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. We carry out many optimizations on the crucial part of the big integer addition and multiplication, achieving 60% of peak performance for the big integer multiplication. We also optimize the sum of multiple big integers and vector addition for balls. Besides, we optimize the high-precision quad-double floating-point arithmetic and achieve the theoretical peak performance.

1. INTRODUCTION

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 ball arithmetic are of great importance, as ball arithmetic is the most basic operations in many 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 operations. On top of that, we build arbitrary-precision floating-point operations. With these operations available, we implement arbitrary precision ball arithmetic.

We carry out various optimizations for the big integer addition and multiplication, finding out the bottleneck for addition and reaching a 16 times speedup for multiplication. We implement and optimize two new operations of great use: the sum of multiple big integers and the vector addition of two ball arrays. Moreover, we optimize the quad-double arithmetic and get decent results.

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 support 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 number with at least 212 bits of significand. It implements 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 ball arithmetic and quad-double arithmetic. Then we do an analysis on the cost of different operations in this 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 a 64-bit integer.

Ball Arithmetic Operations. The rules for the four basic operations of ball arithmetic is defined as follow: addition: $[a \pm r] + [b \pm s] = [a + b, r + s]$ (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] reduces the asymptotic complexity to $O(n^{\log_2 3})$. We implement the arbitrary-precision division based on Newton–Raphson division [6]. The high-level idea is to choose an appropriate initial value, and then iterate to certain times based on the required precision.

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 the 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 consist of operations in the big integer with only a few floating-point computations 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.

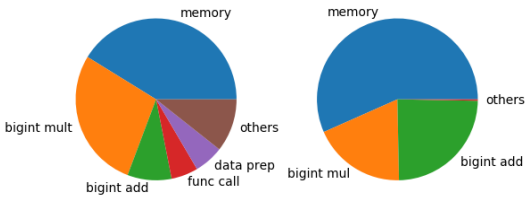

Fig. 1. Left: profiling for the multiplication of two balls. Right: profiling for the division of two balls.

3. METHOD

In this section, we first introduce the data structure we use to store the ball. Then we present the optimizations we have done to speed up several operations. We also add some new operations that are common in usage and easy to optimize. Besides, we try to optimize the quad-double arithmetic.

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 the 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 in some steps. This is crucial for our SIMD vectorization in big integer multiplication, which will be explained later. With such a 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 the mantissa and a 64-bit signed integer as the 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[7] that only implements 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. Figure 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 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

is 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 the next unsigned integer addition. In this way, there are $4n$ operations for the addition of two big integers 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 function, where there are various cases such as operands being zero and different signs of operands. This reduces the function call overhead and enables 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 can use the carry bit in the 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 individual additions are done, we extract the carry bits one by one and further propagate them.

Big Integer Multiplication. We implement our baseline with the quadratic 'primary school' algorithm. 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 its performance suffer.

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 scalar replacement 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 of each other. So we unroll the outer for-loop a little bit and compute 4 digits at a time. This trick is enabled by our design of the data structure: we use only 32 bits of the entire 64 bits of an unsigned long integer. Therefore, we can forget about the overflow in the multiplication for one step.

To boost performance even further, we use Intel 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. Together with loop unrolling, this SIMD optimization gives us a great performance boost.

Another direction we try is to reduce the number of integer operations in the multiplication. We notice that the propagation of the carry bit is time-consuming as it takes 3 instructions (1 and, 1 shift, 1 addition) to propagate the carry bit one step forward. We need to do this immediately after the multiplication as the product of two 32 bits unsigned integers will be 64 bits long, where there is no space to store the additional carry bit if we add two 64 bits long integers together. If we don't propagate the carry right away, the result could be wrong.

We try to reduce the number of integer operations introduced in this carry propagation process by storing fewer bits per 64 bits. If we store n bits in every 64-bit unit, the product of two such integers will be $2n$ bits long in a 64-bit container. Then we can add 2^{64-2n} of such products together without worrying about the overflow problem, being able to do the carry propagation less frequently.

In table 1, we summarize how much we can reduce the number of integer operations with a different number of actual bits in the 64-bit container. *0x* is the version without this particular optimization. Column *#bits* is the number of bits 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. The last column is the number of integer operations after doing this optimization.

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

However, there is a trade-off: as we store fewer bits in each unit, we need more units to preserve the same level of precision. In essence, it is the trade-off between memory and computation. If a machine is compute-bound in the multiplication, it would help to further reduce the runtime.

Lastly, we try to optimize the multiplication for a specific 256-bit precision. We unroll all loops and rearrange them to provide the most instruction-level parallelism we can possibly have.

Sum of Big Integers. Suppose we want to compute the sum of k balls, our baseline implementation will invoke ball addition function $k-1$ times, leading to an invocation of big integer addition k times. For each call of big integer addition, we will do the carry propagation for the entire array. However, if we know will add multiple big integers beforehand, we can sum up the numbers of all operands in each position, and then do a carry propagation for only one time, instead of $k-1$ times. In this way, we not only save the number of operations needed but also make SIMD using vector intrinsics possible.

Vector Operations for Balls. We also implement vector operations of the ball arithmetic in our library. More specifically, a n dimension vector is made up of n balls, and each ball has its own radius and centre. 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 the vector addition and its optimizations in this project, but other operations should be similar and straightforward.

The 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 every four elements in the same register and use vector intrinsics to increase parallelism. We use 64-bit unsigned long to store 32-bit unsigned integers and each SIMD slot has 256 bits in total, so the maximal theoretical speedup is 4x for this application.

Vecotr Operations for Quad-doubles. For the quad-double representation of high precision floating point numbers, we define a data structure, quad-double array, which consists of n quad-doubles, to enable vector operations. We mainly optimize the vector addition and the vector multiplication on quad-double arrays. Again, both the addition and the multiplication in the quad-double arithmetic are one-pass algorithms, so there is no memory reused and no much space for locality optimizations.

First, we do one-level inline in simple functions (consisting of several double operations), though this by itself has no effect on the runtime.

Second, we vectorize the addition and the multiplication using AVX2 in a way that the functions called in addition and multiplication are vectorized. We implement the quad-double array as 4 double arrays of size n, the ith array represents the ith doubles of n quad-doubles. By our design of the data structure, we can retrieve doubles at a specific position of 4 quad-doubles at the same time. Since the memory allocation is 32B aligned, 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 speedup. The re-normalization is the last step of the operation, so it does not affect the ILP of vectorization and unrolling.

We do not vectorize or inline the re-normalization part based on the following reasons. One way to vectorize the re-normalization function is to calculate the result in all branches and use a computed mask to pick the result. The original version needs to execute 9 floating-point operations while the vectorized version needs to execute 40 floating-point operations excluding the computation of mask and assembling and disassembling of ymm registers, as every branch has to be considered. Therefore, we decide not to vectorize it even though the vectorized version can also be inlined to gain more speedup. Also, we used `perf` to measure the branch misprediction rate of the original version. When the data is smaller than 8MB, the branch miss rate is about 0.4%, which means the branch prediction is not a bottleneck here.

We also try some other optimizations. One is reordering instructions by hand to decrease data dependency, and the other is renaming variables to resolve WAW, WAR conditions. However, we gain no speedup and it is possible that the compiler has done all of these.

4. EXPERIMENTAL RESULTS

In this section, we display the experiment results of all optimization methods mentioned in the previous section.

Experimental setup. We use the notebook Intel i5-6300HQ CPU @ 2.30GHz for the benchmarking. The L1, L2, L3 cache sizes are 32KB, 256KB, and 6MB respectively. We mainly use `gcc` 7.5.0 with `-O3` flag.

Big Integer Additions. Figure 2 is the speedup plot for the big integer addition. We use the speedup plot in-

stead of the performance plot since the number of integer operations differs significantly between methods and even data-dependent in some methods. For a fair comparison, we preallocate enough space for all the operands and the results to remove the influence of memory allocation, copy, and deallocation. We measure the performance in the cold cache scenario. The black line is the baseline implementation mentioned in the previous section. The blue line is the inline version of where we inline all function written by us. The red line is a version using the addition with carry intrinsic to significantly reduce the number of integer instructions needed. The orange line is a modified version of the previous method using intrinsics by splitting the operands into halves to enable ILP. The cyan line is the method where we ignore the carry bit first to use SIMD instructions and later propagate the carry bits.

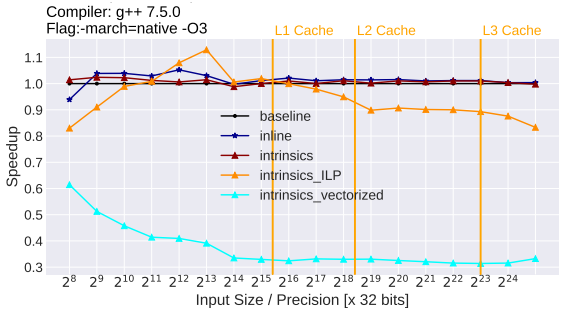

Fig. 2. Speedup for Big Integer Addition

From the figure 2, we can see that the optimizations we have done do not help much. This is reasonable as the operational intensity is O(1) for all these methods. For the `intrinsics_vectorized` version, visiting the array for a second time makes the performance degrade a lot. The `intrinsics_ILP` version could possibly visit part of the array for a second time to propagate the carry bit, so it is also slow when the operands cannot fit in the L1 cache. The problem with the intrinsics version (both the red line and the orange line) is that `gcc` does not works well with the `_addcarryx_u32`. It cannot generate a decent assembly code for this intrinsic after the examination of the assembly code. We try to compile the code with `clang` and `icc`, but it still cannot generate the `ADOX` instruction we desire. We could reduce the data transferred by using more bits in the 64-bit container. However, for the optimizations in the big integer multiplication, we have to use the current data structure to avoid overflow in multiplications.

Big Integer Multiplication. Figure 3 is the performance plot for multiplication. The black line at 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 integer operations per 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 are 3 ports that can issue integer vector instructions in each cycle. Therefore, the theoretical maximum performance should be 12 integer operations per cycle considering vector instructions. We achieve around 60% - 70% of theoretical maximum performance.

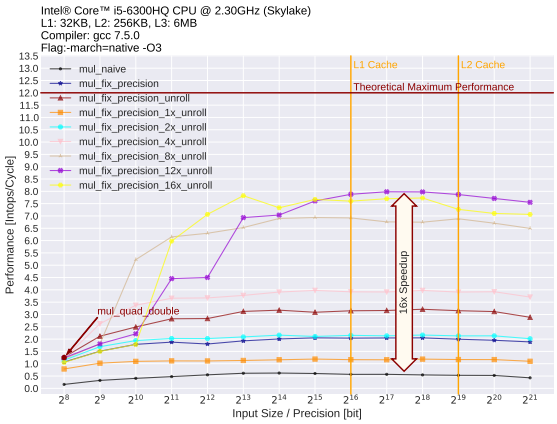

Fig. 3. Performance Plot for Multiplication

Figure 4 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.

Sum of Big Integers. Figure 5 shows doing the sum directly can be a lot faster than doing addition repeatedly. The dark blue line *sum_4* represents the sum of 4 balls. The dark red line *sum_8* represents the sum of 8 balls, and so on. Through this optimization, we can achieve as high as 4x speedup.

Figure 6 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 the L1 cache. When the working data is larger than caches, the performance drops significantly. Data movement becomes a bottleneck.

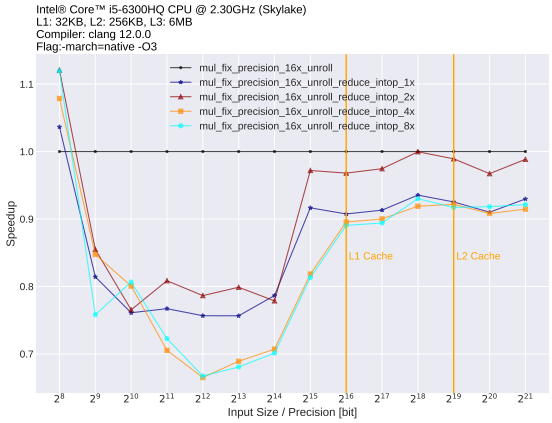

Fig. 4. Speedup for Reducing # Intop

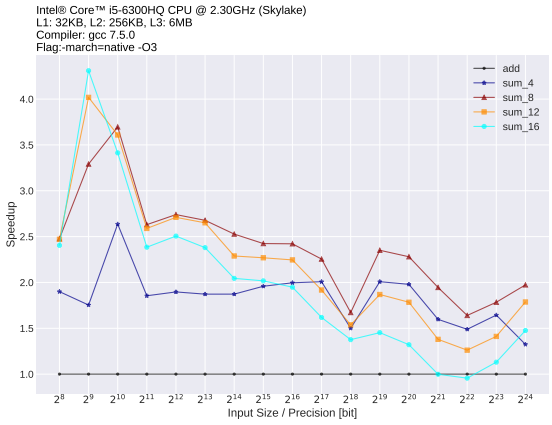

Fig. 5. Speedup for Sum of Big Integers

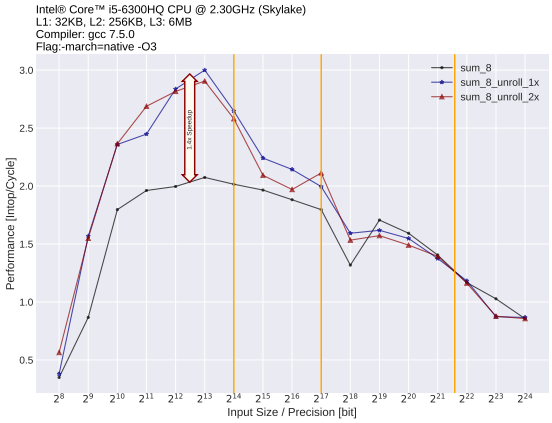

Fig. 6. Speedup for Sum of Big Integers Using Vector Intrinsics

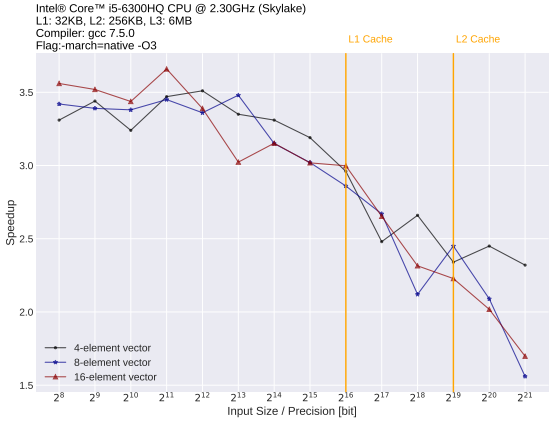

Fig. 7. Speedup for Vector add

Vector Operations for Balls. Figure 7 shows the relative speedup 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 4 times. 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 the same index.

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

Figure 8 shows the performance of quad-double addition. The black line is the naive implementation. `add_inplace` refers to the 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 the highest performance with a 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 big integer arithmetic. The big integer precision is set to 2^3, 256 bits that are closest to > 212 bits of quad-double. Optimized big integer requires 258 cycles to compute. Optimized quad-double requires only 40 cycles on average with an input array size of 2^{13}.

It's actually not fair to compare the both, because some

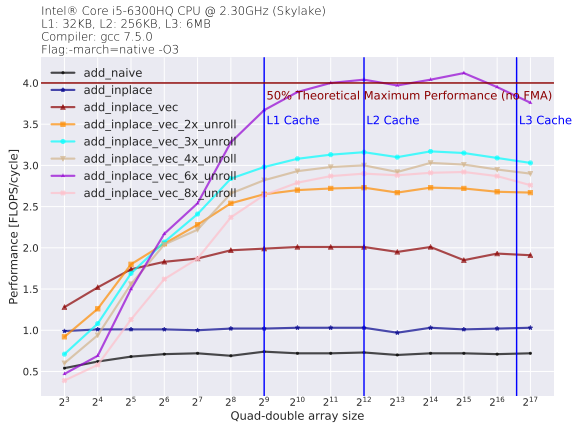

Fig. 8. Performance of quad-double batch add

facts as follows. First, quad-double is based on double operations while big integer is based on integer arithmetic. Second, the algorithm for quad-double is well-designed for quad-double and is not trivially portable to n-double, while the algorithm behind the big integer is heuristic and portable to arbitrary precision. Third, quad-double is optimized for batch operation, while big integer is not.

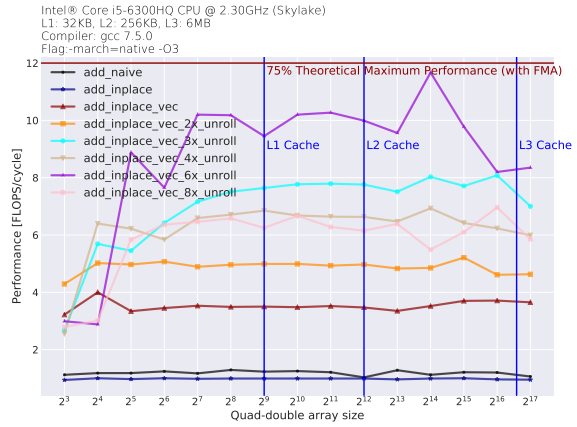

Fig. 9. Performance of quad-double batch add (overhead subtracted)

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 9. 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

In arbitrary precision ball arithmetic, the most crucial part for efficiency is the big integer addition and the big integer multiplication. The big integer addition is memory bound and there is no much space for memory optimization as there is no temporal locality on the input. We try several optimizations to reduce the number of integer operations and to enable ILP, but the gain in performance is limited. For the big integer multiplication, it benefits from the design of the data structure: we use an array of 64 bits unsigned long to store the integer, where the upper 32 bits are left as empty to store the temporary results in multiplication. With this design, we can vectorize the nested for-loop in the multiplication. With loop unrolling and scalar replacement, we achieve a 16 times speedup and reach about 60% of the theoretical max performance. We also tried to store fewer bits per 64 bits to reduce the number of integer operations. However, this attempt does not give us a boost in performance due to the memory-computation tradeoff.

In addition, we implement several new operations useful and easy to optimize. The first is the sum of multiple big integers, which will naturally occur when we sum up multiple balls at one time. The other operation we implement is the vector addition for balls, where we add two arrays of balls. When the data fits in the L1 cache, we achieve desired speedup for these two new operations.

We also try to optimize the quad-double representation of the high precision floating-point arithmetic. With inline, vectorization and loop unrolling, we achieve the peak performance of 4 floating-point operations per 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 the sum of big integers 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 the 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 on the big integer arithmetic implementations and optimizations. Focused on the optimization of big integer additions. Also responsible for the profiling

part of the library.

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] 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.

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