Geometry*, vol. 18, pp. 305–363, 1997.

[3] M. Joldes, V. Popescu, and W. Tucker, "Searching for sinks for the hénon map using a multipleprecision gpu arithmetic library," *SIGARCH Comput. Archit. News*, vol. 42, no. 4, pp. 63–68, Dec. 2014.

[4] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[5] D. M. Priest, "On properties of floating-point arithmetics: Numerical stability and the cost of accurate computations," Ph.D. dissertation, University of California at Berkeley, 1992.

[6] S. M. Rump, T. Ogita, and S. Oishi, "Accurate floating-point summation part I: Faithful rounding," *SIAM Journal on Scientific Computing*, vol. 31, no. 1, pp. 189–224, 2008.

[7] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM Journal on Scientific Computing*, vol. 26, no. 6, pp. 1955–1988, 2005.

[8] Y. Hida, X. S. Li, and D. H. Bailey, "Algorithms for quad-double precision floating-point arithmetic," in *ARITH-16*, Jun. 2001, pp. 155–162.

[9] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel AVX: New frontiers in performance improvements and energy efficiency," Intel white paper, Tech. Rep., 2008.

[10] NVIDIA, "Kepler GK110 architecture," NVIDIA Whitepaper, Tech. Rep., 2012.

[11] M. Joldes, O. Marty, J.-M. Muller, and V. Popescu, "Arithmetic algorithms for extended precision using floating-point expansions," *IEEE Transactions on Computers*, vol. PP, no. 99, p. 1, 2015.

[12] M. Garland and D. B. Kirk, "Understanding throughput-oriented architectures," *Communications of the ACM*, vol. 53, no. 11, pp. 58–66, 2010.