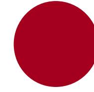
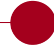



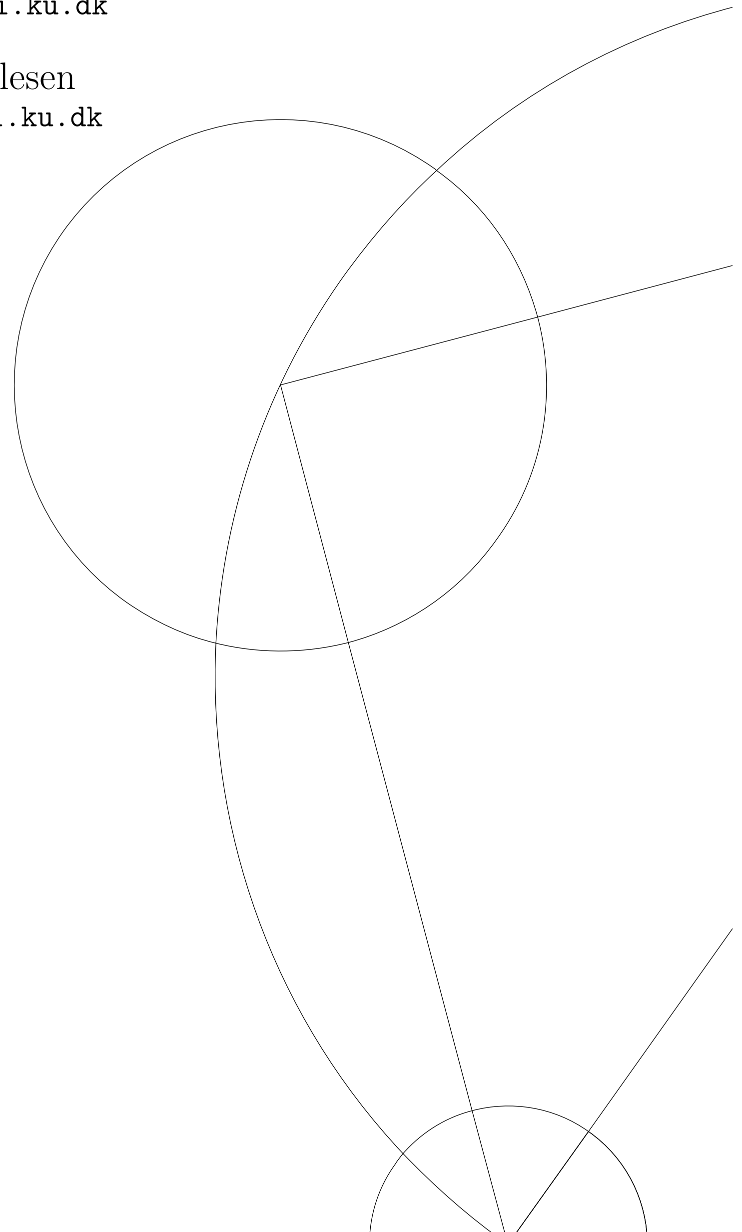
Data Parallel Programming B2-21/22
Multiple-precision Integer Arithmetic

Multiple imprecise programmers

Amar Topalovic
hck338@alumni.ku.dk

Walter Restelli-Nielsen
sdb472@alumni.ku.dk

Kristian Olesen
mjt368@alumni.ku.dk



Contents

1 Introduction 3

2 Practical matters 3
- 2.1 Project structure 3
- 2.2 Python interface 3
- 2.3 Building, testing and benchmarking 3

3 Internal representation 4
- 3.1 Signed integers 4

4 Addition 4
- 4.1 Implementation 4
- 4.2 Neutral element 5
- 4.3 Proof of associativity 5

5 Sum 6

6 Multiplication 6

7 Division 8
- 7.1 Division with a regular precision divisor 9

8 Other functionality 10
- 8.1 Comparison 11
- 8.2 Count leading zeros 11
- 8.3 Displaying big integers 12

9 Tests 12
- 9.1 Futhark tests 12
- 9.2 Python tests 13

10 Benchmarks 14
- 10.1 Addition 14
  - 10.1.1 Single number 14
  - 10.1.2 Multiple 3200-bit numbers 14
- 10.2 Multiplication 15
  - 10.2.1 Single number 15
  - 10.2.2 Multiple 320-bit numbers 15
- 10.3 Division 15
- 10.4 Small_division 15
- 10.5 Sum 15
- 10.6 Other tests 16

11 Discussion 16

12 Improvements 16

13 Conclusion 17

1 Introduction

In this project, we attempt to solve the problem of computing with higher precision integers. More precisely, our goal is to create a Futhark implementation of integers of arbitrary precision; not to be confused with unbounded precision integers. The implementation enables the user to work with $32n$-bit precision integers, for any n (up to the allowed maximum array size in Futhark). We will refer to these as *big integers* or *bigints* going forward.

Our goal is to implement a number of operations for these high precision integers, in particular, addition, multiplication and division. We will go through the implementation of these three in the following sections. We will briefly mention a few other operations in Section 8.

Existing libraries, in other languages, for this purpose, include GMP, which handles unbounded precision integers sequentially, and CGBN, which is made to handle many fixed precision integers of up to $32k$ bits of precision. One of the strategies employed in the development in CGBN is reportedly to confine the parallelism of computations on these integers to small groups of threads, meaning that the potential parallelism is not fully exploited. If enough of these computations are made in parallel, applying this strategy has a reported speedup factor of about 5-6, according to their own benchmarks. For this project, however, we will focus on the use-case involving computations with few, incredibly large numbers. Setting this to be the project focus allows us to focus on making simpler algorithms that work well in parallel, rather than implementing state of the art sequential algorithms with the aim of running these in large batches. Whether or not this use-case is in demand, or if this is even viable given the current strengths and weaknesses of current available machines, we do not know. What we do know is that choosing this focus will involve more of the course topics pertaining to Futhark programming and reasoning in terms of parallel algorithms.

2 Practical matters

2.1 Project structure

- The root folder contains the files `addrs.fut sumrs.fut mults.fut divrs.fut`, the core source code for additions, sums, multiplications and divisions respectively. We can also find a Makefile where shortcuts to benchmark and test are defined, and `bigint.fut` containing the module `bigu32` that defines an API for our bigint type.
- The folder `bench/` contains a number of `.fut`-files intended to be run with `make bench_{add|mul|sum|div}` that use Futhark's random data generator and `futhark bench` tool to benchmark.
- The folder `tests/` contains a number of Python-scripts and Futhark programs testing various properties of the implemented functions, such as commutativity for multiplication. We can also find the subfolders `gmp/` and `CGBN/`, containing benchmarking code for the libraries we compare performance to.

2.2 Python interface

Compiling with the `pyopencl` Futhark backend to a library our bigint API is callable from a python interface `interface.py` that contains a `bigint` class that converts to/from the representation of numbers used in the Futhark program and Python `ints`. As a default it just loops forever testing the implemented operations with random data.

2.3 Building, testing and benchmarking

Building the futhark source can be done with

```
make build
```

Running the Futhark tests and (some of) Python test suite can be done with

```
make test
```

and executing the Futhark benchmarks can be done with

```
make bench
```

To run the rest of the Python tests (an infinite loop of tests) can be done with

```
make python
```

3 Internal representation

We have chosen to represent our bigints in Futhark as arrays of `u32`-values, effectively representing any number D in an array A of length n as $D = \sum_{i=0}^{n-1} A[i] \cdot 2^{32 \cdot i}$. In other words, each entry in the array will represent a digit in base 2^{32} of the combined number. Thus any desired fixed precision can be set in increments of 32 bits by choosing the length of the array (number of digits) accordingly. Encoding the numbers in this way allows for the optimal space efficiency of the representation of numbers in hardware, and has the benefit of being able to take advantage of built-in arithmetic operations. The downside to this approach is that numbers in this representation are not easily representable as in a base 10 format, though algorithms for this are not hard to implement.

3.1 Signed integers

Our main focus has been on unsigned integers, but a `bigu32_signed`-module can be found in `bigint.fut`, implementing some rudimentary functionality using two's complement as a way of representing negative numbers.

4 Addition

A problem with implementing a parallel algorithm for addition is that fundamentally, the result of adding two numbers of one digit can have an effect on the next digit, which in turn can influence the number in the sequence. A simple example of this would be for base 10: adding the numbers 1 and 9999, the carry computed for the first digits affect the rest of the number. Leaving the possibility of a fully parallel algorithm aside, we have explored the possibility of exploiting some levels of parallelism from addition in the following section.

4.1 Implementation

Addition is implemented with the following code:

Listing 1 Implementation of addition

```
1  let add_op (a : (bool, bool)) (b : (bool, bool)) : (bool, bool) =
2      let (ov1, mx1) = a
3      let (ov2, mx2) = b
4
5      let ov = (ov1 && mx2) || ov2
6      let mx = mx1 && mx2
7
8      in (ov, mx)
9
10 let add32 [n] (a : [n]u32) (b : [n]u32) : [n]u32 =
11   let (res, cs) = unzip <|
12     map2 (\x y -> let xy = x + y in (xy, (xy < x, xy == u32.highest))) a b
13   let (carries, _) = unzip <| rotate (-1) <| scan add_op (false,true) cs
14   let carries[0] = false -- overflow doesn't loop
15   in map2 (\x f -> x + u32.bool f) res carries
```

First, each of the digits are individually added together in a map and we keep track of which digits overflow and which digits are one off from overflowing. The latter two properties are then used in a scan to compute the results of carries cascading. The result of the scan is then rotated to line up where the carries need to be added to the final result in the map on the last line.

The way the scan operator keeps track of which values to overflow is as follows:
For combining two elements a and b into $a \oplus b$ to result in an overflow, either a overflows and this causes b to overflow, because it was only one off overflowing, or b is already overflowing. This can be seen in line 5.
Generalizing to two arbitrarily sized consecutive sections of the input number $(\oplus \sum_{j\in 0..i}^{i \in 0..n} a_j) \oplus (\oplus \sum_{k=i+1}^{d \in i+1..n} b_k)$: For b_d to overflow, it must either already have overflowed or a_i overflows and causes a cascading overflow such that values at indexes $i+1$ up to d overflow as well. For the latter scenario to happen, all of the values from index $i+1$ up to d must be equal to the highest digit, which is kept track of in line 6 by `&&`-ing the max digit properties.

The scan dominates the complexity of all the other operations, resulting in a span complexity of $\mathcal{O}(\log N)$ and work complexity of $\mathcal{O}(N)$ as the operator used is $\mathcal{O}(1)$.

4.2 Neutral element

The element `(false, true) = (0,1)` can be shown to be a (left-)neutral element for our scan operator $\oplus$ with the following exhaustive evaulation:

```
(0,1) op (0,0) = (0 && 0) || 0, 1 && 0 = (0,0)
(0,1) op (0,1) = (0 && 1) || 0, 1 && 1 = (0,1)
(0,1) op (1,0) = (0 && 0) || 1, 1 && 0 = (1,0)
(0,1) op (1,1) = (0 && 1) || 1, 1 && 1 = (1,1)
```

4.3 Proof of associativity

$$\left(\begin{bmatrix} o_1 \\ m_1 \end{bmatrix} \oplus \begin{bmatrix} o_2 \\ m_2 \end{bmatrix}\right) \oplus \begin{bmatrix} o_3 \\ m_3 \end{bmatrix} = \begin{bmatrix} (o_1 \wedge m_2) \vee o_2 \\ m_1 \wedge m_2 \end{bmatrix} \oplus \begin{bmatrix} o_3 \\ m_3 \end{bmatrix} \tag{1}$$

$$= \begin{bmatrix} (((o_1 \wedge m_2) \vee o_2) \wedge m_3) \vee o_3 \\ (m_1 \wedge m_2) \wedge m_3 \end{bmatrix} \tag{2}$$

$$= \begin{bmatrix} (((o_1 \wedge m_2) \wedge m_3) \vee (o_2 \wedge m_3)) \vee o_3 \\ m_1 \wedge (m_2 \wedge m_3) \end{bmatrix} \tag{3}$$

$$= \begin{bmatrix} ((o_1 \wedge (m_2 \wedge m_3)) \vee (o_2 \wedge m_3)) \vee o_3 \\ m_1 \wedge (m_2 \wedge m_3) \end{bmatrix} \tag{4}$$

$$= \begin{bmatrix} o_1 \\ m_1 \end{bmatrix} \oplus \begin{bmatrix} (o_2 \wedge m_3) \vee o_3 \\ m_2 \wedge m_3 \end{bmatrix} \tag{5}$$

$$= \begin{bmatrix} o_1 \\ m_1 \end{bmatrix} \oplus \left(\begin{bmatrix} o_2 \\ m_2 \end{bmatrix} \oplus \begin{bmatrix} o_3 \\ m_3 \end{bmatrix}\right) \tag{6}$$

Steps 1 and 2 are two applications of the operator definition. Step 3 is distributing m_3 in the first row of the vector and moving the parentheses by the associative property of $\wedge$ in the second row. Step 4 is moving the innermost parenthesis in $((\mathbf{o_1} \wedge \mathbf{m_2}) \wedge \mathbf{m_3})$ to $(\mathbf{o_1} \wedge (\mathbf{m_2} \wedge \mathbf{m_3}))$ by associativity. Steps 5 and 6 are reverse applications of the operator definition, with parenthesis placed around the final reverse application.

5 Sum

Summation of big integers can be implemented trivially as a reduction using the implemented addition. This encounters the problem that reduce does not exploit parallelism within the operator, meaning that the span for an addition operation would become $O(N \cdot log(M))$ for M vector integers of size N.
Two ideas can be employed to make the algorithm more parallel: The first idea is to develop a form of addition based on "delay carry addition", which takes two integers a, b of a certain base and returns $(a + b, a + b\%B)$. This ostensibly transforms the problem from a `reduce add` into a `add <| reduce map`. The second idea is to move the map operation outside of the reduce, to `map reduce`, to fully exploit the parallelism of the problem. This can be done by transposing the arrays and can be seen in the code in Listing 2.

Listing 2 Implementation of `sum32_alt`

```
let sum32_alt [n][m] (xs: [m][n]u32) : [n]u32 =
  let xst = transpose xs
  let op = \(s1, c1) (s2, c2) ->
       let s  = s1 + s2
       let of = u32.bool (s < s1)
       in (s, c1 + c2 + of)
  let add_rem_f = \ys -> map (\y -> (y, 0)) ys
  let ne = (0u32, 0u32)
  let (sum, car) = map (reduce op ne <-< add_rem_f) xst |> unzip
  in add_shl sum car
```

This algorithm has a span of complexity $O(log(N) + log(M))$ from applying `map reduce` and addition, as well as work in the order of $O(N \cdot M)$ as before. This solution only works if M is less than the 32bit range of 4 billion, which is deemed a sufficiently large number for most applications.

6 Multiplication

As with the other operations, there are two aspects to consider, namely, how to implement the operations correctly, and how to leverage all the potential parallelism we can.

The idea behind our implementation of multiplication is to reduce the problem to regular multiplication of single precision integers and addition of big integers. This is done in the natural way with *long multiplication* (or *grade-school multiplication*) in base 2^{32}, that is, by realising that, if we have two non-negative integers, a and n, below $2^{32 \cdot n}$, written in base 2^{32} as

$$a = \sum_{i=0}^{n-1} a_i 2^{32 \cdot i} \qquad \text{and} \qquad b = \sum_{j=0}^{n-1} b_j 2^{32 \cdot j}, \tag{7}$$

for $a_i, b_i \in \{0, 1, \ldots, 2^{32 \cdot n} - 1\}$, for all $i = 0, 1, \ldots, n-1$, then their product is given by:

$$a \cdot b = \sum_{k=0}^{n-1} \Big(\sum_{i+j=k} (a_i \cdot b_j) \Big) 2^{32 \cdot k}$$

In this way we only need to be able to multiply 32-bit numbers without loss of precision, and, luckily, this is easily done in Futhark. Such as product will at most take up 64-bit, and while regular multiplication returns the lowers 32-bit, then `u32.mul_hi` will return the upper 32-bit. If we let u and l denote functions that return the lower 32-bit and the upper 32-bit respectively, then the product can be written as

$$a \cdot b = \sum_{k=0}^{n-1} \Big(\sum_{i+j=k} l(a_i \cdot b_j) + \sum_{i+j=k-1} h(a_i \cdot b_j) \Big) 2^{32 \cdot k}. \tag{8}$$

There are two important observations to take away from this equation:

- All the coefficients are 32-bit numbers, so they can be calculated with the built-in multiplication. Thus, whether the result is correct comes down to whether we can add the numbers correctly.
- There are no dependencies between the terms in the summation, so we can calculate the values $l(a_i \cdot b_j)$ and $h(a_i \cdot b_j)$ in parallel. Thus, exploiting potential parallelism comes down to doing a good job during addition, which is simpler.

Now, we have tried several implementations of multiplications, and they all try to calculate the multiplications, high and low parts, in parallel and then add these, as we said. We do not use the additions we have implemented (and which are described in Section 4), but use similar methods to calculate the additions with overflow.

Listing 3 Implementation of multiplication using `reduce_by_index`.

```
1  let mul_concat [n] (A : [n]u32) (B : [n]u32) : [n]u32 =
2    let map_op (i : i64) (j : i64) =
3      if i + j < n
4      then ( ((A[i] * B[j],          0), i + j     )
5           , ((u32.mul_hi A[i] B[j], 0), i + j + 1 ) )
6      else let w = ((0, 0), -1) in (w, w)
7    let (ys1, ys2) = map (\i -> map (map_op i) (iota n)) (iota n)
8                     |> flatten
9                     |> unzip
10   let k = length ys1 + length ys2
11   let (vs, is) = unzip <| concat_to k ys1 ys2
12   let empty      = replicate n (0,0)
13   let (res, car) = reduce_by_index empty add_op (0,0) is vs |> unzip
14   in add_shl res car
```

One of our implementations, `mul_concat`, is given in Listing 3, and as mentioned, we start by computing the high and low parts of the products. This is done with the function defined in lines 2–6 of Listing 3, which, to a set of indices (i, j) returns the high and the low parts of the product $a_i \cdot b_j$ along with the associated k from the sum (8). The reason for this is that this k is the index in the big integer representation of the product the values corresponds to. Now, we should also mention that, for (i, j) that does not appear in any of the inner sums in (8) we return $((0, 0), -1)$ for both the high and the low , as the index -1 will be out of bounds for the following `reduce_by_index`.

Now, this `reduce_by_index` is done with values given by the concatenation of the lower parts of the products with the higher parts, and indices given by their corresponding indices (the k's). The operator used, `add_op`, is the same as the reduction operator of summation (the one named `op` in Listing 2), which keeps adding numbers and incrementing the carry every time there is an overflow. In the end we are left with the sum without carries and the carries, which we then add together.

When it comes to work and span we can break down the function as follows:

- The nested map on line 7 clearly has work $O(n^2)$ and span $O(1)$, as the work and span of `map_op` is clearly both $O(1)$.
- The two lists `ys1` and `ys2` both have length n^2, so the following concatenation operator has work $O(n^2)$ and span $O(1)$.
- The `reduce_by_index` is a bit complicated. The work is $O(n^2)$ as the operator `redOp` has work and span $O(1)$, but, according to the documentation, best case span is $O(1)$ and worst case span is $O(n^2)$, given the work and span of `redOp`. From the description of `reduce_by_index`,[1]

[1] https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html#939

it seems like the span might be O of the maximum number of updates to a single index times the work of the functions, which in our case is $O(n)$, as we clearly do not update any entry more than n times.

- We know from earlier (see Section 4) that the work and span of `add_shl` is $O(n)$ and $O(\log n)$, respectively.

Tallying up the result, we see that the work and span of the `mul_concat` function is $O(n^2)$ and (probably[2]) $O(n)$, respectively.

Listing 4 Implementation of multiplication using `reduce_stream_per`.

```
let mul_rbs [n] (A : [n]u32) (B : [n]u32) : [n]u32 =
  let op (v : (i64, u32)) : [n](u32, u32) =
    let (i, a) = v
    in map (\j -> let k = j - i
                  let lo = if k >= 0 then a * B[k] else 0
                  let hi = if k >  0 then u32.mul_hi a B[k-1] else 0
                  let s = lo + hi
                  in (s, u32.bool (s < lo))
           ) (iota n)
  let f (i : i64) (as : [i](i64, u32)) : [n](u32, u32) =
    reduce_comm add_partial (replicate n (0,0)) (map op as)
  let (sum, car) = unzip <| reduce_stream_per add_partial f <| zip (iota n) A
  in add_shl sum car
```

Another implementation of multiplication, `mul_rbs`, is given in Figure 4. This one is a bit different, as we do not do all multiplications up front, but rather divide the work into chunks and calculate $(a_i 2^{32 \cdot i}) \cdot b$ for all i and then aggregate the result. The computation of $(a_i 2^{32 \cdot i}) \cdot b$ with carries separate can be done easily in parallel (and is done in `op`) and we can then aggregate the partial results and their carries with the `add_partial` (which is just an `map add_op` for `add_op` in Listing 3). This whole thing can then be accomplished by a `reduce_stream_per` that calculates the $(a_i 2^{32 \cdot i}) \cdot b$ values and then aggregates the results. In the end we just need to add the carries to the partial result we end up with, as previously.

Regarding work and span for `mul_rbs`, we know from the description of `reduce_stream_per`,[3] that the work is $O(n \cdot W(\texttt{add_partial}) + W(\texttt{f}))$ and the span is $O(\log(n) \cdot W(\texttt{add}_{\texttt{p}}\texttt{artial}))$. As mentioned `add_partial` is just `map add_op` for the operator from before, so that its works and span is $O(n)$ and $O(1)$, and it is easy to see that `op` has the same work and span. With this we see that the argument to `reudce_comm` must have work $O(n^2)$ and span $O(1)$, and that the work and span of `f` must therefore be $O(n^2)$ and $O(n \log n)$ (using the guarantees[4] of `reduce_comm`). All in all, we end up with `mul_rbs` having work and span $O(n^2)$ and $O(n \log n)$, respectively.

7 Division

Division is implemented as described in Donald Knuth's book as "algorithm D" (see [1]), which is also referenced as a starting point for further optimization in the GMP library. The algorithm takes two arguments $U = u_0 \cdot B^0 ... u_{n-1} \cdot B^{n-1}, V = v_0 \cdot B^0 ... v_{m-1} \cdot B^{m-1}$, encoded in the usual big integer format as arrays, and returns an array of size $n - m$ with the result $Q = \lfloor \frac{U}{V} \rfloor$. Since Futhark incorporates the lengths of arrays into their types, U, V and Q are of size n in the code.

We have decided to implement the algorithm by taking advantage of 64bit integer division, combining two 32bit array values at a time of the leading digits of U and dividing these by the lead digit of V to form an estimate of q. Ensuring that this estimate fits into a 32 bit integer is done

[2] Probably meaning that we suspect our conclusion above above `reduce_by_index` is correct. Otherwise the span may be $O(n^2)$.

[3] `https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html#1123`

[4] `https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html#925`

by normalizing both U and V in a way that ensures that $v_{m-1} < \frac{2^{32}}{2}$. This normalization step also ensures that the estimate of q is never too low and never more than 2 values off the true value, according to the proofs in the source.
While this approach is sequential and therefore does not gain any advantage from being run on a parallel machine, the presence of a big integer division operator is deemed necessary for the completeness of the library. Each loop iteration has span $O(log(N))$ and does work amounting to $O(N)$ due to the subtraction of two big integers and the multiplication of one big and one 32bit integer, resulting in a combined span and work complexity of $O(N \cdot log(N))$ and $O(N^2)$. Such a result may, on sufficiently parallel hardware, still outperform an implementation in a purely sequentially compiled language if run in parallel for a sufficiently large amount of numbers. Even this, however, is likely to fall short of the performance of the CGBN implementation, which incorporates parts of the GMP library for this approach.

7.1 Division with a regular precision divisor

Dividing by a single precision divisor can surprisingly be done efficiently in parallel. Using the same setup as previously, a number $U = u_0 \cdot B^0 ... u_{n-1} \cdot B^{n-1}$ divided by v, it can be seen that each partial result $q_0...q_{n-1}$ can be computed in parallel as follows:

For $i \in \{0..n-1\}$,

$$q_i = \frac{\left(\sum_{j=i+1}^{n-1} u_j \cdot B^{j-i}\right) + u_i}{v} \% B$$

This accounts for the division of u_i and v added to the remainder of the step before, which also takes into account all previous remainders. Because of the modulo and division operators, it is known that $\frac{x \cdot B}{v} \% B = \frac{x \% v \cdot B}{v} \% B$, since $\frac{x \cdot v \cdot B}{v} \% B = 0$ and therefore

Given any x, we can write $x = qv + (x \% v)$, for some q, and so we see that

$$\frac{x \cdot B}{v} \% B = \frac{q \cdot v \cdot B + (x\%v) \cdot B}{v} \% B = \left(q \cdot B + \frac{(x\%v) \cdot B}{v}\right) \% B = \frac{x\%v \cdot B}{v} \% B$$

This allows for the equation to be rewritten as:

$$q_i = \frac{\left(\sum_{j=i+1}^{n-1} u_j \cdot B^{j-i}\right) + u_i}{v} \% B \tag{9}$$

$$= \frac{\left(\sum_{j=i+1}^{n-1} u_j \cdot B^{j-i-1}\right) \cdot B + u_i}{v} \% B \tag{10}$$

$$= \frac{\left(\left(\sum_{j=i+1}^{n-1} u_j \cdot B^{j-i-1}\right) \% v\right) \cdot B + u_i}{v} \% B \tag{11}$$

$$= \frac{\left(\left(\sum_{j=i+1}^{n-1} u_j \cdot (B^{j-i-1} \% v)\right) \% v\right) \cdot B + u_i}{v} \% B \tag{12}$$

This is a useful result, as it not only shows that each value of q can be computed independently, but also that the difficult part of the computation $B^x \% v$ can be computed in parallel and the partial results of this can be used at each step of the computation. This is best done using a scan operation.

The augmented datatype for the proposed scan includes $B^{x-1} \% v$, also referred to as distance, where x is the amount of array indices reduced. Adding two partial results, comprized of q, r, d for current division estimate, current remainder and distance, gives:

$$\begin{bmatrix} q_1 \\ r_1 \\ d_1 \end{bmatrix} \oplus \begin{bmatrix} q_2 \\ r_2 \\ d_2 \end{bmatrix} = \begin{bmatrix} q_2 + ((q_1 \cdot d_2) \% v \cdot B + r_2)/v \\ (q_1 \cdot d_2 \cdot B + r_2) \% v \\ (d_1 \cdot d_2 \cdot B) \% v \end{bmatrix}$$

Without further analysis, it is clear that the calculation of distances is associative. The calculations for r and q are also associative, since represent a weighted sum of the number $(q \cdot v + r) \% (v \cdot B)$.

For scan to work in Futhark, an extra boolean parameter is added to represent if the value is the neutral element. Combined, the operator can be seen in the following code:

```
let op = \(q1, remainder1, distance1,is_neutral_1) (q2, remainder2, distance2,is_neutral_2) ->
    if is_neutral_1 then (q2, remainder2, distance2,is_neutral_2) else
    if is_neutral_2 then (q1, remainder1, distance1,is_neutral_1) else

    let distance = mul_mod V B <| mul_mod V (u64.u32 distance1)  (u64.u32 distance2)

    let r1d2 = mul_mod V (u64.u32 remainder1) (u64.u32 distance2)
    let remainder_to_add = mul_mod V B r1d2
    let remainder = u32.u64 <| (remainder_to_add + (u64.u32 remainder2)) % V

    let q_to_add = mul_div V B r1d2
    let q_to_add2= (remainder_to_add + (u64.u32 remainder2)) / V
    let q = u32.u64 <| (u64.u32 q2) + q_to_add + q_to_add2

    in (q, remainder, u32.u64 distance, false)
```

Using the straight forwardly declared "mul_mod" and "mul_div" functions.

This operator can be computed in constant time using the 64bit precision operators, giving the algorithm, which mostly consists of a scan, a work and span complexity of $O(N)$ and $O(log(N))$, which is deemed to be a tight bound.

8 Other functionality

Besides addition, multiplication and division, we have implemented a small amount of extra functionality. First of all, we have created a module type to wrap the implemntation in, and to have an API to expose. This can be found in `bigint_type.fut`, and looks something like:

```
module type bigint = {
  type t[n]

  val zero        : (n: i64) -> t[n]
  val one         : (n: i64) -> t[n]
  val highest     : (n: i64) -> t[n]
  val lowest      : (n: i64) -> t[n]

  val +   [n]     : t[n] -> t[n] -> t[n]
  val *   [n]     : t[n] -> t[n] -> t[n]
  val /   [n]     : t[n] -> t[n] -> t[n]
  val %   [n]     : t[n] -> t[n] -> t[n]

  ...
}
```

We then implement this module as a module `bigu32`, which is a bit integer with 32-bit arrays as underlying data-structure, with

```
module bigu32 : (bigint with t [n] = [n] u32) = ...
```

We have already talked about how to implement the last four of the operations seen above in the module type, and the other four are quite easy. Besides this, we have some comparison based operators between big integers, and a function for finding the leading number of zeroes.

8.1 Comparison

All the comparison based operators we have implemented, `<`, `>`, `<=`, `>=`, `==`, `!=`, `min` and `max`, are easily implemented if we given two numbers x and y can determine if $x > y$, if $x = y$ or $x < y$. We have implemented a function for this, namely,

```
type c = #lt | #eq | #gt

let cmp [n] (x: [n]u32) (y: [n]u32) : c =
  let ec = \a b -> if a < b then (#lt: c)
                   else if a == b then (#eq: c)
                   else (#gt: c)
  let cs = map2 ec x y
  in reduce (\c1 c2 -> if c2 == (#eq: c) then c1 else c2) (#eq: c) cs
```

This function finds, by a reduction, the largest digit (base 2^{32}) where the two numbers do not agree, and return the relationship between these digits, unless they are equal in which case it returns `#eq`. This clearly means that if the function return `#lt`, then $x < y$, and similarly with `#gt`

Now, it is not too difficult to see that this is an associative operator with neutral element `#eq`. If we use `#eq` from the left, then `c2 == #eq` is true, as we return the right hand element, and from the right it is also neutral element. For this reason one would only need to check associativity when the operator is applies to a bunch of `#lt` and `#gt`, as the equation is trivial if one of the element is the neutral element. However, if none are `#eq`, then we just return the right-most value, which we know is associative.

With this operator, we can very easily implement all of the mentioned comparisons/operators.

8.2 Count leading zeros

We have also implemented a function for counting the number of leading zeros (bits, so base 2, not base 2^{32}) of a big integer. The function is given by

```
let clz  [n] (x: [n]u32) : i64 =
  let mop = \d -> ( u32.clz d |> u32.i32
                  , d u32.== 0 )
  let rop = \(d1, _) (d2, b2) -> ( if b2 then d1 + d2 else d2
                                 , b1 && b2 )
  in map mop x |> reduce rop (0u32, true) |> (.0) |> i64.u32
```

Now, the intuition of the implementation is as follows. We create an array with the number of leading zeros of each of the digits, as well as a boolean that indicates if the digit was only zeroes. The operator then applies the principle that, given two numbers their juxtaposition is only purely zeros if they are both purely zeros, and the number of leading zeros in this juxtaposition is the number of leading zeros in the right most one (our most significant digits are to to the right), unless this one is entirely zeroes, in which case it the sum of the leading zeros in both numbers.

Like before, it is not too difficult to see that this operator is associative with a neutral element. If we denote the operator by $\odot$, then one of the cases of associativity is

$$\left(\begin{bmatrix} n_1 \\ \top \end{bmatrix} \odot \begin{bmatrix} n_2 \\ \bot \end{bmatrix}\right) \odot \begin{bmatrix} n_3 \\ \top \end{bmatrix} = \begin{bmatrix} n_2 \\ \bot \end{bmatrix} \odot \begin{bmatrix} n_3 \\ \top \end{bmatrix} = \begin{bmatrix} n_2 + n_3 \\ \bot \end{bmatrix}$$

and the other way arraound:

$$\begin{bmatrix} n_1 \\ \top \end{bmatrix} \odot \left(\begin{bmatrix} n_2 \\ \bot \end{bmatrix} \odot \begin{bmatrix} n_3 \\ \top \end{bmatrix}\right) = \begin{bmatrix} n_1 \\ \top \end{bmatrix} \odot \begin{bmatrix} n_2 + n_3 \\ \bot \end{bmatrix} = \begin{bmatrix} n_2 + n_3 \\ \bot \end{bmatrix}$$

which is as it should. The test of the cases are similar.

8.3 Displaying big integers

To convert from our chosen inner representation of big-integers to readable numbers, the basic idea is that each digit in the readable format with position i must be computed as: $D/10^i\%10$. For this project, this is done sequentially using the small division algorithm, though, since finding the remainder of a big-integer and a regular divisor is essentially a reduction, this could also have been done with a segmented reduction. If printing, or converting between internal representations of numbers, were ever to become a performance critical task, an efficient algorithm for this purpose could theoretically be constructed.

9 Tests

We have generally made use of two kinds of test; Futhark tests and Python tests. The reason for including Python tests is that Python has unlimited integer precision, which Python can be tested up against. With the former, we mostly have property testing.

9.1 Futhark tests

As mentioned we mostly do property testing in this part. The part that is maybe not property testing is the tests in `compare.fut`, which tests on a number of inputs that the different versions of addition, multiplication and summation agree. Though having multiple implementations of the same function might not be as relevant from a library point of view, it serves well for testing. Indeed, it might be unlikely that the different implementation are erroneous in the same manner, and then the test might fail. The test is on random input, though, so they might not reach a particular corner case that fails.

Now, in the files `correct_add.fut` and `correct_mul.fut`, we test that addition and multiplication operators are associative and commutative, and that they give the right result on 64-bit integers, as we can test this the build-in versions. Moreover, in `correct_add_mul.fut` we check that multiplication distributed over addition. All these tests are done on random input. We have chosen to test this with small numbers (such as $2^{32\cdot 10}$-bit), as these should be large enough to exhibit problems if there are some, but small anough that we are more likely to get into corner cases.

In the files `correct_add.fut` and `correct_sum.fut` we also have tests that check that the results are correct modulo $2^{32}-1$, as this can be done entirely with the build-in operations. Indeed, given two numbers as in 7, we have that, modulo $2^{32}-1$

$$a + b = \left(\sum_{i=0}^{n-1} a_i 2^{32\cdot i}\right) + \left(\sum_{j=0}^{n-1} b_j 2^{32\cdot j}\right) \equiv \left(\sum_{i=0}^{n-1} a_i\right) + \left(\sum_{j=0}^{n-1} b_j\right) \tag{13}$$

since $2^{32\cdot k} \equiv 1$ modulo $2^{32}-1$ for all k. However, the sum on the left if a sum of 32-bit integers, so if there are not too many of them, then we are guaranteed that the computation can be done with 64-bit integers without overflow. But if the result of our addition ($2^{32\cdot n}$-bit precision) is

$$c = \sum_{i=0}^{n-1} c_i 2^{32\cdot i}$$

then the left hand side of (13) must equal

$$\sum_{i=0}^{n-1} c_i \tag{14}$$

as this is equal to c modulo $2^{32}-1$. Likewise this result can be computed without overflow in a 64-bit integer (when n is reasonably sized), so we must have that the remainders of the left hand side of (13) and the left hand side of (14) by division with $2^{32}-1$ are equal.

Now, the above describes the test for addition, but the test is the same with summation. The point is that we can calculate the remainder before and after addition and it should give the same result. But when doing it before we can use built-in operations.

Lastly, we have also tried in `correct_div.fut` to implement a test that, given two numbers x and y, tests that, if we calculate *our* division/quotient, q, of x by y and the corresponding remainder, r, then, with our multiplication and addition, we should have $x = y \cdot q + r$ and $r < y$. These equations uniquely determines q and r (for positive numbers), assuming that our multiplication and addition are correct. However, for reasons that have escaped us, we have not been able to make these work, though our python tests of the quotients and remainders indicate that the implementation is correct.

9.2 Python tests

As mentioned earlier, the python tests tests that the big integer operations work as they should by testing them up agains the unbounded precission integer arithmetics of Python. There are a few simple tests, in `test_add.py`, `test_div.py` and `test_mul.py`, which reads three Futhark arrays. The two inputs and the output of the given operator (the quotient in case of division), and checks that this matches the corresponding operator in Python. The two input arrays are generated by `futhark dataset` and the last one is generated by running the Futhark implementation with this input and piping the output to a new file. These tests can all be run with `make py-test`.

Now, the above tests run with new random input each time, but only test a single application of the operator. The `interface.py` file, however, will test the operations repeatedly, by use of `pyopencl`. Indeed, running `make python` will start and infinite loop that generates random numbers, calls the python functions, addition, multiplication and division and checks that the results are correct. As opposed to the simple python tests, this also tests the remainder of the division.

10 Benchmarks

All benchmarks are done on an AMD Ryzen 7 4800H processor and an `NVidia RTX 2060` graphics card in a laptop running Ubuntu 20.04 LTS. All times are in μs. Futhark benchmarks are executed with `futhark bench` using the `cuda` backend.

The `gmp` benchmarking code can be found in `bench/gmp/`.
The `CGBN` benchmarking code can be found in `bench/CGBN/benchmarks/{add|mul|div}` and built+run with `make <nvidia_architecture_name> && ./{add|mul|div}` given all dependencies are installed. `CGBN` is limited to 32k bits of precision, so any benchmarks with higher precision leaves these fields blank.

10.1 Addition

10.1.1 Single number

bits	bigint.fut add_ind	GMP	CGBN	speedup vs GMP	speedup vs CGBN
3200	32	1	3	0.03x	0.09x
32k	22	1	8	0.05x	0.36x
320k	29	2		0.06x	
3.2m	32	23		0.72x	
32m	168	6076		36.17x	
320m	1018	60352		59.28x	

Table 1: Adding two numbers, varying bits of precision

10.1.2 Multiple 3200-bit numbers

# of adds	GMP	bigint.fut	CGBN
10	1	16 (44)	3
100	4	15 (45)	11
1000	44	27 (67)	14
10k	176	143 (260)	64
100k	1753	1333 (1120)	689
1m	17513	16109 (14020)	5940
10m	175982	err	mem

Table 2: Varying number of pairs of values added. Times in parenthesis are from an alternative version (add_bit_small) of add that performed notably well

10.2 Multiplication

10.2.1 Single number

bits	times(fut)	times(GMP)	times(CGBN)	speedup vs GMP	speedup vs CGBN
320	95	1	3	0.01x	0.03x
3200	94	1	7	0.01x	0.07x
32k	203	28	497	0.13x	2.44x
320k	18646	797		0.04x	

Table 3: Multiplying two numbers, varying bits of precision

10.2.2 Multiple 320-bit numbers

# of mults	bigint.fut	GMP	CGBN
10	80	2	3
100	68	5	3
1000	218	53	5
10k	197	468	30
100k	2720	1856	217
1m	26581	18649	1835
10m		186325	14557

Table 4:

10.3 Division

bits	div (fut)	div(GMP)	div(CGBN)
320	1455	2	4
3200	6954	1	6
32k	53226	22	26
320k	504933	722	

Table 5:

10.4 Small_division

Div	Futhark-good	Futhark-old	GMP
10	97	1549	1
100	97	11860	1
1k	121	86564	1
10k	123	846837	14
100k	238	8438497	145
1m	481	err	1462
10m	3951	err	14568

Futhark-old run significantly better with opencl, running the 10 million case in 9 seconds

10.5 Sum

100 numbers

size of number *32	sum_alt	sum
10	47	84
100	29	58
1k	36	69
10k	115	228
100k	963	1246
1m	8650	11239

10.6 Other tests

A good way of comparing the futhark implementation with the other libraries would be to design a benchmark that combined operations on big-integers. The functions for fibonacci, factorial and gcd have been coded for Futhark and compared with GMP, but the results are not reported.

11 Discussion

The results of the benchmarking show that the algorithms that have been made parallel while not increasing the underlying work, our implementation performs better than the existing libraries for sufficiently large numbers. Specifically, these are addition and small division (though small division has not been tested in CGBM). Regular division, which we have implemented sequentially, performs very poorly against both CGBM and GMP, which we reason has to do with finding a better algorithm and optimizing its implementation at a low level.

Other algorithms, for which we have found parallel implementations that increased the complexity of the work done, struggle to find a competitive footing against the established big-integer libraries. If run for arrays of large integers in parallel, the low work complexity algorithms run at comparable speeds to the CGBM big-integers and slightly faster than purely sequential additions.

12 Improvements

Another optimization approach, utilized by GMP, is to use multiple version of the same algorithms for different circumstances. For example, in the case of division by precise powers of 2, the problem can be computed as a bit-shifting operation. Sequentially, this can easily be coded as a conditional statement calling different functions.
These types of optimizations are harder to exploit in a parallel environment since a group of threads encountering a conditional statement may need to execute both branches sequentially to compute both results. This seems to be avoided in CGBN by limiting a group of threads to the processing of a single number, but is not easily replicable in the more abstract language of Futhark. There are, however, algorithms in GMP which take advantage of any information known by the programmer at the time of writing the program, and use this information to pick the algorithm to employ. These algorithms can then take advantage of this information, by, for example, being able to employ a division algorithm that only works when numbers divide without remainders.
A possible improvement strategy for this, and any other library, can always be to implement the library directly into the compiler, taking advantage of more complex compiler primitives and low level optimizations. While the goal of this project has been to develop a library to do parallel operations on few, extremely large big-integers, if the library is to be made to handle parallel operations of many smaller sized big-integers, a better strategy is to take the GCBN approach. As discussed, some of the design decisions in the project have prioritized optimizing span over maintaining work efficiency, reducing their performance greatly if run sequentially or when the parallelism of the underlying machines is exceeded, which they likely would be in this use-case. For this, sequential, highly work-optimized algorithms, such as the ones found in the GNU library can be seen outperforming our implementation. This form of parallelism is, however, more technically difficult to implement and would touch on few aspects of the course.

13 Conclusion

In conclusion, we have implemented a big-integer library for Futhark and compared it to existing libraries for other programming languages. We have tested these against themselves and with python scripts. This, as well as the use of sound mathematical reasoning documented in this report make us reasonably confident in the implementation's correctness. Because of our focus few, large numbers, this library can outperform the other libraries in some arithmetic operations, while being woefully uncompetitive with others.

References

[1] *The Art Of Computer Programming, Volume 2: Seminumerical Algorithms, 3/E.* Number vb. 2. Pearson Education, 1998. ISBN 9788177583359.