ASL PROJECT: FAST CODE FOR ADDITIONS AND MULTIPLICATIONS OF FLOATING-POINT EXPANSIONS

András Geiszl, Alec Pauli, Luca Pinter, Tom Wartmann

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

In this paper, we present optimized versions of the addition, multiplication, and truncated multiplication algorithms of floating point expansions. In today's world, more and more high-precision computations are done every day. Most of this high-precision computation exceeds the precision of doubles. As these implementations get larger and more common the need also arises to have optimized algorithms for use on the CPU. We propose several optimized versions of the above-mentioned algorithms that are based on code rewriting, vectorization, and other common optimization techniques. These implementations were tested using different compilers, processors, and systems, and all significantly outperform in runtime the implementations proposed in libraries, such as the Campary Library. Our paper lays a solid basis for future software packages that want to support floating point expansion on the CPU with the maxmimum possible performance.

1. INTRODUCTION

In this section we start with the motivation, then explain what our contribution to the research was, and finally list some related work.

Motivation. Machine precision, also known as floating-point precision, is a fundamental concept in computational science and numerical analysis and refers to the inherent limitation in representing real numbers on a computer system. While the standard double-precision is certainly sufficient for the normal user, there are very important areas that require higher precision such as calculating the long-term stability of the solar system. To achieve a higher precision than the standard double-precision there are several possibilities, one of which is the multiple-term representation. This is where each number is represented as an unevaluated sum of several standard floating-point numbers, so-called *floating-point expansion* (FP expansion), allowing for more significant digits to be preserved during calculations. The algorithms of the standard arithmetic operations such as addition, multiplication, division, and so on must be adapted to this representation of numbers, which is what the two papers [1] and [2] dealt with. Among other things, they presented an algorithm for addition, as well as two algorithms for multiplications of floating-point expansions and their correctness proof, error analysis, and complexity analysis. For the use of these algorithms in the real world, high performance is very important to reduce computational costs.

Contribution. We present implementations with optimized performance for the addition and multiplication algorithms from [1] and the truncated multiplication algorithm from [2]. For the addition and the truncated multiplication, we achieve a higher performance for all input sizes than the comparable implementations from CAMPARY. While the basic instructions such as multiplication and addition can be used fairly directly in the respective use cases, there are a variety of algorithms that perform multiple multiplications independently and in parallel. For such use cases, we wanted to make the resulting additional parallelization possibilities as well as pipelining possibilities usable in a function that is as simple and intuitive as possible. For this purpose, we created the function fourmult. This function takes four expansions of doubles, i.e. the same expansions as a multiplication function. To increase the parallelizability this function was limited to take doubles of the same precision length or to output the same precision. We deliberately chose the dimension of 4 basic inputs, because this allows all additional potentials while being the smallest possible unit. This means that larger independent multiplications can be composed of additional blocks of four.

Related work. As a direct reference for most implementations, we used the Campary library (CudA Multiple Precision ARithmetic librarY), available at `https://homepages.laas.fr/mmjoldes/campary/` . The Campary Library makes use of the principle of unevaluated double sum in order to represent high-precision numbers. Thus, the hardware implementation of computing doubles can be used. The main downside of the Campary library is the fact that it's optimized for the CUDA architecture, thus, for GPUs. Despite a CPU version of Campary being available these

implementations are not particularly fine-tuned. While GPUs are often available high precision arithmetic is also used in scenarios where we only rely on CPUs. One scenario comes up in cryptographic processes or in sensoric data that needs to be pre-processed in low-end hardware. But many other scenarios are possible. Our implementations focus mainly on decreasing the running time on CPU-based systems.

2. BACKGROUND ON THE ALGORITHM/APPLICATION

First, we would like to give a brief introduction to the concept of non-overlapping when storing FP expansions. It is important that the terms of an FP expansion do not overlap much, otherwise, it would be very memory inefficient. A simple example would be to store the number 4.2 as the unevaluated sum of 1.0, 2.0, and 1.2, this is why there are overlapping restrictions for example *ulp-nonoverlapping*, defined in the following definition.

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

Where ulp() stands for *unit in the last place* and is basically just the part of the exponent of a floating-point number without the mantissa. This means that for an FP expansion that is *ulp-nonoverlapping*, the largest number is at the first place, and each subsequent number overlaps with the previous number by at most one bit.

Another important concept in floating-point arithmetic is the accurate calculation of the sum of two numbers, where in our case, two different algorithms were used, the *TwoSum* and the *Fast2Sum*. A clear comparison and complete analysis of the two algorithms can be found in Muller et al. [3]. We list here again the two algorithms for clarity in Algorithm 1 and 2, which calculate the exact sum of *a* and *b* and return the sum and the overflow in *s* and *e* respectively, whereas RN stands for rounding to the next mode. As you can see in Algorithm 1, the *TwoSum* function first calculates the sum in *s* and then the overflow in *e*, taking into account the two cases where either *a* or *b* is greater.

Algorithm 1 *TwoSum* (a, b)

Input: a, b
$s \leftarrow RN(a + b)$
$t \leftarrow RN(s - b)$
$e \leftarrow RN(RN(a - t) + RN(b - RN(s - t)))$
return s, e

On the other hand, *Fast2Sum* (Algo. 2) has the constraint that $|a| \geq |b|$ and therefore needs much less operations to calculate the exact *e*.

Algorithm 2 *Fast2Sum* (a, b)

Input: a, b
$s \leftarrow RN(a + b)$
$t \leftarrow RN(s - b)$
$e \leftarrow RN(b - t)$
return s, e

Addition. The addition algorithm receives as an input two floating point expansions as well as a parameter indicating the output size. These floating-point expansions could potentially be of different sizes. An additional note can be made on the parameter r as its size is certainly limited by the size of the sum of the two input expansions. The output of the addition algorithm consists of a floating-point expansion with r elements that represent the addition. This is achieved by first merging the two input elements and then normalizing the resulting output from the merging step. As the merging step often introduces additional overlapping of the representation even a significantly smaller normalized result could potentially lead to no loss in precision. Even though loss in precision is of course always possible.

Multiplication. The algorithm calculates the k most significant components of the product $r = a \cdot b$. Opposing to the truncated multiplication this algorithm computes all multiplication terms and adds them up afterwards. Thus, the limiting factor is the precision. In the algorithm, an additional error correction term is used in the renormalization. An intuitive explanation of the algorithm is as follows. It uses a "k input - k output" variant and an "error-free transformation scheme" k+1 times. The products are added according to their magnitude and the remaining terms are used in the next iteration. The (k + 1)-th component rk is obtained by simply summing all remaining errors with the simple products of order $O(\epsilon k \Lambda)$

Truncated Multiplication. This algorithm receives as input two ulp-nonoverlapping FP expansions and returns as output one ulp-nonoverlapping expansion which is the product of the two input numbers. At a high level, this multiplication is done in a pen-and-paper approach, so each number of one expansion is multiplied by each number of the other expansion and then all partial products are summed up together in bins. In this process, all terms with a precision smaller than the one of the output are discarded, which is where the name truncated multiplication for this algorithm comes from. In the end, the result is normalized such that it is ulp-nonoverlapping again. By initializing the bins with a start value and by the restriction that the input is ulp-nonoverlapping, it is ensured that for each addition of two numbers inside the algorithm, the prerequisite for using the *Fast2Sum* is fulfilled, which is why this algorithm makes exclusive use of this faster version of the addition.

Cost Analysis. We are computing the number of flops

based on the number of floating point operations that are necessary for a mathematical implementation of the algorithm. In particular, we are ignoring index computations or memory creation movements, etc. This is because as mentioned previously the allocation of memory isn't counted also a zeroing of memory wouldn't be needed in a mathematical algorithm. Also for example in an assignement $f[i] \mathrel{+}= f[i+5]$ the flop computing $i+5$ wouldn't be needed in a math formula and is therefore omitted. The only counted flops in such an implementation are the $ftmp[i] + ftmp[i+5]$ ones as they would be needed in the mathematical representation. All computations are directly computed on the fly with implementations of the algorithm that are simply instead of computing the result computing the flop count. These algorithms are located in the benchmark.cpp file of our implementation. The fact that we are ignoring memory movement computations and other computations that are not needed in a mathematical representation drags our measured flops/cycle down for example for memory movement the processor is certainly doing work but in the flops, it's not counted. But overall it makes the performance more comparable across different implementations that possibly use less memory/memory movement.

Cost of Truncated Multiplication. We mentioned that this algorithm uses the stronger precondition of the input floating-point expansion being ulp-nonoverlapping, this allows the use of *Fast2Sum* instead of the regular *TwoSum* in the calculation. Also, since the result is truncated, meaning the output is of a fixed size, we can discard all contributions of the calculation that are too small.
Assuming both the input expansions, as well as the output expansion have k terms, then a total of

$$\frac{9}{2}k^2 + 11k + 5\left\lfloor \frac{k-53}{45} \right\rfloor + 13$$

floating-point operations are performed during the calculation.

Profiling. We used the GNU Profiler to discover parts of our functions with high runtime. For example for the *addition* implementation we have seen that the most time was spent on the function twoSum (61%), followed by vecSumErr (11%), merge (1.5%) and vecSumErrBranch(1.4%). It showed in all functions that the *twoSum* or *Fast2Sum* function is the bottleneck and since they are mostly dependent on a previous call the optimization possibilities are strongly limited (see some of the attempted workarounds of this problem in section 3).

3. YOUR PROPOSED METHOD

In this section we explain the optimizations we have applied to each function and the rationale behind them. It is divided into three paragraphs, one for each main function.

Addition. To optimize *Addition*, we first optimized the two sub-functions it consisted of: *Merge*, and *Renormalization*. Since the *Merge* algorithm only consist of taking absolute values, a few comparisons, and memory operations, its performance could not be improved significantly. We nevertheless eliminated the absolute functions in favor of using ternary operators, and took out an integer addition that was done multiple times in it. We also performed an (informal) analysis on parallelizing the merge algorithm: In theory, we could compare one element from *a* with the next 4 (in case of AVX2) elements of *b*, then, if all of them are larger than the single element from *a*, we can move them to the result array at once. In the best case, this would result in 1 comparison instead of 4. However, this would only be beneficial if most of the elements in one input is larger than those in the other inputs. If we don't assume anything about the elements of the input arrays, most of the times this case would not happen and instead we would incur performance penalties from the overhead of setting up the vector operations. Therefore, we decided against this optimization.

Optimizing *Renormalization* was significantly more complex and resulted in much greater performance improvements. To speed up *Renormalization*, we first optimized the sub-functions it consisted of: *VecSum*, *VecSumErrBranch* and *VecSumErr*. Similarly to *Merge*, *VecSumErrBranch* and *VecSumErr* include mostly comparisons and memory operations. Unlike *Merge*, they also contain a loop of *TwoSum* calls, which include a series of floating point operations. However, all of these operations depend on the previous operation's result and the error in each *TwoSum* depends on the error in the *TwoSum* in the previous iteration of the loop (because the error is accumulated across iterations). Therefore, they could not be parallelized. Overall, the only optimization made in these two functions are eliminating the external function calls, using a single double-precision value for the running error instead of an array, saving an integer operation in a variable, and changing an *if* inside *else* to an *else if* in *VecSumErrBranch*. These changes alone did not result in a significant performance improvement.

Optimizing *VecSum*, on the other hand, offered substantially more opportunities of speedup. First, we took advantage of the fact that we did not need a sum after the next one has been calculated, and changed its array to a single double-precision variable. Next, we tried vectorizing the loop. Similarly to *VecSumErrBranch* and *VecSumErr*, in *VecSum* we have many dependent instructions. In particular, here, the sum in one *TwoSum* call is dependent on the sum in the previous call of *TwoSum*. Keeping this in mind, we tried two strategies of vectorization: In the first one, we would perform the dependent parts of the computation (the sums) of 4 *TwoSum*s, then perform the rest (error calculation) in parallel, repeating this for all *TwoSum*s. In the second strategy, we would perform all the dependent calculations first in

a separate loop, then we were free to calculate as many errors in parallel as it was necessary. Both strategies required an additional loop for computing the rest of the *TwoSum*s when the number of iterations was not divisible by 4. Interestingly, our performance benchmarks shown that each of these strategies are better for different input sizes. Based on experimental results on one machine, we chose to use the first strategy when the input size was less than 20 and the second one otherwise. Additionally, we could further improve performance for larger input sizes by taking advantage of instruction-level parallelism and calculating 8 errors at once instead of 4 (by creating multiple accumulators variables and unrolling the loop). According to our experiments, this gave better speedup than the second strategy for input sizes larger than 33, therefore we chose to use it in this range. At the end, we also confirmed these results on other machines and they showed a similar range of input sizes where each of the mentioned strategies were advantageous.

After optimizing the main components of *Renormalization* we put them together in one function. To save memory we reused one temporary array for all of the sub-functions. We inlined all of them into one function to save multiple function calls and used comparisons of input sizes with the above mentioned range limits to execute the right *VecSum* implementation for the current input size. We also simplified the first or last iteration of the loops in the sub-functions where it was possible (for example, the first iteration of *VecSumErrBranch* can be simplified, since we know that the initial value of the error is 0). The loop conditions for *VecSumErrBranch* and *VecSumErr* could also be changed depending on which one of the input or output sizes are smaller (we can just stop if the result is smaller than the output, then fill the rest with zeros).

Additionally to the optimizations above, we also vectorized the loop of *VecSumErr* calls at the end of *Renormalization*. This was a challenge, since generally these *VecSumErr* calls depend on each other's results. However, one can notice that, to calculate one specific element of one iteration of the loop of *VecSumErr* calls, one only needs the error up until that element, and the result of the sum of elements up until the next element of the previous iteration. For example, to calculate $f_3^{(1)}$ (where the upper index is the index of the iteration, and the lower index is the index of the element in that iteration's result array) we need the error from $f_1^{(1)} + f_2^{(1)}$ and the result of $e^{(0)} + f_4^{(0)}$ (where $e^{(0)}$ is the running error in iteration 0). Therefore, we only need to calculate the sums until element 4 in the previous iteration to start calculating the results of the current iteration. Using this insight we can calculate elements of 4 iterations in parallel using AVX2 vectors. The result of these computations will be an array similar to the input array (but shorter) that can be used to start 4 more iterations of calculations. This vectorization strategy, however, cannot fully utilize the AVX2 vectors, since not all 4 elements of each vector will be available from the start, but will be later obtained from the result of summing the first few elements. This means that for smaller input sizes this optimization reduces the performance. From our experiments we saw that, if the input size is 20 or more, this strategy results in a shorter runtime. Below that, we decided to use the unvectorized implementation.

To obtain the final, fully optimized, implementation of *Addition*, we put together the improved *Merge* and *Renormalization* implementations and used the input size limits discussed above to compare to, so that we execute the vectorized and unvectorized implementations appropriately.

After optimizing the *Addition* algorithm given in the paper, we also replaced *TwoSum* with *Fast2Sum* wherever possible to reduce the number of instructions. Since the number of floating point operations is reduced, this resulted in a better runtime, but a worse performance (defined as $flops/runtime$). This is described in more detail in section 4.

Multiplication. For the multiplication we created two different incrementally optimized versions. Furthermore, we created a third version, which made use of the FastSum computation. The first optimized version uses the previously described subfunctions directly to speed up the code. Thus, with this version we can relatively directly assess to what extent our optimized subfunctions generate a performance benefit. Since most of the running time takes place in the renormalization of an already optimized subfunction, a clear performance benefit was measurable, as expected. In the multiplication3 variant, increased attention was now also paid to optimizing the code primarily in a version without vector instrincts. For this, all subfunctions except for the renormalization direct were inlined, which made certain further optimizations possible and gave us a better overview of possible optimizations. We also performed all optimizations such as code motion and strength reductions, which will most likely be taken over by the compiler, but which we nevertheless wanted to implement already, since we want to be as little dependent as possible on external factors. Finally, the code was vectorized where possible. These optimizations in the multiplication3 variant led to a reduction of about 10-15 percent (measured on an Intel core i7-6700k) of the total cycles. This is encouraging but significantly less than the jump from the reoptimized version to our multiplication2 version. This can be seen particularly well in our measurements of performance in flops/cylces. Our latest version, multiplication3, now also integrates renormalization directly into the code. Here again further parts like the VecSumErr part are vectorized. Furthermore as in the previous versions the Two Sum fast is used.

Truncated Multiplication. In this function, we have implemented basic optimizations such as pre-computations

and the elimination of complex functions and function calls. Further we compared our implementations with the *truncatedMul* function of the CAMPARY package. We have not applied vectorization, because of many read-after-write dependencies, nor block optimization, because the maximum size of the input and working set, at double precision, are three expansions of length 29, which fits into cache. We have chosen a maximum input and output size of 29 for this function, since each number of a double-precision floating point expansion that is ulp-nonoverlapping covers a range in the exponent of 53, and the highest exponent is +511 to prevent overflow during multiplication. This results in $\lceil (511 - (-1022)/53) \rceil = 29$ for an upper bound in the number of terms for the input of this multiplication.

The *Baseline* is a naive implementation of Algorithm 1 from [2] where in principle the pseudo code was translated exactly into C code. During the first optimization (*Opt: 0*), repetitive calculations were pre-calculated, such as repeatedly accessing the exponent of numbers or calculating constants to initialize start values. In the next optimization (*Opt: 1*), complex functions were additionally replaced with simpler ones. For example, the rounding function was replaced by a cast to an integer and the divisions by a constant with the multiplication of their reciprocal value. In the optimization (*Opt: 2*), in addition to the previously mentioned optimizations, all function calls were eliminated by inlining, to provide opportunities to the compiler for further optimizations. In optimization 3 (*Opt: 3*), the main loop that iterates over the two input arrays was additionally unrolled twice and the intermediate results were stored in separate variables.

4. EXPERIMENTAL RESULTS

We start this section by introducing our experimental setup and then presenting the results of our optimizations in three separate paragraphs.

Experimental setup. The plots shown here were created on Windows 11 (build 22621.1926) with the MSVC CL compiler (version 19.36.32534). We used the following flags: /std:c++17 /EHsc /O2 /arch:AVX2 /GL. This optimizes for speed (O2) and uses global optimizations (GL), but does not enable fast floating point operations, which could make some of our floating point calculations incorrect. It also does not vectorize code automatically and does not generate FMA instructions (we manually use these features in our code when appropriate). We executed the benchmarks using an AMD Ryzen 7 4800H mobile processor. Furthermore, all results were compared on Linux (version : Ubuntu 22.04.1 LTS) using the GCC compiler (version : 11.3.0) with the flags: -O3 -fno-tree-vectorize -ffp-contract=off -march=native -std=c++17. The processor here was an Intel Core i7-6700k. These comparisons can be found on our GitLab repo. But the differences are not very significant in general.
Additionally, since we did not use any hardware specific optimizations in the *truncated Multiplication*, the benchmarks for it were also run on an Apple M1 processor. The programs were compiled on macOS Ventura 13.4 also using the GCC compiler. Because of the hardware change, the compiler flags had to be adjusted to -std=c++17 -O3 -fno-tree-vectorize -framework Accelerate.

Measurements for the benchmarks were collected using the *QueryPerformanceCounter* function on Windows, the *rdtsc* CPU instruction on Linux, and the *chrono::high_resolution_clock* on macOS. Each function was measured as many times as it took to measure at least 100 million cycles, then the measurement was divided by the number of runs. This process was repeated 50 times and the final runtime was taken as the median of those values.

Each main function and the sub-functions of *Addition* and *Multiplication* were measured separately on different input sizes and their performance and runtime were plotted against them. The maximum value for the input size of a function depends on the requirements of overlap of that function. There are three different requirements of overlap: ulp-nonoverlapping (Definition: 1), at most d-digit overlapping (Definition 3.1 in [1]), and S-nonoverlapping (Definition 2.4 in [1]). Each function was measured with input sizes up until the maximum its overlapping requirements allowed. Where no restrictions were mentioned by the paper, we chose to use the restriction of one of its sub-functions or one of the other functions used in the same function to be consistent and easier to visualize. For example, *Multiplication* uses *Renormalization*, which needs its inputs to be at most d-digit overlapping with $d \leq p-2$ (where p is the precision, so 53 in our case of double-precision). In this case we chose to give both *Multiplication* and *Renormalization* at most d-digit nonoverlapping inputs with $d = 51$. This allows the input expansions to have at most 742 terms so that their product does not exceed the maximum value of a double-precision floating point number. The input sizes to measure with were then chosen to be a geometric sequence between the minimum and maximum, except for *Multiplication* and *Truncated Multiplication*, where using larger input sizes exceeded the maximum stack size, so a lower maximum and an arithmetic sequence were chosen.

Next, we describe in detail the runtime and performance results of our measurements.

Addition. To analyze the performance of the *Addition* function, we will first discuss the performance of its sub-functions. We also report the maximum speedup achieved for any input size for each of these functions.

Since *VecSumErrBranch* and *VecSumErr* did not offer significant opportunities of optimization, their performance were increased by only 4.4 and 4.1 times respectively mostly

due to eliminating function calls, memory reads and writes. By using *Fast2Sum*, we were able to further reduce the runtime for an overall 5.6 and 6.3 speedup, however, this made performance worse. This is due to the fact that replacing *TwoSum* with *Fast2Sum* reduces the number of floating point operations performed from 6 to 3, but the number of memory operations does not change. Therefore, the floating point operations are reduced more than the runtime, so the overall performance decreases. Because the reference implementations by CAMPARY use *Fast2Sum*, their performance is comparable to our implementations using *Fast2Sum*.

As mentioned in the previous section, optimizing *VecSum* was more successful. Due to the vectorization and increasing the instruction-level parallelism, a maximum speedup of 7.2 times was achieved for larger input sizes. Using *Fast2Sum*, we increased the runtime reduction to 9.0 times, which is close to the performance of the reference implementation from CAMPARY. For larger sizes, it is even slightly better. For inputs smaller than 30, however, their implementation is marginally better. This is due to the fact that they are using the C++ templating feature to generate statically optimized code for their implementation for each specific input size we run it for. Since our code is strictly in C, we cannot take advantage of this, and therefore our code will be necessarily less optimal.

Putting the above sub-functions together (including using specific implementations of *VecSum* for specific input sizes), and making additional improvements for the loop condition and special-case iterations of the loop, we get a 5.7 speedup for *Renormalization*. This is in line with the fact that in *Renormalization* the most executed function is *VecSumErr*, so its speedup (4.1x) will be the main determining factor for the speedup of the entire function. Using *Fast2Sum*, the runtime speedup can be increased to 7.5 times (again determined mainly by the speedup of *VecSumErr* using *Fast2Sum*). Introducing the vectorization of the loop of *VecSumErr* calls increases this to 7.5 times (and increases speedups for larger input sizes). Using this vectorized implementation, we achieved an improvement up to 71% even compared to the reference implementation from CAMPARY.

Putting together the optimized implementation of *Renormalization* with the *Merge* implementation, we got a 3.7 times performance increase for *Addition*. The likely reason for lower performance to previous speedups is the *Merge* function and its linearity (as mentioned in section 3). Using *Fast2Sum* and the vectorized loop of *VecSumErr* calls we could increase runtime speedup to 6.1 times, which is 69% better than the reference implementation from CAMPARY. These results are shown on figure 1. Note, that because this is a performance graph, it only reflects the performance comparison between the different implementations. To get an accurate view of the runtime differences, we included the runtime speedup in parenthesis on the legend.

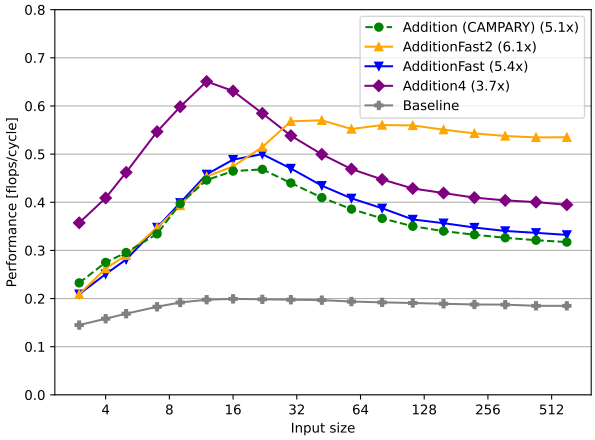

Fig. 1. Performance of our optimizations of the **Addition** Algorithm and its reference function from CAMPARY. The speedup of runtime is shown in parenthesis in the legend.

Multiplication. We can see how all our optimizations perform significantly better than the base implementation with up to a speedup of 4.5x for the Multiplication3. Our optimized versions also increase their performance more and more with larger and larger input sizes. Beginning with an input size of around 15 examples we also outperform the Campary package in flops/cycle. Important to note here are two details. First of all in terms of runtime all optimizations of use are simmilar or better to Campary, especially also the MultiplicationFast. And most importantly the two algorithms aren't identical. We have chosen a generalization of Bailey's multiplication algorithm that simulates the paper-and-pencil method from the campary library as a direct comparison wasn't possinble. This implementation has a higher flop count as it turned out.

Truncated Multiplication. Each optimization increased the performance of the truncated multiplication algorithm step by step as we can see in Figure 3. An exception to this is *Opt: 3*, where (on Intel) the performance decreases significantly due to the unrolling. Across all versions, performance increases with increasing input size, converging to a maximum after an input size of 21. Part of the improvement from the Baseline to the *Opt: 0* can also be explained by a small reduction in the number of flops. The highest increase in performance has *Opt: 1* and *Opt: 2* with 2.1x compared to the Baseline. Comparing *Opt: 2* to the implementation of CAMPARY, one can see that our implementation was about 1.4x faster.

Truncated Multiplication on the Apple M1 processor. The same optimizations done to Truncated Multiplication were also measured on an Apple M1 processor, the results are shown in Figure 4. Contrary to the previous result, the

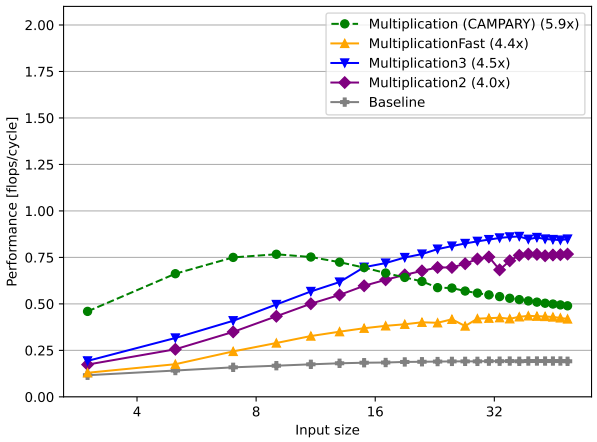

Fig. 2. Performance of our optimizations of the **Multiplication** Algorithm and its closest reference function from CAMPARY. The speedup of runtime is shown in parenthesis in the legend.

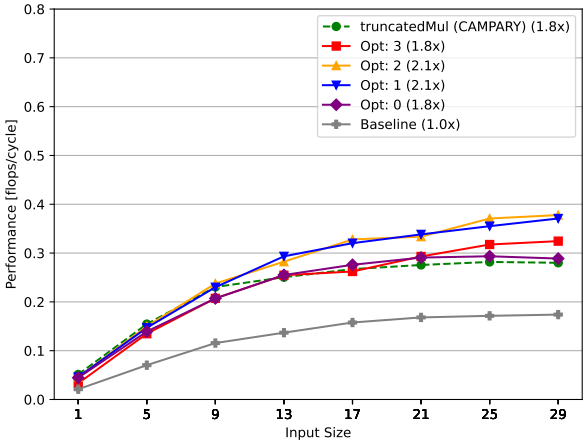

Fig. 3. Performance of our optimizations of the **truncated Multiplication** Algorithm and the *truncatedMul* function from CAMPARY. The operations count are roughly the same. The percentage change of performance compared to the Baseline is shown in parentheses in the legend.

performance does not increase with input size, but instead remain approximately constant. One exception occurs at the smallest input size, where the performance is incredibly high, especially for the CAMPARY algorithm. This is possibly not because the program is highly efficient at low input sizes, but it is rather due to inconsistencies in the clock frequency, leading to inflated performance values, that is why the plots only start at input size 5. Overall, the number of flops per cycle are noticeably higher than on the Intel processor. The best performance is also achieved by *Opt: 2* with a 93% increase compared to the baseline, while *Opt: 3* did also decrease performance, though not as significantly. At higher input sizes, our best optimization achieves about 2x the performance of the implementation of CAMPARY.

Roofline. On figure 5 we can see each algorithm's (and their sub-functions') most optimized implementation plotted on a roofline plot. Their operational intensity is calculated from the minimum memory movements required (loading the inputs, then writing the output at the end). This allows us to see how far the performance of our implementation is from a theoretical maximum (which might be practically unachievable). From this, we can note that the sub-functions are memory-bound and they are close to their theoretical performance limit. The gap between the memory bound and their performance can be explained by the high number of dependencies between the instructions of *TwoSum* and between sequence of *TwoSum*s in the algorithm. Due to these dependencies, many ports of the processor is underutilized most of the time and therefore performance is low.

This is even more apparent from the plots of the main functions, which, even though are in the CPU-bound re-

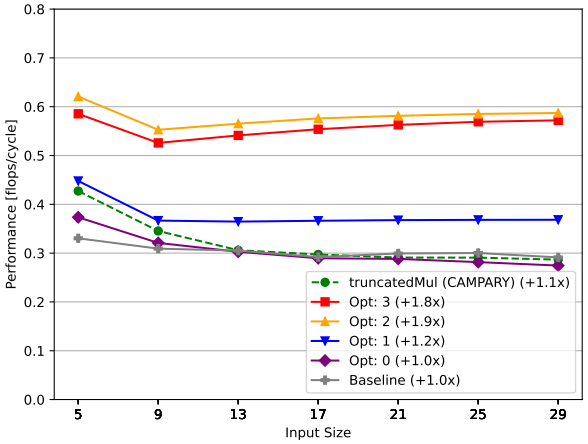

Fig. 4. Performance of our optimizations of the **truncated Multiplication** Algorithm and the *truncatedMul* function from CAMPARY. The operations count are roughly the same. The percentage change of performance compared to the Baseline is shown in parentheses in the legend. *Apple M1 CPU @ 3.2GHz, Flags: -O3 -fno-tree-vectorize -framework Accelerate*

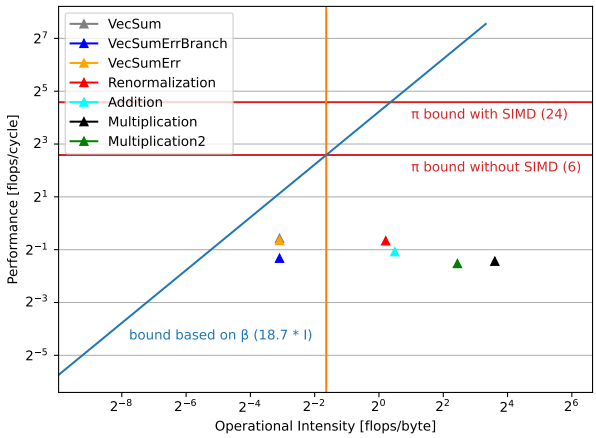

Fig. 5. Roofline plot of the most optimized implementations.

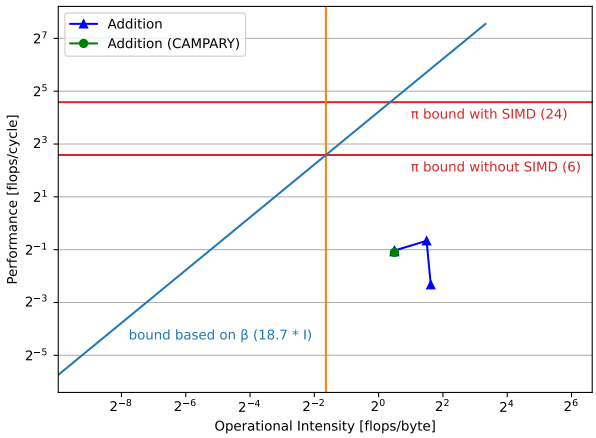

Fig. 6. Roofline plot of the **Addition** algorithm and its reference implementation from CAMPARY.

gion, due to these dependencies and some additional memory operations that are done to compute and store intermediate values, their gap between performance and the bound is even higher. This is also the reason why these functions might not allow much further optimizations and the theoretical peak performance may be practically impossible to achieve using the current algorithm.

On figure 6 the different implementations of *Addition* are shown plotted on a roofline plot. From this, we can clearly see as the performance improved with optimizations, then decreased by replacing *TwoSum* with *Fast2Sum*. We can also see that, by using *Fast2Sum*, we got closer to the memory-bound region as the operational intensity is reduced. This is generally true for the other functions as well (and their roofline plots look similar). Due to initial optimizations, their performance increases, and after changing to *Fast2Sum*, their operational intensity decreases and they become more memory-bound.

5. CONCLUSIONS

In this work we have implemented addition, multiplication and truncated multiplication algorithms for floating point expansions in the programming language C and measured, optimized and compared their performance. We were able to achieve a substantial speedup in all algorithms through various optimization techniques, and we even achieved higher performance than the comparative CAMPARY package. Due to many dependencies within our algorithms, as well as many compare and memory operations, the improvements are not huge and are in the range of 2-6x. Further we did present a new function that can perform 4 multiplications of FP expansions in parallel. We also created a framework to test the correctness of all functions and tested the code on different operating systems (Windows, Linux and Mac) and processors Intel, AMD and M1 and with various compilers. Since this framework is easily adaptable it could be used very well for testing and improving other functions. However, it would be interesting to compare our results with other packages that do the addition/multiplication of FP expansions to get a better understanding of our results.

6. CONTRIBUTIONS OF TEAM MEMBERS

András. Optimized *Addition* (and its sub-functions), including vectorizations, except the *VecSum* vectorizations, which were done by Alec. Also moved out sum calculation in *VecSum* to a separate loop and created the unrolled vectorized version. Wrote code for measuring functions, generating input and benchmarking them and their references on multiple input sizes (including flop calculations). Processed benchmark results, created performance, runtime and roofline plots for *Addition* and *Multiplication*. Created a ”playground” for comparing different implementations of an algorithm for easier optimization.

Alec. Generally, I worked with András mainly together on the parts of the following paper.[1] For that, I created the base implementations of the following functions (all from the paper): twoSum, twoMultFMA, vecSum, renormalization algorithm, addition, multiplication. Further, I created the test framework of these functions and the optimized counterparts. For the optimizations I created the vectorized and inlined version of the multiplication algorithm and worked on the flop count. Further on I created the fourmult implementation that implements four multiplications in one. **Luca.** Focused together with Tom on the optimization of the truncated Multiplication. Also made the benchmarks compile on macOS.

Tom. Focused together with Luca on the optimization of the truncated Multiplication. Created optimizations with pre-computations, replacement of complex with simpler functions, removed function calls and also tried loop unrolling and blocking. Further I did flop counts, compared the performance when using TwoSum vs. Fast2Sum and tried out some profiling tools.

7. REFERENCES

[1] Mioara Joldeş, Olivier Marty, Jean-Michel Muller, and Valentina Popescu, "Arithmetic algorithms for extended precision using floating-point expansions," *IEEE Transactions on Computers*, vol. 65, no. 4, pp. 1197–1210, 2016.

[2] Jean-Michel Muller, Valentina Popescu, and Ping Tak Peter Tang, "A new multiplication algorithm for extended precision using floating-point expansions," in *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, 2016, pp. 39–46.

[3] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al., *Handbook of floating-point arithmetic*, Springer, 2018.