OPTIMIZATIONS OF THE BALL ARITHMETIC

Tiancheng Chen, Ran Liao, Yunxin Sun, Lixin Xue

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

In scientific computing, there's a growing demand for high precision computing in order to retrieve the most accurate result. However, the common single precision (float) and double precision (double) floating point numbers available in most prevalent high level programming languages are limited in precision. They only provide 24 or 53 bits of mantissa respectively. In this project, we designed and implemented an efficient library for arbitrary-precision ball arithmetic, a interval arithmetic using the midpoint-radius representation.

We also support significantly optimized quad-double precision (at least 212 bits) ball arithmetic.

1. INTRODUCTION

Motivation. High precision floating point number arithmetic is one of the most common computations in every field of science and engineering. The midpoint-radius representation for ball arithmetic allows for efficient and rigorous high-precision numerical evaluation with error bounds. As such, it can be used in rounding error analysis, tolerance analysis, fuzzy interval arithmetic, and computer-assisted proof [1]. Ball arithmetic is about twice as fast as interval arithmetic and uses half as much space [1]. Therefore, it is also widely used in the scientific computing and engineering fields. The correctness and efficiency of the ball arithmetic are of great importance, as the ball arithmetic is the most basic operations in many the scientific computing and engineering applications.

However, implementing an efficient library of ball arithmetic is challenging. First, the algorithms for different operations vary a lot so there is no uniform way to perform the optimization. Second, the dependency between instructions, such as the carry bit in the integer addition, makes it hard to perform vectorization and utilize the instruction level dependency. Third, the support for arbitrary precision leads to constant memory allocation, copy, and deallocation, which are quite expensive.

Contribution. In this project, we first implement functions for arbitrary precision integer (denoted by Big Integer) operations. On top of that, we build arbitrary precision floating point (denoted by Big Float) operations. With these operations available, it is straightforward to implement the arbitrary precision ball arithmetic.

Related work. Arb [1] is a sophisticated C library that implements many complicated ball arithmetic operations. However, it is built on other arbitrary-precision integer arithmetic and floating-point arithmetic like GMP[2] and MPFR[3], which are heavy and error-prone. Instead, we build our own functions for arbitrary-precision integer and floating-point operations necessary for simple ball arithmetic operations like addition, multiplication, and division. We also supports fixed high precision operations to reduce memory accesses. QD[4] is a library using the unevaluated sum of four IEEE double precision numbers called quad-double to represent a numbet with at least 212 bits of significand. It implement four basic operations and various algebraic and transcendental operations for quad-double numbers. We build our fixed-precision ball arithmetic on top of this library and perform additional optimizations.

2. BACKGROUND ON THE ALGORITHM

In this section, we first introduce the representations and algorithms for the ball arithmetic and the quad-double arithmetic. Then we do a analysis on the cost of different operations in these arithmetic.

Ball Arithmetic Representation. As the other name of ball arithmetic, midpoint-radius arithmetic, suggests, a real number in ball arithmetic is represented by a midpoint m and its radius r, both of which are floating-point numbers and represent an interval $[m \pm r]$. In arbitrary precision ball arithmetic, the midpoint m is tracked to full precision and a common fixed precision floating number suffices for the radius r.

We implement the ball with an arbitrary precision floating point number as midpoint and a double precision floating number as radius. In the arbitrary precision floating point number, the mantissa is implemented as an arbitrary precision integer, while the exponent is represented by an 64-bit integer.

Ball Arithmetic Operations. The rules for the four basic operations of ball arithmetic is defined as follow: addi-

tion: $[a \pm r] + [b \pm s] = [a + b, r + s]$ (similarly for subtraction); multiplication: $[a \pm r] \times [b \pm s] = [a \times b, |a \times s| + |b \times r| + r \times s]$; division: $[a \pm r] \div [b \pm s] = [a \div b, |a \div b| + |a \div b \div b \times s| + |r \div b| + |r \times s \div b \div b|]$. Here the computation for the midpoints is full precision, while the computation for radii is only in double precision, where the arbitrary precision floating point numbers are first converted to double-precision and then being computed with other double-precision floating numbers. Therefore, the basic operations for ball arithmetic reduce to the basic operations of arbitrary-precision floating-point numbers, which can be implemented with the basic operations of the arbitrary-precision integers using the mantissa-exponent representation.

Big Integer Arithmetic. The naive algorithms for the addition and the multiplication of two n-digit numbers requires a number of elementary operations proportional to n and n^2 respectively. The divide-and-conquer Karatsuba algorithm[5] reduce the asymptotic complexity to $O(n^{\log_2 3})$.

Quad-double Arithmetic. A quad-double number utilizes the mantissa of four IEEE doubles precision numbers to represent a number with at least 212-bit precision, as the length of mantissa of a double is 53 bits. Each double represents at least 53-bit precision of the quad-double number, and the sum of four doubles is the actual value of the number. The four basic operations in quad-double arithmetic can be reduced to the additions and multiplications of double precision numbers and one re-normalization operation in the end[4].

Cost Analysis. For the ball arithmetic we build from scratch, the four basic operations consists of operations in the big integer with only a few floating point computation for radii. Therefore, we use the number of integer operations per cycle (including shifting and logical operations) as the metric for the performance evaluation of the ball arithmetic operations. For the quad-double implementation, as all the operations are basic arithmetic operations of doubles, we use FLOPs as the metric.

3. METHOD

In this section, we first introduce the data structure we use to store the ball. Then we introduce optimizations we have done to speedup several operations and some new operations we implement.

Data structure.

The big integer is implemented as a struct of an array of unsigned long integers, a sign field, a size field, and a capacity field. The size and the capacity fields are similar to those of the ‘std::vector‘ in C++ standard template library. One specific thing of our design is to use each 64-bit unsigned long integer to store 32-bit unsigned integer only, thus we can use additions and multiplications of unsigned long without worrying about type conversion and overflow

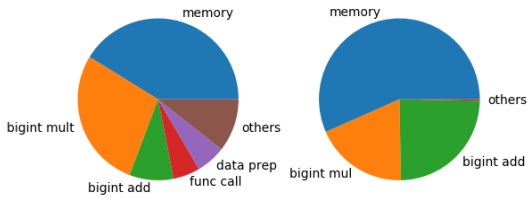

Fig. 1. Left: profiling for the multiplication of two balls. Right: profiling for the division of two balls.

in some steps. This is crucial for our SIMD vectorization in big integer multiplication, which will be explained later. With such design, the absolute value of the big integer is just the concatenation of the lower 32bits of all unsigned 64-bit integer in the array. The big float is implemented with a big integer as mantissa and a 64-bit signed integer as exponent. The ball is composed of a big float as midpoint and a double as radius.

Profiling.

We implement the library in C based on an open source big integer library[6] which only implement the addition and the subtraction operations. We add many more functions as we need and build the ball arithmetic library on top of it. We use this modular implementation as the baseline. We first profile the naive implementation to find out the bottleneck of the basic operations in ball arithmetic. We use perf to find the bottleneck of the basic operations for ball arithmetic. 3 shows the decomposition of runtime (measured in CPU cycles) of the multiplication and the division operations of the two balls. As shown in the charts, the memory allocation, deallocation, and copy take a large part of the time for both the operations. The multiplication and the addition of big integers are also quite time consuming. Therefore, we mainly focus on these three parts to optimize our library.

Memory Optimization.

Due to the support for arbitrary precision, we need to allocate and copy memory once the current storage is not enough to store the result, which are quite common in multiplication. This is very expensive as shown in the profiling section. One way is to allocate enough space in the very beginning. However, we have no prior on the typical usage of this library, thus cannot preallocate enough memory in the first place. So one optimization we have done is to provide the fixed precision version of all the functions, where only a fixed number of the most significant bits are kept and all other bits are discarded (treated as 0). With this simplification, now we can significantly reduce the memory operations.

Besides, there are some unnecessary memory operations in the baseline due to our modular implementation. We remove all these unnecessary memory accesses by inline the

functions, which also reduce the cost of function calls and enable the compiler to do further optimization.

Big Integer Addition.

We implement the baseline in a straightforward way by adding the unsigned integers starting from lower bits and then propagate the carry bit to next unsigned integer addition. In this way, there are $4n$ operations for the addition of two big integer of size n: $2n$ integer additions for the operands and the carry, n logical and operations to take the lower 32 bits of the sum, and n shift operations to get the carry.

The optimization for the addition is hard for two reasons: first, all the data are visited once, all the cache misses are compulsory misses, where we cannot do much about it; second, the existence of the carry bit leads to the dependency between instructions, make it hard to vectorize the code. Still. we try several methods to optimize it and it indeed boosts the performance a little bit.

The first thing we do is to inline all the function calls in the addition for various cases such as operands being zero and different signs of operands. This reduce the function call overhead and enable the compiler to do further optimization.

The second thing we do is to use _addcarryx_u32, the addition with carry intrinsic from Intel Intrinsics to speed up the performance. This reduces the number of integer instructions to n only as we now have use the carry bit in register to store the carry information. Since the gap of this intrinsic is 0.5, meaning we can issue two of such instructions every CPU cycle, we split the two operands into halves and add them independently to further enable instruction level parallelism. We propagate the carry bit from the place where we cut it to make the result correct, thus having a maximum overhead of $n/2$ instructions.

The last thing we try is to ignore the carry bit for now and use _mm256_add_epi64 intrinsic to add 4 numbers simultaneously. After all the summation is done, we extract the carry bits one by one and further propagate it.

Big Integer Multiplication.

We implement our baseline the most straight forward way. We implement a function that can multiply a single digit with another entire operand. Then we invoke this function n times and add their output together. This baseline implementation is simple, correct but quite inefficient. Too much unnecessary memory copy and allocation makes performance suffers.

Then we fix the precision to a specific value and inline everything we can, we use a 2-level-nested for loop to do the computation and thus remove unnecessary time consuming memory copy and allocation process. We also use scale replacement technology to reduce unnecessary computation.

To further improve performance, we try to increase instruction level parallelism. In the schoolbook long multiplication algorithm, we multiply each digit in one operand with the other operand. It's clear that each digit can do this process in parallel. Their results are independent to each other. So we unroll the outer for loop a little bit and compute 4 digits at a time.

To boost performance even further, we try to make use of vector intrinsics. We use *_mm256_mul_epu32* to multiply the lower 32 bits of input data and get a 64 bits output. We use *_mm256_add_epi64* for additions, *_mm256_and_si256* for and operations and *_mm256_srli_epi64* for shift operations.

Then, we try to unroll more to increase parallelism in these vector instructions.

Moreover, we try to reduce the number of integer operations in multiplication. We notice that propagating carry bit forward is time consuming. We need to do 3 operations (1 and, 1 shift, 1 add) to propagate it one step forward. We need to do this repeatedly because we store 32 bits of data in each computation unit. We use 64 bits *long* as the underlying data structure to store it. The multiplication of two 32 bits integer is a 64 bits integer. Unless we propagate it forward immediately, it might overflow and the output will become incorrect.

ver	#bits	#bits^2	#add max	# add req	intop
0x	32	64	0	0	$5n^2$
1x	30	60	16	> 4	$3.5n^2$
2x	30	60	16	> 12	$2.75n^2$
4x	29	58	64	> 28	$2.37n^2$
8x	29	58	64	> 60	$2.18n^2$

Table 1. Reduced Intop

We want to reduce the number of integer operations introduced by this propagating process. If we store less data in each unit, then the data after multiplication will be smaller. Thus we can do more computation before it overflows. We will be able to do propagation less frequently.

In table 1, we summarize how much we can reduce the number of integer operations in our different implementation. *0x* is the version without this particular optimization. Column *#bits* is the number of bits data we store in each unit. Column *#bits*^{2} is the number of bits after multiplication. Column *#add max* is the number of addition we can do before such multiplied number overflows. Column *# add req* is the number of addition we need to enable this particular optimization. And the last column is the number of integer operations after doing this optimization.

However, as we store less data in each unit, we need more units to preserve the same level of precision. There's a trade-off here. From our prospective, implementation *2x* looks the most promising.

Lastly, we try to optimize it even further for precision 256 bits. It's roughly the precision for quad-double. we

unroll all loops and rearrange them to provide the most instruction level parallelism we can possibly have.

Summation.

Suppose we want to compute the sum of k balls, our baseline implementation is invoke ball addition function $k-1$ times.

Then we try to inline it. We create a for loop that add all i*th* digits from input data together and store the result in i*th* position. And we propagate carry bit forward as we computes in each loop iteration.

It's also possible to use vector intrinsics here. Instruction *_mm256_add_epi64* can add 4 number at the same time. We use it to speedup the computation. Then only at the end of the whole computation, we propagate the carry bit forward.

Division.

We implement the arbitrary-precision division based on Newton–Raphson division [7]. The high level idea is to choose an appropriate initial value, and then iterate to a certain times based on the required precision.

Vector Operation.

We also implement vector operations of the ball arithmetic in our project. More specifically, a n dimension vector is made up of n balls, and each ball has its own radius and center. The vector operation is defined as operating on the elements of the same index of the operand vectors, and store the result in the result vector. We only implement vector add and its optimizations in this project, but other operations should be similar and straightforward.

Now we will talk about the implementation and the optimization of the vector add operation. A straightforward idea is to just perform n ball arithmetic add for two n dimension vectors. This implies n ball arithmetic add and servers as our baseline. However, there is no dependency between each element, and we could actually put each four elements in a SIMD slot, so that we can increase our parallelism. We use unsigned long for integers and each SIMD slot has 256 bits in total, so the maximal theoretical speed up is 4x. However, the performance gain will always be less than 4 because there is additional overhead as well.

Quad-double Addition and Multiplication.

We define a data structure: quad-double array, which consists of n quad-doubles. And we mainly optimize batch addition and multiplication on quad-double arrays to gain more potential in optimization.

Quad-double arithmetic is a fixed algorithm. It's not possible to be scaled to n-double precision at compile time or run time.

The addition and multiplication are one-pass algorithms, so there's no memory reused. Then it's not necessary to perform locality optimizations.

First, we do one-level inlining on simple functions (consisting of several double operations) and this has no effect on runtime. Probably because the compiler has inlined these simple functions as there's no memory aliasing.

Second, we vectorize addition and multiplication using AVX2 in a way that the functions called in addition and multiplication are vectorized. Also the memory allocation is 32B aligned, so that aligned load and store can be utilized.

Third, we inlined all the functions called except for re-normalization. So that we can perform loop unrolling to achieve further speed up. The re-normalization is at the end of functions, so it does not affect ILP of vectorization and unrolling.

We did not vectorize or inline the re-normalization based on these facts: One way to vectorize the re-normalization function is to calculate the result in all branches and use computed mask to pick the result. In the branching, the original version has 9 FLOPs and the vectorized version has 40 FLOPs (because every branch has to be considered), not including the computation of mask and assembling and disassembling of ymm registers. Even though the vectorized version can be inlined and gains further speedup, we decide not to vectorize it. Also, we used `perf` to measure the original version. Under 8MB data size, the branch miss is about 0.4%, which means the branching is not a bottleneck here.

We also did some optimizations which turned out to be useless. One is reordering instructions by hand to decrease data dependency, and another is renaming variables to resolve WAW, WAR conditions, but it's clear that the compiler has done all of these.

4. EXPERIMENTAL RESULTS

Multiplication.

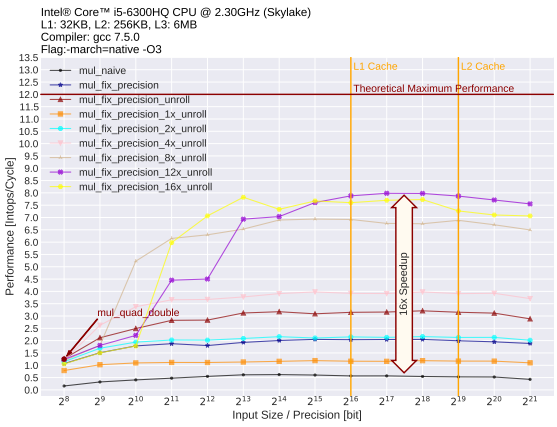

Fig. 2. Performance Plot for Multiplication

Figure 2 is the performance plot for multiplication. The black line in the bottom is the baseline implementation we mentioned in the beginning. The blue one is the fixed-precision and inlined implementation. The purple and the

yellow line is the most optimized version we have. The speedup gain from vector intrinsics. Basically, the more we unroll, the more instruction level parallelism we have, the higher the performance. And the red dot in the left bottom corner represents the performance that we optimized specifically for 256 bits precision. It's around 1.25 intop/cycle. a little bit higher than all other implementation in this particular precision. From the baseline to the most optimized version, we have approximate 16x speedup. In our test machine there're 3 ports that can issue integer vector instructions in each cycle. Therefore, the theoretical maximum performance should be 12 intop/cycle considering vector instructions. We achieve around 60% - 70% of theoretical maximum performance.

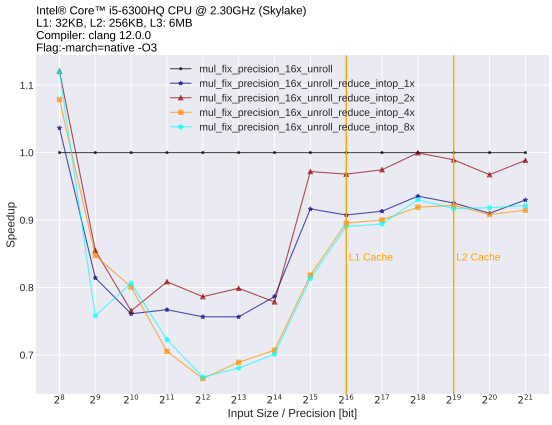

Fig. 3. Speedup for Reducing # Intop

Figure 3 is the speedup plot after we apply the reducing integer operations trick on the most optimized implementation. Most of the data points fall below 1.0, which indicating this optimization doesn't work. One possible reason behind this could be the memory bound. Though we indeed reduce the number of integer operations, we still need to access the output array repeatedly. There're lots of load (*_mm256_loadu_si256*) and store (*_mm256_storeu_si256*) instructions. This part becomes the bottleneck and limits the performance.

Summation.

Figure 4 shows doing summation directly can be a lot faster than doing addition repeatedly. Dark blue line *sum_4* represents the summation of 4 balls. Dark red line *sum_8* represents the summation of 8 balls, and so on. Through this optimization, we can achieve as high as 4x speedup.

Figure 5 is the performance plot after we apply the vector intrinsics optimization to the best previous implementation. It can help to achieve another 1.5x speedup when the working data is small enough to fit into L1 cache. When the working data is larger than caches, the performance drops significantly. Data movement becomes bottleneck.

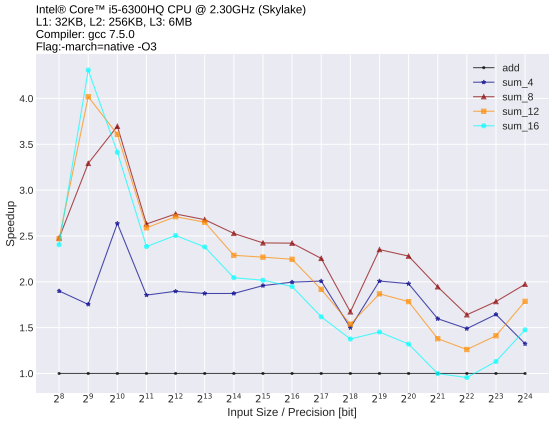

Fig. 4. Speedup for Summation

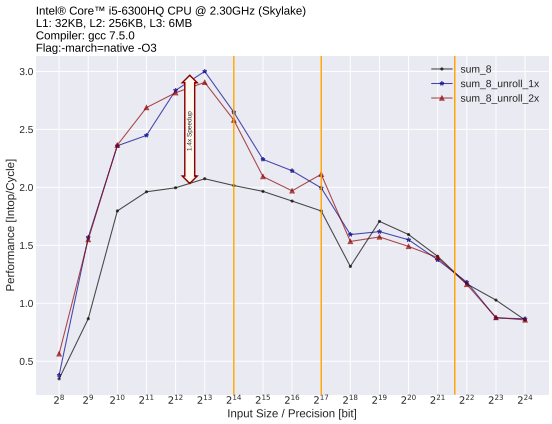

Fig. 5. Speedup for Summation using vector intrinsics

Vector Add.

Figure 6 shows the relative speed up of performing vector add using SIMD compared with the baseline. The baseline is just computing the ball arithmetic of every vector element in a straightforward way. We conduct the experiment in 4-element vectors, 8-element vectors, and 16-element vectors. The theoretical max speedup is 4x because one SIMD slot can hold four unsigned long integers of our implementation. In practice, we could see that when the data size fits into the L1 cache, the speedup is almost as 4x. When the working set size is larger than the L1 cache size, the speedup gets smaller and smaller because there's some overhead in the memory stack to move the data back and forth.

Quad-double Addition.

The meaning of x-axis: quad-double array size n means two quad-double arrays, each of which has n quad-doubles. And operation performed on quad-double at same index.

The result of quad-double multiplication is similar to addition, so only addition will be shown here for clarity.

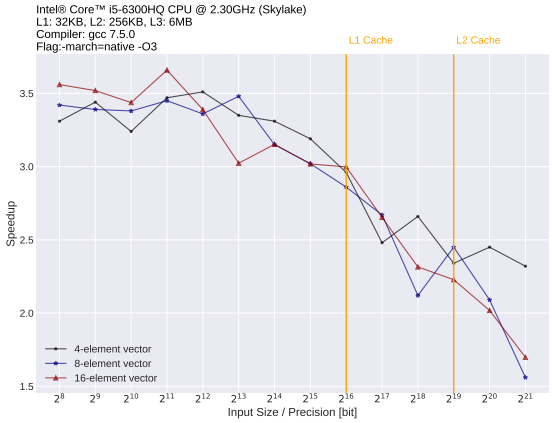

Fig. 6. Speedup for Vector add

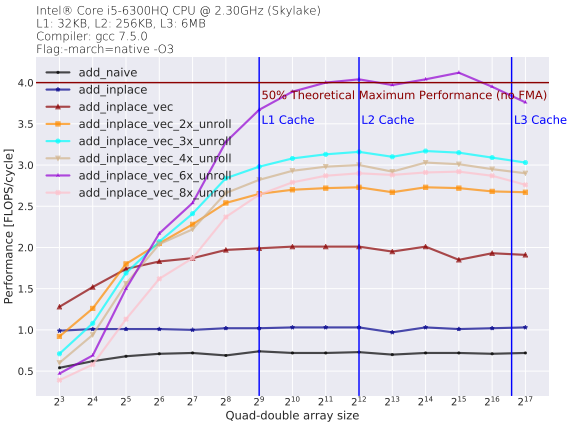

Fig. 7. Performance of quad-double batch add

Figure 7 shows the performance of quad-double addition. The black line is the naive implementation. `add_inplace` refers to inplace version. `add_inplace_vec` refers to SIMD and inlined version. The rest are versions of different loop unrolling factor. From the result, the version with 6x loop unrolling (24 pairs of quad-doubles in a loop because of SIMD) has highest performance with large data size, which is about 5.5x speedup. And peak performance is 4 FLOPs/cycle. The performance does not decrease much after data size exceeds the L3 cache, which means memory bandwidth is not the bottleneck.

Here we compare the quad-double arithmetic with bigInteger arithmetic. The bigInteger precision is set to 2^3, 256 bits that is closest to $>$ 212 bits of quad-double. Optimized bigInteger requires 258 cycles to compute. Optimized quad-double requires only 40 cycles on average (array size 2^{13}).

It's actually not fair to compare the both, because some facts as follows. First, quad-double is based on double operations while bigInteger is based on integer. Second, the algorithm for quad-double is well-designed for quad-double and is not trivially portable to n-double, while algorithm behind bigInteger is heuristic and portable to arbitrary precision. Third, quad-double is optimized for batch operation, while bigInteger is not.

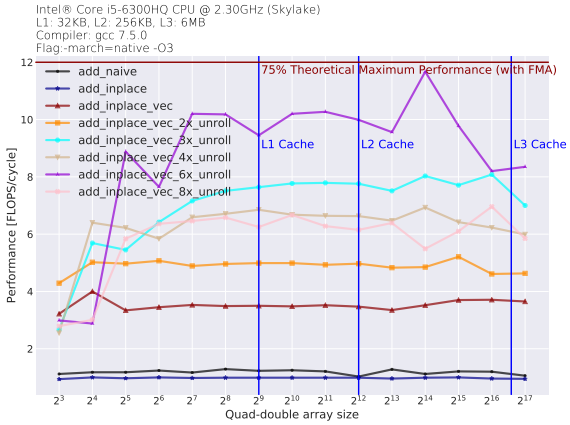

Fig. 8. Performance of quad-double batch add (overhead subtracted)

We also tested a version with only memory allocation and re-normalization, to see if the main part of the function is optimized to our best. Subtracting cycles of overhead from original data, the result is shown in figure 8. The highest speedup is around 8x to 10x. The peak performance is around 10 to 12 FLOPs/cycle which is between the theoretical peak without FMA (8 FLOPs/cycle) and with FMA (16 FLOPs/cycle). The number of FMA operations in the function is fixed and only consist of a small portion. So this figure indicates we have nearly achieved peak performance.

5. CONCLUSIONS

Here you need to briefly summarize what you did and why this is important. *Do not take the abstract* and put it in the past tense. Remember, now the reader has (hopefully) read the paper, so it is a very different situation from the abstract. Try to highlight important results and say the things you really want to get across (e.g., the results show that we are within 2x of the optimal performance ... Even though we only considered the DFT, our optimization techniques should be also applicable ....) You can also formulate next steps if you want. Be brief.

In arbitrary ball arithmetic, the cost of calculation of ball radius and exponent of middle point is constant. Only mantissa is input (precision) dependent. So we focused on optimization of integer arithmetic.

Multiplication: We used 64 bit unsigned long array to store the mantissa. The upper 32 bits are normally zero to

store temporary bits in multiplication. In this way, we vectorized the nested for-loop. We also did scalar replacement and loop unrolling to achieve 16x speedup and about 60% theoretical max performance. We also tried to reduce integer operations. If we store less data in lower 32 bit, then we can propagate the carry bits less often. However this optimization did not work probably because memory bound.

Summation: Previously if we add k balls, the addition is called $k-1$ times. We optimized this case by inlining addition function and add data at same index together then propagate carry bit once. We also tried vectorization, and achieved 1.5x speedup with input data fit in L1-cache.

Vector add: We use vectorization and inlining to perform 4/8/16 ball additions in a row by putting elements at same index in ymm register. When test data fits into L1 cache, the speedup is about 4x, which is the theoretical peak. For larger data, data movement becomes a bottleneck.

Quad-double add: We vectorized and inlined most part of the function and unrolled the loop with different factors. The best version achieved max performance 4 FLOPs/cycle. To verify if there's still some space for further optimization, we tested the overhead. And the overhead-subtracted version achieves max performance of about 10 FLOPs/cycle.

6. CONTRIBUTIONS OF TEAM MEMBERS

Tiancheng Chen. Worked with Yunxin on float and ball arithmetic implementation. Focused on addition, multiplication and casting between double and our BigFloat. Also implemented optimization of quad-double batch addition and multiplication with function inlining and SIMD. Tested and benchmarked the performance and run time. Analysed based on the result.

Ran Liao. Focused on multiplication part and summation part, including using SIMD instruction, reducing integer operation trick, 256 bits multiplication optimization and test their performance/runtime.

Yunxin Sun. Worked with Tiancheng on float and ball arithmetic implementation. Focused on division part. Also implemented a new data operation, vector for ball arithmetics. Used SIMD to parallelize the vector operation, and tested the performance/runtime.

Lixin Xue. Worked with Ran on infinite precision integer operations. Also focused on optimizing the integer add operation using SIMD instructions, and test the performance/runtime.

7. REFERENCES

[1] Fredrik Johansson, "Arb-a c library for arbitrary-precision ball arithmetic," 2018.

[2] The GMP development team, "Gmp: The gnu multiple precision arithmetic library," http://gmplib.org.

[3] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann, "Mpfr: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, pp. 13–es, June 2007.

[4] Yozo Hida, Xiaoye S Li, and David H Bailey, "Library for double-double and quad-double arithmetic," *NERSC Division, Lawrence Berkeley National Laboratory*, p. 19, 2007.

[5] A. Karatsuba and Yu. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Proceedings of the USSR Academy of Sciences*, 1962, vol. 145, p. 293–294.

[6] Andre Azevedo Pinto, "Ansi c biginteger," https://github.com/andreazevedo/biginteger.

[7] Liang-Kai Wang and Michael J Schulte, "Decimal floating-point division using newton-raphson iteration," in *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004.* IEEE, 2004, pp. 84–95.