ARBITRARY PRECISION BALL ARITHMETIC

Kouroche Bouchiat, Gyorgy Rethy, Akanksha Baranwal, Jiaqi Chen

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Arbitrary precision arithmetic is important when we consider calculations with large numbers with interval arithmetic. We present an arbitrary precision ball arithmetic library that performs addition, subtraction, and multiplication. We focus on an optimized implementation for x86-64 systems using midpoint-radius intervals. We show that we achieve better performance in some cases when compared to libraries that exist out there.

1. INTRODUCTION

Motivation. Numerical computations are important for every field in science, with applications in cryptography, physics, and mathematics research. As such, having a library that accurately implements arithmetic calculations and algorithms is crucial. These computations should be reliable, efficient, be able to handle arbitrary precision, and maintain high quality error bounds. There exist interval arithmetic libraries that implement these computations. In this paper we present an arbitrary precision ball arithmetic library that enables efficient numerical calculations.

Contribution. Our main contributions is an arbitrary precision ball arithmetic library that implements the following elementary operations: addition, subtraction, long multiplication, and multiplication with the Karatsuba algorithm. The floating point numbers are represented by a midpoint value and a radius value. We find that our performance is comparable to that of existing libraries for some operations and investigate bottlenecks when it is not.

Related Work. Our library is very similar to the work of Johansson et al.[1] and Hoeven et al. [2] "`arblib`" uses arbitrary precision floating point arithmetic to perform midpoint-radius interval arithmetic. The midpoint has arbitrary precision and the radius of a ball is unsigned and has a fixed precision. In Hoeven et al., the authors provides a survey on the implementation of midpoint-radius interval arithmetic. They describe representations for the radius and midpoint, namely the radius is fixed-precision and the midpoint is an arbitrary precision floating point number. Their work also details trade-offs between the complexity and quality of calculations.

We attempt to make low-level optimizations in our library at multiple scales so that these arbitrary-precision types can also be exploited in high performance workflows.

2. BACKGROUND

In this section we outline the general structure of our library as well as the representation types of our ball arithmetic floating point numbers. *Please note that barith and `apbar` refer to the same types. The distinction is only made to avoid name conflict issues.*

Arbitrary Precision Ball Arithmetic. The basic idea of ball arithmetic is that instead of having error bounds represented by intervals with differing upper and lower bounds, ball arithmetic uses a radius and a midpoint value to represent the interval. Calculations are done on the midpoint and radius separately and the solution aims to have as small of an interval as possible. The advantage of midpoint-radius representations is that only the mantissa needs to be full precision, and we save space in representing the interval with just one radius value instead of an upper and lower interval bound.

Overarching Implementation Scheme. We represent our numbers in a hierarchical manner. We start with arbitrary precision integers, `apint`, and from there we build arbitrary precision floating point numbers, `apfp`, and finally we create arbitrary precision ball arithmetic floating point numbers, `apbar`.

We implement 3 main functions, addition, subtraction, and multiplication. For multiplication we also explore the Karatsuba multiplication algorithm [3]. We start by implementing addition, subtraction, and multiplication at the `apint` level, and from there we build into the `apfp` and finally the `apbar` level. As for the radius, we follow the radius calculations from [1].

Cost Analysis. We measure the number of additions, and multiplications. Here we only look at the cost analysis for the major functions, namely ball arithmetic addition, subtraction, and multiplication. For ball arithmetic addition, our most basic first implementation has $5N$ flops where we count both calculations for adding the midpoint and adding an arbitrary precision radius. Our most optimized ball arithmetic addition method has $2N$. Note that the most optimized `barith_add` has optimizations that include changing the representation of our midpoint, making the radius fixed precision, and using intrinsics. Additionally, when adding SIMD to `barith_add`, our cost was $8N$ but we calculate 4 adds at once. The radius calculations here still incur a cost but since we maintain fixed precision for them, that cost is negligible. For ball arithmetic subtraction, our cost analysis follows that of `barith_add` since subtraction and addition are the same just taking into consideration different signs. For ball arithmetic multiplication, our first implementation incurs a cost of $4N^2 + 8N^2 + 4N$ where the $4N^2$ comes from the midpoint calculation and the $8N^2 + 4N$ comes from the arbitrary precision radius calculation. After making the radius fixed precision, using intrinsics, and optimizing the overall representation of the midpoint and radius, our cost goes down to $4N^2$. Since we change the representation of the radius, it also incurs a fixed cost that is negligible.

Complexity Analysis. The complexity of addition is $O(n)$, the same for subtraction. For multiplication, the complexity of the straightforward algorithm is $O(n^2)$, where n is the length of the inputs. The complexity of the Karatsuba algorithm however, is $O(n^{\log_2 3})$. In all cases, n is the length of the floating point number, or the number of limbs.

3. IMPLEMENTATION

Simplifying assumptions. During the project, our efforts were focused on improving the performance of our arbitrary precision ball arithmetic library. In order to limit the scope of the project to optimizing core functionality, we made some simplifying assumptions regarding the arbitrary precision data types.

The *IEEE* 754 standard format can represent finite numbers of the form $m \cdot 2^e$, including two zero values ($+0$, positive zero; and -0, negative zero), two infinities ($+\infty$ and $-\infty$) and a special value *NaN* (Not-a-Number) which is used when the value is undefined or unrepresentable. Handling of the two infinities and *NaN* requires a lot of low computation code and "unlikely" branching, which is not particularly interesting to optimize, therefore our API doesn't define or handle these. We have partial support for zeros by setting the mantissa to zero, but it is mostly untested.

Most arbitrary precision floating point libraries support operations on data types that have different precisions. Due to time-constraints our library only functions with inputs and outputs with equal precision, but it is possible to "re-precision" our data types by creating a new structure and manually copying limbs. We also did not have time to explore supporting division and advanced operations like square-root, sin, cos, etc...

3.1. Arbitrary precision types

In order to develop a straightforward baseline and to simplify dividing work at the start, we separated the project into three layers with three respective data types. At the bottom we have an arbitrary-precision integer layer codenamed '*apint*'. The arbitrary precision number is represented using an array of 64-bit *unsigned* integers called "limbs". The structure contains the pointer and the size of the limb list and a boolean representing the sign of the number (positive or negative).

Next, we have arbitrary precision floating point numbers, referred to as '*apfp*'. We use our arbitrary precision integer type to represent the floating-point mantissa and the structure also contains a 64-bit *signed* integer for the exponent.

Finally, at the top-most level we have an API for arbitrary precision ball arithmetic which has its own encapsulating type. The '*apbar*' structure contains an arbitrary-precision floating point number representing the midpoint of the ball. The radius of the ball is represented using a 64-bit mantissa and 64-bit exponent, and in some optimized cases, is simplified to a double value. Using 64-bit integers for radius is significantly more precise but the tradeoff is having to manually maintain a consistent floating-point mantissa in software; work which is done in hardware for doubles.

One interesting aspect of our data structure is how the floating-point mantissa is stored in the limb list. Our team debated the advantages and disadvantages of using a *"left-aligned"* or *"right-aligned"* mantissa. When using *"left-aligned"* representation, the leading 1 of the mantissas is always aligned to the most-significant bit of the last limb. This has the advantage that the value of the 'exponent' field corresponds to the true exponent of the floating-point number, but it also means that last-limb overflow in addition is much more frequent. During addition, an overflow in the last limb requires a final shift of the whole mantissa by one bit to the right. In contrast, in *"right-aligned"* representation the leading 1 of the mantissa can be anywhere in the limb (and limb list), which means last-limb overflows in addition are much less frequent. On the other hand, finding the true exponent of the mantissa requires scanning through from the top for the very first leading 1. Knowing the true exponent is necessary when adding floating-point numbers since the mantissas need to be shifted so that the exponents are "aligned."

We settled with a variant of left-alignment that we refer to as *"middle-alignment."* When multiplying the mantissas for floating-point multiplication we know beforehand that the leading 1 of the result will land at $2p$ or $2p$–1, where p is the precision of the floating-point number. In order to avoid reallocating extra space for the multiplication result, we pre-allocate twice the precision when the user creates a new floating-point number. This means maintaining alignment of the leading 1 to the most-significant bit of the middle-right limb in the limb list.

During optimization we decided to “collapse” all the abstraction layers into a single structure that we call ‘*apbar2*’. It contains all the combined fields from the abstracted types: the pointer and size of the mantissa limb list, a signed 64-bit integer for the exponent, a double for the radius, and a sign boolean. Work on this type is joined as much as possible. For example, we perform mantissa limb alignment and addition simultaneously in `apbar2_add`.

3.2. Addition/Subtraction

Ball arithmetic addition/subtraction is defined as:
$\beta(x, r) \cdot \beta(y, s) = \beta(x + y, r + s)$ [2].
The major operations are: shift to align the mantissa of either operand, addition of the aligned mantissa, shift to realign to mid point representation. The optimization methodology for addition subtraction is similar so we have discussed the optimizations for addition only in detail.

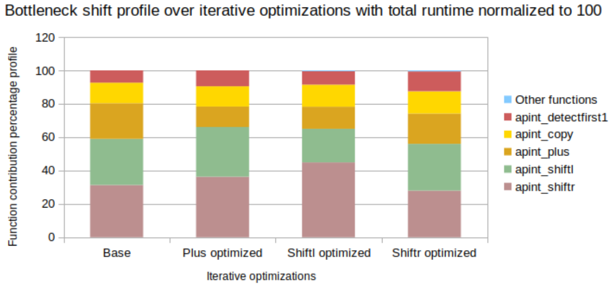

Fig. 1. Bottleneck shift profile for ball arithmetic addition for the worst case scenario

Iterative profiling and identifying bottlenecks. As visible from Fig. 1, the main bottlenecks of addition computation are shift operations, detecting first set (part of realigning to mid point representation), copy (part of aligning the mantissa) and the actual addition.

First the addition has been optimized to consider the mid point representation which reduces the number of iterations of addition, instead of iterating over the entire limb length we iterate only over half of it. The optimization also includes using vector intrinsics instead of the portable add. Since add has a carry dependency, it was not possible to vectorize this to improve it since the carry needs to propagate as we add digits.

For the shift operations the optimizations included are reorganizing the code to reduce unnecessary branching in the shift loop, which also helped reduce the total number of shifting iterations. Similar to addition, detecting the first set bit is difficult to optimize because of across loop dependency. This makes it difficult to optimize `apint_detectfirst1` any further.

Using Intel intrinsics. To improve upon the performance of our portable addition method, we replaced the addition with the Intel intrinsic `_addcarryx_u64`. This is the function `barith_add_intrinsics` in Fig. 2.

Customizing the functions for addition optimization. Since these separate generic functions were difficult to optimize due to across loop dependencies, we merged all of these functions into a common function. This enabled us to limit the generic functionality of the functions and instead customize them for addition. This was especially useful while handling overflow, because now we can leverage the fact that addition could have an overflow of at-most one bit. This is the function `apbar_add_merged` in the code.

Incorporating mid-point representation. We observed that there were multiple loops for shift, copy, detectfirst1 for which the number of iterations could be reduced instead of going over the entire limb length. Also there was some opportunity for scalar replacement in this version which helped improve performance. This is the function `barith_add_midptrep` in Fig. 2.

Reducing nested branching. When we added support for negative numbers to the code, a lot of branching was introduced. This optimization involved merging all concurrent branches into two cases depending on the sign of the operands. It was required to repeat some of the code in either branch but overall this redundancy in fact helped to improve the performance. This is the function `barith_add_nestedbranch` in Fig. 2.

Moving to the collapsed representation for optimization. Using the collapsed representation helped improve the performance of addition because it removed the hierarchical access to subsequent data types for any processing or information. This new data type also reduced the complexity of handling different signs as this code was much more simplistic. This is the function `barith2_add` in Fig. 2.

Optimizing the collapsed representation. This new simplistic code also had the opportunity of introducing scalar replacement, reorganizing the code to remove unnecessary branching which helped improve performance significantly. An interesting effect of this simplification was that the compiler was now able to vectorize one of the loops in shift operation which improved the overall performance. This is the function `barith_add2_optim1` in Fig. 2.

3.3. Multiplication

In ball arithmetic multiplication is defined as:
$\beta(x, r) \cdot \beta(y, s) = \beta(x \cdot y, \Delta((|x| + r)s + r|y|))$, where $\Delta(y)$, is the error bound function [2]. In simpler terms there are three distinct operations: the multiplication of the mid-

points, the radius calculations and the error bound calculations.

For the midpoint we rely on apfp_mul, which simply adds the two exponents, multiplies the mantissas (with the help of apint_mul) and then adjusts for any overflows that might have occurred. Multiplying two integers is a surprisingly difficult problem with a large number of algorithms. In the beginning we opted to implemented the so-called long or grade-school multiplication algorithm. This has a complexity of $O(n^2)$, but can be done in-place and has no hidden constants.

The radius and error bound calculations happen inside the top-most function (apbar_mul). In the beginning we performed all calculations that involved the midpoint in the precision of the midpoint, which lead to more accurate results (and a tighter error bound). However, this is extremely wasteful and yields worse performance than interval arithmetic. Therefore one of the first optimizations we performed was to narrow the midpoint to the precision of the radius (64 bits), decreasing the op-count substantially. The trade off here is that our error-bound slightly increases, but since we use more bits for the radius our error bound is still tighter than that of arblib.

After changing how we perform calculations on the radius, profiling revealed that essentially all the time is spent in apint_mul. First we found an intrinsic instruction that performs the multiplication and returns the overflow.

After this we applied more optimisations e.g. scalar replacement seemed to give a small benefit, however unrolling for ILP caused significant slowdowns. We found that the unrolled version was already very close to performing at the throughput of mulx, which means unrolling could not give us any performance benefits in the first place, but introduced some other overheads.

In the end as mentioned before we created a collapsed version, where we aimed to reduce unnecessary instructions such as function calls, pointer chasing etc... On top of the collapsed implementation we implemented the same standard optimizations.

At this stage we knew that our integer multiplication is bounded by the throughput of the multiplication instruction, we also eliminated as much overhead as possible, yet arblib seemed still visibly faster. To further optimize multiplication we started looking into other, more sophisticated algorithms.

Karatsuba multiplication. Karatsuba is a recursive algorithm that separates the inputs a and b that we want to multiply into their higher and lower bits. Then adds the higher and lower bits together for each input a and b. And then three different recursive calls are performed on these higher and lower bits, as well as the added bits, in order to get three new values. These values are then shifted appropriately and added together for the final result [3]. We wanted to explore the Karatsuba multiplication algorithm because of the potential speedup we though we could get from using a faster multiplication algorithm that was $O(n^{\log_2 3})$.

Method	Weight (seconds)	Weight (percentage)
apint_free	159.4ms	50.5%
apint_init	131.3ms	41.5%
apint_add	7.1ms	2.2%
apint_mul	2.9ms	1.0%
apint_copyover	1.4ms	0.3%
apint_sub	1.0ms	0.3%
apint_shiftl	0.9ms	0.2%

Table 1. Profiling for Karatsuba

The naive implementation of Karatsuba follows the algorithm straightforwardly. It is worth noting that in order to perform all the necessary operations, the recursive method has to initialize and free a lot of apint's. These are essentially temporary variables used to perform the calculations in Karatsuba. As such, Table 1 shows that our biggest bottle neck is in apint_free and apint_init. This aside, our next bottleneck was in apint_add. We gave considerations to optimizing the init's and free's however there was the trade off of having to allocate a large amount of space beforehand for all the values we need versus the size and precision of numbers we can handle.

After looking at the profiling for Karatsuba, we optimized with the following methods: optimizing adds, subs, shifts, unrolling apint_mul, or increasing the base case of Karatsuba. For the first set of optimizations, we extended the base case. This essentially means that instead of recursing down to limbs of size 1 and then performing regular apint multiplication, we enter the base case when the size of our inputs reach length 8. The next optimization was inlining all the method calls so we do not incur overheads in function calls. The last set of optimizations was using the most optimized sub-methods that were needed within Karatsuba, for example, unrolling apint_mul. The performance improvements are analyzed in Section 4.

3.4. Vectorization

Another path of optimization that we took was focusing on vectorizing arbitrary precision ball addition. The main difficulty of vectorizing arbitrary-precision types is overflow detection. SSE/AVX does not have an equivalent for carry flags when using vector registers, so overflow detection must be performed "manually" which comes at a cost.

There is no "free" carry detection like in the case of scalars when we use the '_addcarryx_u64' intrinsic, but instead we can use unsigned integer comparison to detect overflows after the addition of the limbs. When adding two unsigned integers, if the result is strictly smaller than one of the inputs, then we can assume the addition caused an overflow. AVX512 provides instructions for unsigned 64-bit integer vector comparison (`VPCMPUQ`), but we didn't have an AVX512 capable-CPU available to test on.

Using AVX2 there are multiple workarounds to do unsigned comparison. The first trick consists in taking the maximum and checking for equality. In the case of $a \leq b$ we can perform the same computation by checking if $max(a, b) = b$, but this is not a strict comparison. The other trick relies on converting unsigned values into equivalent signed values so that we can use the standard signed comparison instruction. To do this we flip the most significant bit in each lane which corresponds to the sign when interpreted as a signed integer. If we were applying this to bytes, you could think of this as pushing numbers in the $[0, 255]$ range into $[-128, 127]$ so that they could be compared using signed comparison. In AVX2, this translates into two XOR instructions to flip the most-significant bit on both comparands and a signed comparison instruction. `VPXOR`s can be performed in parallel and have a latency of 1, and `VPCMPGTQ` a latency of 3 on Skylake.

Initially we wanted to vectorize addition of two '*apbar*' types by putting the mantissa limbs "horizontally" into the vector registers, meaning instead of performing addition of 64-bit chunks of the mantissa, we could add up to 256-bit chunks which in theory would result in a 4x speedup compared to scalar addition. The issue is the lanes of the vector registers depend on one-another since a carry in any lane means adding one to the lane above. This is aggravated by the fact that adding a carry can cause an overflow and ripple through all the lanes. Because of this, we must check for overflow every time we add a carry. This means having to perform five carry overflow detections when adding two mantissa vectors. This ends up being much more expensive than performing four `ADCX` instructions, so we decided to explore another direction for vectorization.

Instead, we explored batched arbitrary precision ball addition to see if we could benefit from putting limbs "vertically" in the SIMD vector registers. `apbar2_add4` takes eight *apbar2* inputs and outputs four *apbar2* addition results. This removes the dependency on carry propagation in the same vector, so we only need to propagate carries into the next loop iteration. We perform carry detection twice: one time when adding the limbs and another when adding the carries from the previous iteration.

In batched addition, we used the `VPGATHERQQ` instruction to gather the limbs into the SIMD vectors. There is an equivalent scatter instruction that is available on AVX512 that we could also have benefitted from. We believe it might be possible to squeeze slightly more performance out of `apbar2_add4` by unrolling the loop and applying scalar replacement and other ILP optimization techniques but we did not have time to investigate this.

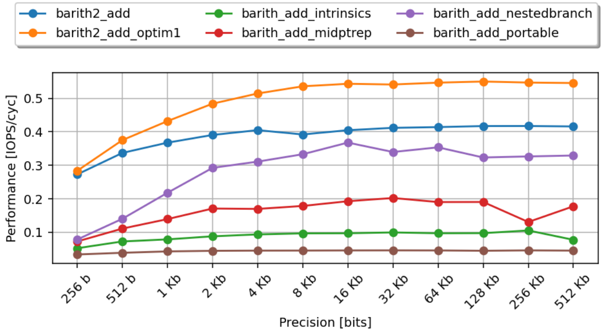

Fig. 2. Performance of different optimizations for addition

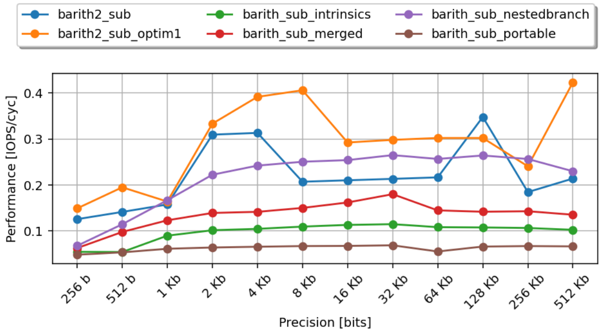

Fig. 3. Performance of different optimizations for subtraction

4. EXPERIMENTAL RESULTS

4.1. Experimental Setup

Our experiments were performed on a Ubuntu 18.04 machine with an Intel i5-8400 processor (TurboBoost off and frequency set 2800 MHz). Exclusive access to one of the cores on the machine was granted with a combination of the `isolcpus` kernel parameter and `taskset`. This was to alleviate any scheduler interference during a longer experiment. All background applications were killed to avoid L3 cache or memory bandwidth interference.

During compilation we used gcc version 9.4.0 with the following flags: `-mavx2 -madx -mbmi2 -mlzcnt -Ofast -march=native -mtune=skylake -funroll-loops`

4.2. Benchmarking Results

Addition/Subtraction. For addition, most of our optimizations are revealed iteratively. Following the order of implementation as detailed in Section 3 under Addition/Subtraction, we see that we achieve the best performance by introducing the collapsed representation of the floating point numbers. This makes sense as we were able to join and merge a bunch of code and remove a lot of unnecessary branching. It is worth noting that the optimizations for subtraction are similarly named and Fig 3 shows the performance improvements.

Figure 6 shows the speed up of our add function when compared to the base line (apbar_add_portable). We achieve better speed up than arblib for small input sizes. And overall, we are able to achieve relatively similar speed up as arblib. Compared to the base line function, we were able to optimize our code for around a 12x speed up. Figure 7 shows the speed up of our subtraction function when compared to the base line (apbar_sub_portable). We do not achieve similar speed ups as arblib except for with small input sizes. We hypothesize that this could be due to branching that happens due to the sign comparisons.

Multiplication. For multiplication most of our substantial optimizations came in the form of reducing op-count. For this reason we opted to show a speedup plot in this report since most functions would have a different cost making a performance plot misleading or incorrect.

Figure 4 shows the speedup against our base case (apbar_mul_portable) of different variants of ball arithmetic multiplications we have implemented. As you can see the collapsed and optimized version (barith2_mul_opt), is the fastest and we were able to achieve a speedup of up to 20x. You can also see how the unrolled version has a slightly worse speedup compared to the non-unrolled version.

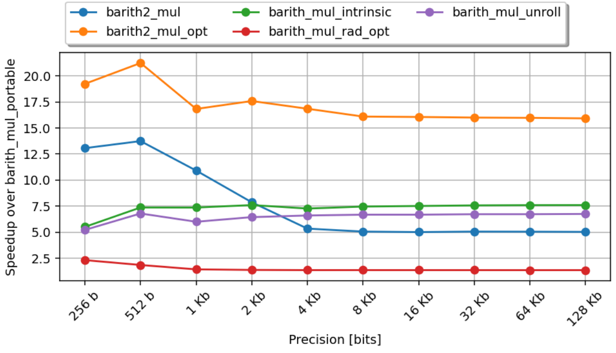

Fig. 4. Speedup gained from different optimizations for multiplication.

Arblib uses the GNU Multi Precision arithmetic library (GMP) for integer operations. This is a highly optimized library that implements seven multiplication algorithms in assembly (for most architectures). In this comparison we chose to compare for input sizes for, which GMP would use their implementation of the same algorithm we used. Figure 5 depicts the speedup achieved by us and arblib over our base implementation. As you can see, even though we were able to achieve speedups of around 17x to 20x, arblib still performs much better. After investigating with Intel VTune we found that the equivalent arblib call executed notably fewer instructions.

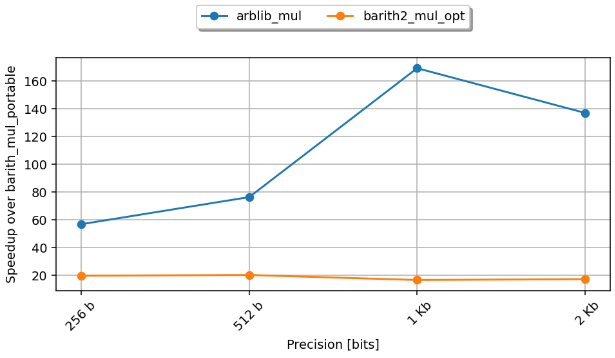

Fig. 5. Comparison of the speedup above our straightforward implementation by our implementation and by arblib

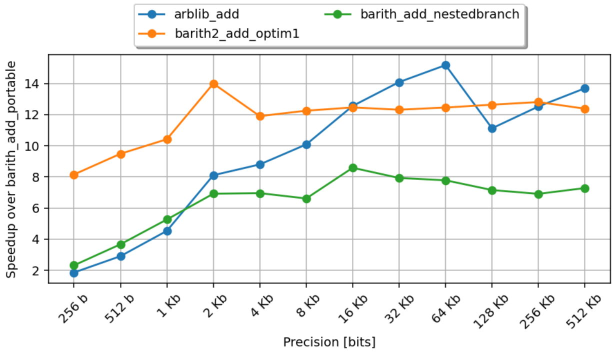

Fig. 6. Speedup gained from different optimizations of addition

Karatsuba Multiplication. Next we look at the improvements with respect to the Karatsuba algorithm. Figure 8 shows the speed up of Karatsuba where the baseline algorithm is apint_mul_portable. We see that the original Karatsuba performs worse than apint_mul_portable. Optimizations of Karatsuba perform better than portable multiplication, but worse than apint_mul. This is due to all of the malloc calls that we have to make in Karatsuba due to the recursion. However, when we look at the improvements with respect to Karatsuba itself, extending the base case gave the biggest improvements. This makes sense because we are using apint_mul more often instead of recursing deeper. Additionally, for small sizes the optimizations are dominated by the fact that we extended the base case. But when we look at larger sizes, we see that opt2 gave some marginal improvements. This makes sense as in opt2 we used the most optimized sub-method calls where

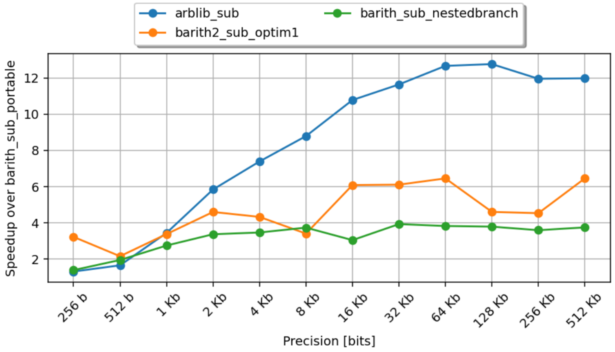

Fig. 7. Speedup gained from different optimizations of subtraction

as in opt1 we only inlined functions, and the inlining could already be done by the compiler. We attribute the dip at 1Kb to the size at which we extended the base case, since we return to using apint_mul at length 8 inputs, which is $8 * 64 = 512$ bits. All in all, Karatsuba is a very fast algorithm in terms of overall complexity, however we were not able to implement it with much improvement over our original multiplication method. This is due to the fact that we had to work with all the malloc calls in order to create the intermediary variables we needed for the recursion.

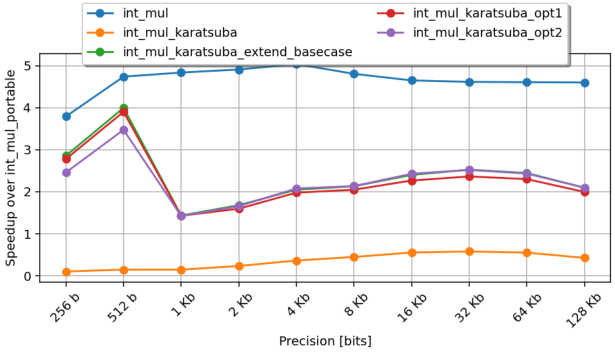

Fig. 8. Benchmarking the different optimizations of Karatsuba Multiplication

Vectorization. Results in Fig. 13, of the benchmark of batched addition versus scalar addition of four *apbar2* balls, appear to show that we were not able to benefit significantly from using AVX2 for batching. This is coherent with our hypothesis that the cost of AVX2 overflow detection and AVX2 gathering/scattering of the limbs is higher than just performing ADCX four times. Four ADCX have a total latency of 4, whereas performing a single AVX2 overflow check is already a latency of 4 on Skylake. We posit that this might be very different had we been able to use AVX512. Firstly, we would have been able to batch up to eight balls instead of four, and secondly, we would have had access to faster overflow detection using unsigned compare and faster scattering of limb results.

4.3. Roofline Analysis

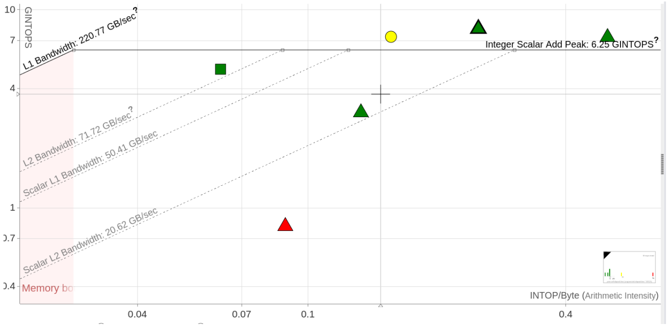

Fig. 9. Roofline plots for addition for precision 512 for the optimizations barith2_optim1 add (square), barith2 add (circle), portable add (triangle).

Addition/Subtraction. We performed roofline analysis for the different barith functions to compare the optimality of our implementation with the peak possible. We have used the Intel Advisor Tool for this section. The number of loops in the portable version are higher usually, so the number of points on the roofline plot corresponding to this version are higher. In the collapsed version these loops have been merged together. We have also plotted shifts, so the points which are above the add roofline are actually these functions. The points above the horizontal add roofline correspond to shift operations

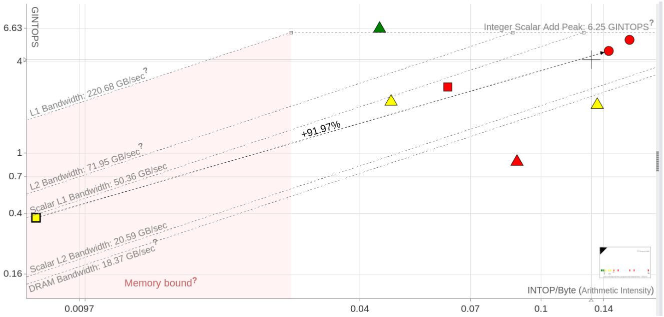

Fig. 10. Roofline plots for addition for precision 32768 for the optimizations barith2_optim1 add (square), barith2 add (circle), portable add (triangle).

The roofline plot includes three different stages of optimizations: the portable version we started with, the collapsed version and the optimized collapsed version. At both higher (32768) 9 and lower precision (512) there is a clear trend of shift towards a closer memory roofline. In portable version there are a few functions which are DRAM bound, whereas the final optimizations lead to better memory access patterns which shifts the performance to be limited by cache memory bound instead. As we discussed before, addition is limited by the carry dependency, that is why we don't see it being compute bound in any case.

Multiplication. We have also performed a roofline analysis on multiplication as well. Figure 11 shows the roofline plot of three functions at different levels of optimisation running with 512 bits of precision. Figure 12 shows the same three functions, except now with 32768 bits of precision. We observe that the most optimal version (barith2_mul_opt) has a higher operation intensity than the similar version, which contains no standard optimisations. We can also observe that the portable version has the highest operational intensity, but a lower performance. The portable version has to perform more instructions to get the same result so this is expected.

We can also observe that as the input size increases both operational intensity and performance increases as well. The portable version containing two loops becomes bound by compute, while the less optimised barith2_mul is bounded DRAM bandwidth. By increasing the operational intensity of our most optimised version we were able to avoid this bound.

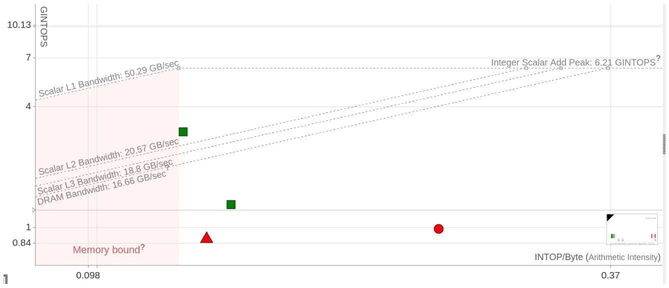

Fig. 11. Roofline plots for multiplication for precision 512 for the optimizations barith2_optim1 mul (square), barith2 mul (circle), portable mul (triangle).

5. CONCLUSIONS

In this project we tried to tackle optimization of arbitrary-precision ball arithmetic for elementary operations (addition, subtraction, multiplication). We were successful in optimizing addition and subtraction for small precisions where we mostly benefited from intrinsics and ILP optimization techniques. In the case of multiplication, we explored implementing the Karatsuba multiplication algorithm for arbitrary precisions but found that memory layout should be the focus of optimization.

We explored vectorization of arbitrary-precision floating-point addition and concluded this is difficult because of inter-lane carry dependency. We also concluded that batched addition of floating points should be faster on AVX512-capable CPUs and later since overflow detection requires workarounds on AVX2. One direction we did not have time to delve into is Software Carry-Save[4]. Reserving extra space in the most-significant bits of limbs for carries should help when performing multiple operations in quick succession before explicitly "re-normalizing" by propagating these carries through. We have noticed that our code spends significant time detecting and properly handling carries, so if this work can be pushed back to a dedicated carry propagation step some workflows would benefit from this.

At the end of project we also noticed that the performance of integer multiplication when ran in itself is much higher than when we are running it through apbar or apbar2. The performance difference seems to be around 2x and this would bring our results much closer to that of arblib. Our hypothesis is that this is due to the layout of data in memory, which would also explain the performance spikes we could see for specific precisions in some operations.

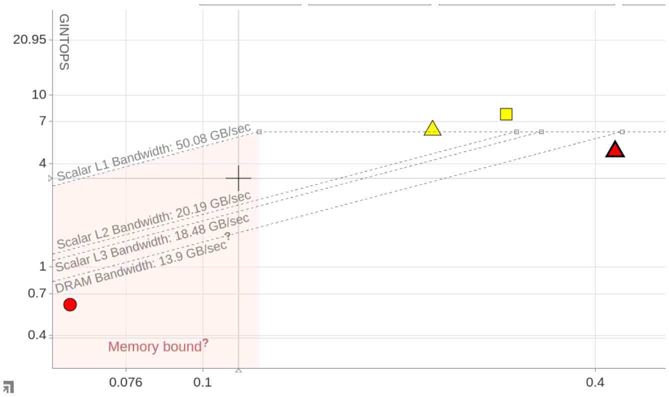

Fig. 12. Roofline plots for multiplication for precision 32768 for the optimizations barith2_optim1 mul (square), barith2 mul (circle), portable mul (triangle).

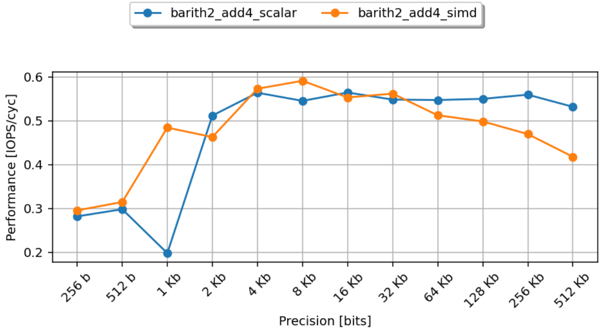

Fig. 13. Scalar vs. batched addition of four *apbar2*: no significant speedup, but would likely benefit from AVX512

6. CONTRIBUTIONS OF TEAM MEMBERS

Kouroche.

- Initial implementation of arbitrary-precision integer structure and operations: addition, shifting, bookkeeping. First implementation used `ADCX` intrinsics (portable version came after).
- Initial implementation of arbitrary-precision floating-point structure and addition.
- Creation of benchmarking framework.
- Prototype for change to middle-alignment in *apfp*.
- Full implementation and unit-testing of the "collapsed" version in `apbar2.h`. Also made some small ILP optimizations there.
- Implementation of SIMD (AVX2) four ball batched addition in `apbar2.h`.

Gyorgy.

- Initial implementation and fixes for apbar (ball arithmetic)
- Creation of unit testing framework and a lot of testing
- Multiplication and ball radius optimizations
- Benchmarking and performance/speedup plot generation
- Portable versions of underlying integer operations
- Middle alignment implementation in apfp (add and sub)

Akanksha.

- Implementation of ball arithmetic addition subtraction and apfp (arbitrary precision floating point) functions.
- Profiling and optimizations for ball arithmetic addition subtraction uncollapsed version
- Optimizations for ball arithmetic addition subtraction collapsed version
- Optimizations for the arbitrary precision integer functions shift, add, subtract, detectfirstset
- Roofline analysis for the ball arithmetic functions
- Basic sanity checking of the overall implementation
- Support for negative numbers across stack in the uncollapsed version

Julia.

- Implementation of arbitrary-precision integer subtraction and multiplication.
- Implementation of multiplication with Karatsuba algorithm.
- Profiling and benchmarking Karatsuba algorithm.
- Karatsuba optimizations.
- Multiplication optimizations.

7. REFERENCES

[1] Fredrik Johansson, “Arb: Efficient arbitrary-precision midpoint-radius interval arithmetic,” 2016.

[2] Joris Hoeven, “Ball arithmetic,” 06 2010.

[3] Ofman Karatsuba, “Multiplication of many-digital numbers by automatic computers,” 1962.

[4] David Defour and Florent De Dinechin, “Software Carry-Save for Fast Multiple-Precision Algorithms,” Research Report LIP RR-2002-08, Laboratoire de l’informatique du parallélisme, Feb. 2002.